下载APP

07 | 动态链接（上）：地址无关代码是如何生成的？

2021-11-08 海纳

《编程高手必学的内存知识》

课程介绍 >

讲述：海纳

时长 23:07 大小 21.17M

你好，我是海纳。

通过上节课的学习，我们了解到，链接器可以将不同的编译单元所生成的中间文件组合在一起，并且可以为各个编译单元中的变量和函数分配地址，然后将分配好的地址传给引用者。这个过程就是静态链接。

静态链接可以让开发者进行模块化的开发，大大的促进了程序开发的效率。但同时静态链接仍然存在一个比较大的问题，就是无法共享。例如程序 A 与程序 B 都需要调用函数 foo，在采用静态链接的情况下，只能分别将 foo 函数链接到 A 的二进制文件和 B 的二进进

制文件中，这样导致系统同时运行 A 和 B 两个进程的时候，内存中会装载两份 foo 的代码。那么如何消除这种浪费呢，这就是我们接下来两节课的主题：动态链接。

动态链接的重定位发生在加载期间或者运行期间，这节课我们将重点分析加载期间的重定位，它的实现依赖于地址无关代码。我们知道，深入地掌握动态链接库是开发底层基础设施必备的技能之一，如果你想要透彻地理解动态链接机制，就必须掌握地址无关代码技术。

在你掌握了地址无关代码技术后，你还将对程序员眼中的“风骚”操作，比如，如何通过重载动态库对系统进行热更新，如何对动态库里的函数进行 hook 操作，以便于调试和追踪问题等等，都会有更深入的理解。

我们先来一起看一下，动态链接是怎样解决静态链接不能充分共享代码这个问题的。

什么是动态链接

要想解决静态链接的问题，可以把共享的部分抽离出来，组成新的模块。为了让一些公共的库函数能够被多个程序，在运行的过程中进行共享，我们可以让程序在链接和运行过程中，也拆分成不同的模块，即共享模块和私有模块。共享模块用来存放供所有进程公共使用的库函数，私有模块存放本进程独享的函数与数据。

分析到这里，动态链接的基本思路就呼之欲出了。目前解决共享问题，采用的通用的思路是，将常用的公共的函数都放到一个文件中，在整个系统里只会被加载到内存中一次，无论有多少个进程使用它，这个文件在内存中只有一个副本，这种文件就是动态链接库文件。

它在 Linux 里是共享目标文件 (share object, so)，在 windows 下是动态链接库文件 (dynamic linking library, dll)。当然，以上只是一个最基本的想法，要想真正实现动态链接的技术还有很多问题需要考虑。接下来，我们来看最主要的两个问题。

第一个问题是，由于公共库函数的代码要在多个不同的进程中进行共享，也就是说，不同的进程运行的库的代码是同一份，这就要求共享模块的代码必须是地址无关的，因为每个进程都有自己独立的内存空间，系统 loader 无法保证共享模块加载的内存地址，对于每个进程而言都是相同的地址。

例如进程 A 加载的 libfoo.so 的起始地址可能是 0x1000，而进程 B 加载的 libfoo.so 的起始地址可能是 0x3000，如果 libfoo.so 里代码访问的函数或者数据是绝对地址的话，那必然会造成进程 A 与 B 的冲突。

第二个问题是，我们知道，虽然在开发的过程中，开发者可以将程序模块化处理，但还是需要静态链接来将不同模块链接到一起，对符号进行重定位，这样运行时 CPU 才能知道各个函数、变量的真正地址是什么。

同样的，要想让程序在运行过程中也进行模块化，那就意味着，不同模块之间符号的链接过程，需要推迟到加载时进行了，这也是动态链接 (Dynamic Linking) 技术名字的由来。

在讲解动态链接的具体实现之前，我们还是先来看下动态链接的小例子，来对动态链接有一个初步的印象。

如何生成和使用动态链接库

我们通过运行一个例子来展示动态链接和加载的完整过程：

```
1  // foo.h
2  #ifndef _FOO_H_
3  #define _FOO_H_
4
5  void foo();
6
7  #endif
8
9  // foo.c
10 #include <stdio.h>
11 #include "foo.h"
12 void foo() {
13     printf("Hello foo\n");
14 }
15
16 // main_a.c
17 #include <stdio.h>
18 #include "foo.h"
19
20 int main() {
21     printf("A.exe: ");
22     foo();
23     while(1) {
24     }
```

```
25 }
26
27 // main_b.c
28 #include <stdio.h>
29 #include "foo.h"
30
31 int main() {
32     printf("B.exe: ");
33     foo();
34     while(1) {
35     }
36 }
```

以上例子分了三个模块，分别是共享模块的 foo.c，两个主程序 main_a.c 和 main_b.c，主程序都调用了 foo.c 中的 foo 方法，最后放一个死循环用来保证程序不退出，以便于查看进程的相关信息。

我们先把 foo.c 编译成 libfoo.so：

```
1 $ gcc foo.c -fPIC -shared -o libfoo.so
```

其中 -fPIC 目的是开启地址无关代码，一会儿我会给你详细解释；-shared 意思是告诉链接器生成的目标文件是共享目标文件。

然后我们分别编译 main_a.c 和 main_b.c，来生成可执行文件 A.exe 和 B.exe：

```
1 $ gcc main_a.c -L. -lfoo -no-pie -o A.exe
2 $ gcc main_b.c -L. -lfoo -no-pie -o B.exe
```

我先来解释下，这块代码中几个选项的意思。

-L 指定了查找链接库的路径 (或者可以通过设置环境变量 LIBRARY_PATH 来追加路径)。-L. 就是告诉链接器需要到当前目录下查找共享文件。

-l 则指定了具体链接库的名称，需要注意的是，gcc 在处理链接库名称时，会自动加上 lib 的前缀和.so 的后缀，所以，gcc 命令选项写的 -lfoo，就是告诉链接器查找

libfoo.so 这个共享目标文件。

-no-pie 是禁止生成地址无关的可执行文件，方便我们查看进程的内存布局。

此时我们执行“ldd A.exe”或者“ldd B.exe”的时候，就可以看到两个可执行文件依赖的 so 中多了一个 libfoo.so:

```
$ ldd A.exe
        linux-vdso.so.1 (0x00007ffebc5ed000)
        libfoo.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f07e5ffe000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f07e65f1000)
```

我要提醒你的是，上面的命令在输出过程中，libfoo.so 的指向是 not found。这是因为 libfoo.so 所在的路径是当前路径，运行时查找共享库的时候默认并不会来找寻当前路径，因此 libfoo.so 的指向目前是无法确认的。如果此时执行./A.exe 同样也会报错，解决方法就是将当前路径设置到 LD_LIBRARY_PATH 的环境变量中。

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

此时再执行`ldd A.exe`就能找到 libfoo.so 的位置了。运行结果如下所示：

```
$ ldd A.exe
        linux-vdso.so.1 (0x00007ffef1cba000)
        libfoo.so => ./libfoo.so (0x00007fe9d6998000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe9d65a7000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe9d6d9c000)
```

在上面的例子中，我们提到了两个环境变量 LIBRARY_PATH 和 LD_LIBRARY_PATH，你可能对这两个环境变量的作用不是很清楚，或者容易混淆，在这里我们再对这两个变量的作用进行一下对比区分。这两个环境变量都是用来设置库文件的查找路径的，只不过使用的时机不一样。

其中 LIBRARY_PATH 是由链接器来使用的，一般系统默认是 gnu ld。对于大部分开发者来讲，如果 LIBRARY_PATH 没有设置好，在使用 gcc 或者 clang 这些编译器（其实它们都调用了 ld 这个链接器，真正做事情的是 ld）的时候，会碰到类似`/usr/bin/ld: cannot find -lfoo`的错误。LIBRARY_PATH 的一个等价的选项就是上文讲的 -L 指定路径的选项。

而另一个 LD_LIBRARY_PATH 的环境变量是由动态链接器来使用的，即我们通过 ldd 看到的 ld-linux-x86-64.so.2 这个库。动态链接器的知识我们会在下一节课中详细展开。目前这里，我们只需要知道这个动态链接器是在程序加载运行时执行的就可以了。

因此，如果 LD_LIBRARY_PATH 没有设置好的话，会碰到类似`./A.exe: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory`的问题。

总结下来，LIBRARY_PATH 的使用时机是链接器在做链接的时候，LD_LIBRARY_PATH 的使用时机是在程序运行时。

接下来我们再看一下可执行程序运行起来以后，它的内存布局是什么样子的，这样我们就能清楚动态链接技术是怎么节省内存的。

动态链接库内存布局

我们通过执行两个进程，一起看下它们的内存布局：

复制代码

```
1  $ ./A.exe &
2  $ ./B.exe &
3  $ cat /proc/`pidof A.exe`/maps
4  00400000-00401000 r-xp 00000000 08:10 747270         ./A.exe
5  00600000-00601000 r--p 00000000 08:10 747270         ./A.exe
6  00601000-00602000 rw-p 00001000 08:10 747270         ./A.exe
7  01e58000-01e79000 rw-p 00000000 00:00 0              [heap]
8  …
9  7fb25b13d000-7fb25b141000 rw-p 00000000 00:00 0
10 7fb25b141000-7fb25b142000 r-xp 00000000 08:10 747268 ./libfoo.so
11 7fb25b142000-7fb25b341000 ---p 00001000 08:10 747268 ./libfoo.so
12 7fb25b341000-7fb25b342000 r--p 00000000 08:10 747268 ./libfoo.so
13 7fb25b342000-7fb25b343000 rw-p 00001000 08:10 747268 ./libfoo.so
14 …
15 7ffed501b000-7ffed503c000 rw-p 00000000 00:00 0      [stack]
```

```
16 7ffed51bc000-7ffed51c0000 r--p 00000000 00:00 0      [vvar]
17 7ffed51c0000-7ffed51c1000 r-xp 00000000 00:00 0      [vdso]
18
19 $ cat /proc/`pidof B.exe`/maps
20 00400000-00401000 r-xp 00000000 08:10 747269         ./B.exe
21 00600000-00601000 r--p 00000000 08:10 747269         ./B.exe
22 00601000-00602000 rw-p 00001000 08:10 747269         ./B.exe
23 01597000-015b8000 rw-p 00000000 00:00 0              [heap]
24 …
25 7f2991e85000-7f2991e89000 rw-p 00000000 00:00 0
26 7f2991e89000-7f2991e8a000 r-xp 00000000 08:10 747268 ./libfoo.so
27 7f2991e8a000-7f2992089000 ---p 00001000 08:10 747268 ./libfoo.so
28 7f2992089000-7f299208a000 r--p 00000000 08:10 747268 ./libfoo.so
29 7f299208a000-7f299208b000 rw-p 00001000 08:10 747268 ./libfoo.so
30 …
31 7f29922b6000-7f29922b7000 rw-p 00000000 00:00 0
32 7fff73f9e000-7fff73fbf000 rw-p 00000000 00:00 0      [stack]
33 7fff73fde000-7fff73fe2000 r--p 00000000 00:00 0      [vvar]
34 7fff73fe2000-7fff73fe3000 r-xp 00000000 00:00 0      [vdso]
```

从上面的命令运行结果中，我们可以观察到这样两个特点：

第一个特点是，动态库的数据段和代码段是靠在一起的，它并没有和可执行程序的数据段，代码段分别合并，这是与静态链接不同的地方。

第二个特点是，同一个动态库文件在两个进程中的虚拟地址并不相同，A.exe 跟 B.exe 同时加载了 libfoo.so，但所处的位置分别是 0x7fb25b141000 与 0x7f2991e89000，并不相同。

然后我们再看一下两个进程中 libfoo.so 代码段的物理内存占用情况，先看 A 进程的：

```
1 $ cat /proc/`pidof A.exe`/smaps
2 7fb25b141000-7fb25b142000 r-xp 00000000 08:10 747268 ./libfoo.so
3 Size:                  4 kB
4 KernelPageSize:        4 kB
5 MMUPageSize:           4 kB
6 Rss:                   4 kB
7 Pss:                   2 kB
8 ......
```

再看看 B 进程的：

```
$ cat /proc/`pidof B.exe`/smaps
7f2991e89000-7f2991e8a000 r-xp 00000000 08:10 747268 ./libfoo.so
Size:                  4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                   4 kB
Pss:                   2 kB
.......
```

我们通过 smap 的结果来考察物理内存的实际占用情况。在第 1 节课的练习中，我曾经让你自己动手研究 smap 各字段的含义，今天我们来重点分析一下 Rss 与 Pss。Rss 的含义是当前段实际加载到物理内存中的大小，Pss 指的是进程按比例分配当前段所占物理内存的大小。

在这个例子中，因为 libfoo.so 本身代码段不足 4K，但是物理页的单位是 4K，所以这里 libfoo.so 代码段本身需要占据一个物理页，也就是 4K 的大小，即 Rss 值为 4K。

由于多个进程共享了动态库，所以 Pss 的计算方式应该是 Rss 值除以共享进程数。从上面例子可以看到，A.exe 与 B.exe 共享了 libfoo.so 的代码段，按比例分配的话应该分别占用 2K，即 Pss 的值都是 2K。如果此时我们把 B.exe 进程终止掉，你会发现 A.exe 这里的 Pss 值就会变成 4K。命令的输出还包含其他字段，如果你感兴趣的话，可以通过`man proc`命令来查询 proc 的详细信息。

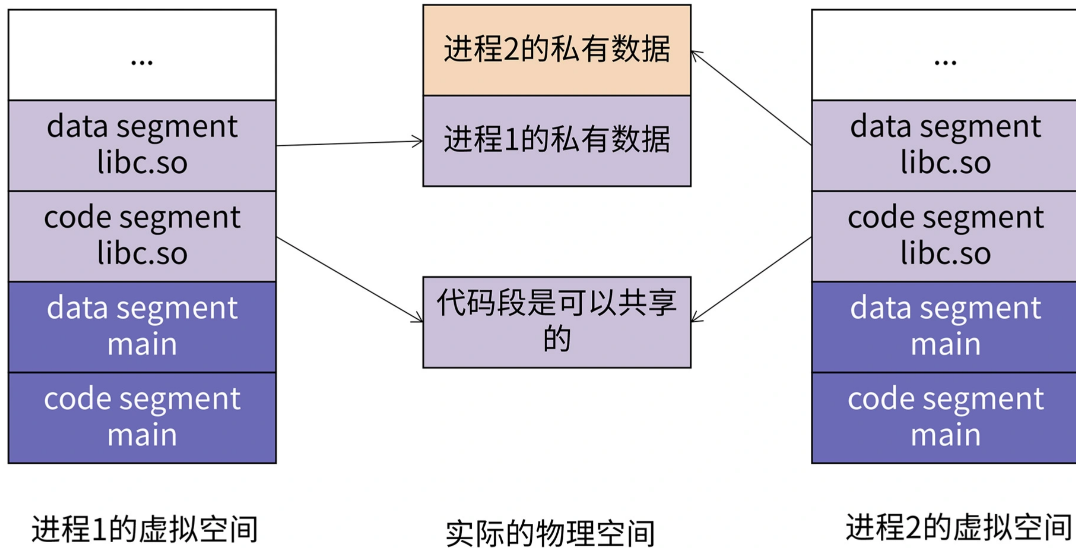
通过这个小例子，我们看到了动态链接技术确实是将共享部分的内存省了下来，但是你也会发现，库文件在不同进程的映射中，虚拟内存地址可以不同。这就要求编译器在生成代码时能适应这个需求，那么地址无关代码技术就诞生了。

为什么会有地址无关代码？

首先，我们思考一下，动态库文件被加载到内存中并且被多个进程共享时，它的内存是什么样子的。

在第 3 节课我们已经看到了，可执行文件或者动态库文件被加载进内存的时候，文件中不同的 section 会被加载进内存中不同 segment，比如.data 和.bss 段被加载进数据段 (data segment)，而.code, .rodata 被加载进代码段 (code segment)。

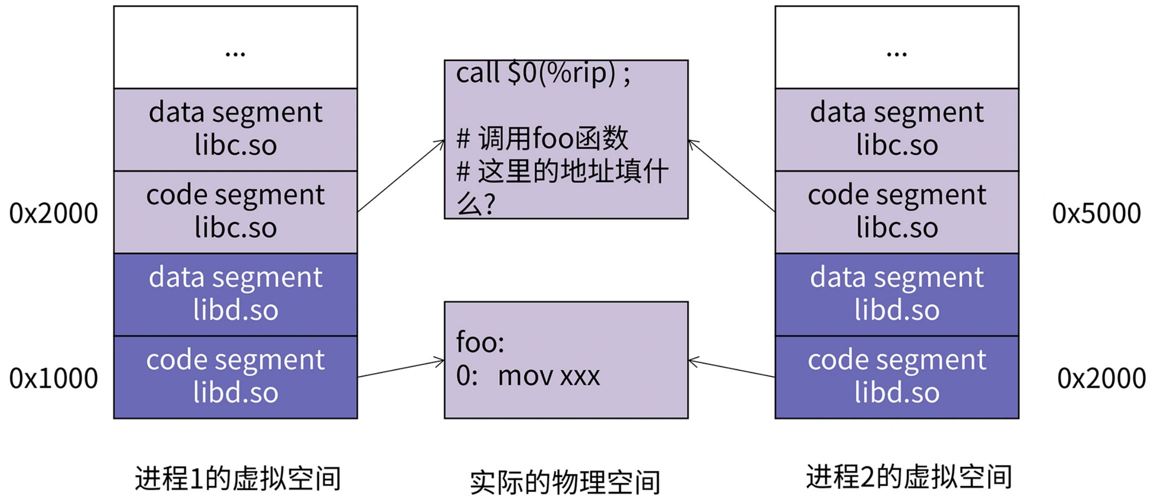
在多进程共享动态库的时候，因为代码段是不可写的，所以进程间共享不存在问题，而数据段可写，系统必须保证一个进程写了共享库的数据段，另外一个进程看不到。这时的内存映射情况如下图所示：



上面这幅图与第 1 节课中的页面映射的图几乎如出一辙。正是虚拟地址技术让我们在进程间共享动态库变得容易，我们只需要在虚拟空间里设置一下到物理地址的映射即可完成共享。

虽然 libc.so 在物理内存中只有一份，但它可以被多个进程进行映射。**而且进程 1 映射 libc.so 代码段的虚拟地址与进程 2 映射 libc.so 代码段的虚拟地址可以不相等**。正如本节课开头所分析的，这样做可以使得多个进程共享一份代码，大大节约了内存。

到目前为止，动态库技术看上去都非常好。但不知道你有没有发现一个问题？如果共享的动态库超过了两个，并且这些动态库之间还有相互引用的时候，情况就变得复杂了。我们还是用图来说明：



如上图所示，如果两个进程共享了 libc.so 和 libd.so 两个动态库，而且 libc 中会调用 libd 中定义的 foo 方法。

进程 1 将 foo 方法映射到自己的虚拟地址 0x1000 处，而调用 foo 方法的指令被映射到 0x2000 处，那么 call 指令如果采用依赖 rip 寄存器的相对寻址的办法，这个偏移量应该填 -0x1000。进程 2 将 foo 方法映射到自己虚拟地址 0x2000 处，调用 foo 方法的指令被映射到 0x5000 处，那么 call 指令的参数就应该填 -0x3000。这就产生了冲突。

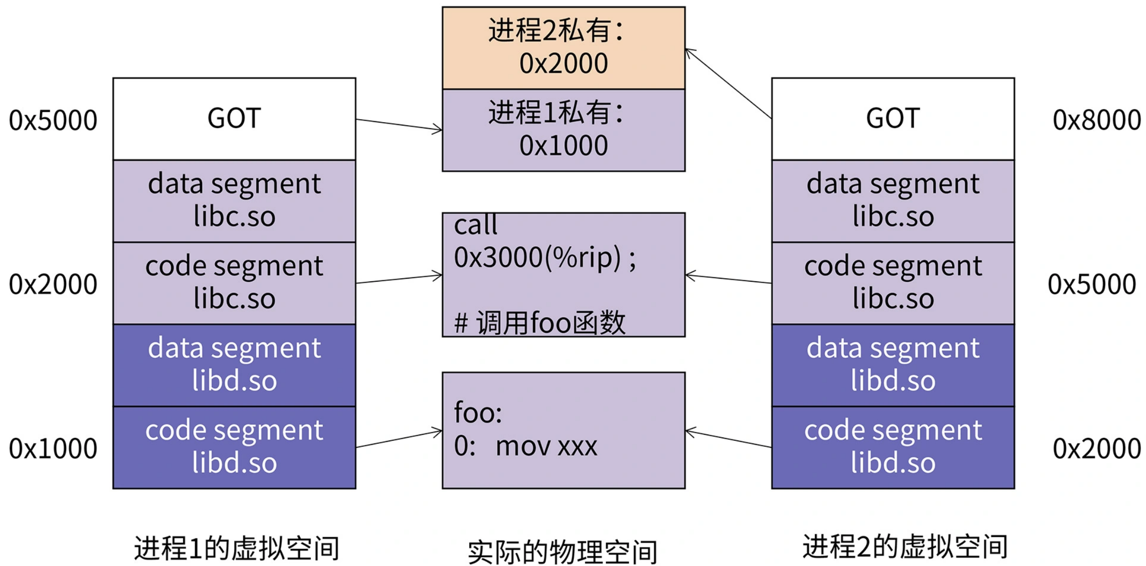
显然，我们第 6 节课所讲的通过 rip 寄存器进行相对寻址的办法在这里行不通了，相对寻址要求目标地址和本条指令的地址之间的相对值是固定的，这种代码就是地址有关的代码。当目标地址和调用者的地址之间的相对值不固定时，就需要地址无关代码技术了。

地址无关代码的核心结构

在计算机科学领域，有一句名言：“计算机领域的所有问题都可以使用新加一层抽象来解决”。这句话的应用在计算机领域随处可见。同样地，要实现代码段的地址无关代码，思路也是通过添加一个中间层，使得对全局符号的访问由直接访问变成间接访问。

我们可以引入一个固定地址，让引用者与这个固定地址之间的相对偏移是固定的，然后这个地址处再填入 foo 函数真正的地址。当然，这个地方必然位于数据段中，是每个进程私

有的，这样才能做到在不同的进程里，可以访问不同的虚拟地址。这个新引入的固定地址就是**全局偏移表 (Global Offset Table, GOT)**。GOT 的工作原理如下图所示：



在上图中，call 指令处被填入了 0x3000，这是因为进程 1 的 GOT 与 call 指令之间的偏移是 0x5000-0x2000=0x3000，同时进程 2 的 GOT 与 call 指令之间的偏移是 0x8000-0x5000=0x3000。所以对于这一段共享代码，不管是进程 1 执行还是进程 2 执行，它们都能跳到自己的 GOT 表里。

然后，进程 1 通过访问自己的 GOT 表，查到 foo 函数的地址是 0x1000，它就能真正地调用到 foo 函数了。进程 2 访问自己的 GOT 表，查到 foo 函数的地址是 0x2000，它也能顺利地调用 foo 函数。这样我们就通过引入了 GOT 这个间接层，解决了 call 指令和 foo 函数定义之间的偏移不固定的问题。

这种技术就是地址无关代码 (Position Independent Code, PIC)。接下来我们用一个实际例子让你加深对 PIC 技术的理解。

与第 6 节课讲解 linker 相似，我们继续用具体的例子来看一下 PIC 技术中对几种常见类型的地址访问是如何处理的。例子如下：

复制代码

```
1 // foo.c
```

```
2
3  static int static_var;
4  int global_var;
5  extern int extern_var;
6  extern int extern_func();
7
8  static int static_func() {
9      return 10;
10 }
11
12 int global_func() {
13     return 20;
14 }
15
16 int demo() {
17     static_var = 1;
18     global_var = 2;
19     extern_var = 3;
20     int ret_var = static_var + global_var + extern_var;
21     ret_var += static_func();
22     ret_var += global_func();
23     ret_var += extern_func();
24     return ret_var;
25 }
```

例子中分别从指令、数据和它的作用域的角度区分了如下几种类型：

1. static_var，表示静态变量的访问；
2. static_func，表示静态函数的访问；
3. extern_var，表示外部变量的访问；
4. extern_func，表示外部函数的访问；
5. global_var，表示全局变量的访问；
6. global_func，表示全局函数的访问；

demo() 函数是用来对以上几种类型进行访问来查看代码的生成。

我们把上边的例子编译成 so 文件，然后反汇编看一下 demo 函数的汇编是怎样的。

```
1  $ gcc foo.c -fPIC -shared -fno-plt -o libfoo.so
2  $ objdump -S libfoo.so
```

```
3 0000000000000680 <demo>:
4  680:   55                      push   %rbp
5  681:   48 89 e5                mov    %rsp,%rbp
6  684:   48 83 ec 10             sub    $0x10,%rsp
7  688:   c7 05 92 09 20 00 01    movl   $0x1,0x200992(%rip)        # 201024 <st
8  68f:   00 00 00
9  692:   48 8b 05 27 09 20 00    mov    0x200927(%rip),%rax        # 200fc0 <gl
10 699:   c7 00 02 00 00 00       movl   $0x2,(%rax)
11 69f:   48 8b 05 4a 09 20 00    mov    0x20094a(%rip),%rax        # 200ff0 <ex
12 6a6:   c7 00 03 00 00 00       movl   $0x3,(%rax)
13 6ac:   8b 15 72 09 20 00       mov    0x200972(%rip),%edx        # 201024 <st
14 6b2:   48 8b 05 07 09 20 00    mov    0x200907(%rip),%rax        # 200fc0 <gl
15 6b9:   8b 00                   mov    (%rax),%eax
16 6bb:   01 c2                   add    %eax,%edx
17 6bd:   48 8b 05 2c 09 20 00    mov    0x20092c(%rip),%rax        # 200ff0 <ex
18 6c4:   8b 00                   mov    (%rax),%eax
19 6c6:   01 d0                   add    %edx,%eax
20 6c8:   89 45 fc                mov    %eax,-0x4(%rbp)
21 6cb:   b8 00 00 00 00          mov    $0x0,%eax
22 6d0:   e8 95 ff ff ff          callq  66a <static_func>
23 6d5:   01 45 fc                add    %eax,-0x4(%rbp)
24 6d8:   b8 00 00 00 00          mov    $0x0,%eax
25 6dd:   ff 15 ed 08 20 00       callq  *0x2008ed(%rip)        # 200fd0 <global
26 6e3:   01 45 fc                add    %eax,-0x4(%rbp)
27 6e6:   b8 00 00 00 00          mov    $0x0,%eax
28 6eb:   ff 15 ef 08 20 00       callq  *0x2008ef(%rip)        # 200fe0 <extern
29 6f1:   01 45 fc                add    %eax,-0x4(%rbp)
30 6f4:   8b 45 fc                mov    -0x4(%rbp),%eax
31 6f7:   c9                      leaveq
32 6f8:   c3                      retq
```

1. 静态变量访问方式

这里先来看一下 static_var。从 demo 的汇编里来看，在 0x688 的位置（第 7 行），我们可以看到这里对 static_var 变量的访问采用的是基于 %rip 的偏移。其中指令后边的注释标明了当前指令访问的虚拟地址 0x201024，通过`objdump -d libfoo.so`查看 0x201024 位置存放的符号是 static_var，在.bss 段中。因此可以看出，在同一个共享文件里边，对 static 变量的访问可以通过 %rip 偏移的方式来确定数据的位置。

目前我们的讲解都是基于 64 位的系统，但这里值得一提的是，32 位系统下由于没有相对 PC 偏移的寻址方式，编译器在生成 32 位 PC 偏移寻址时，是如下的一段汇编：

```
1 000003c0 <__x86.get_pc_thunk.bx>:
```

```
2   3c0:   8b 1c 24               mov    (%esp),%ebx
3   3c3:   c3                     ret
4   ...
5   000004e5 <demo>:
6   ...
7   4ec:   e8 cf fe ff ff         call   3c0 <__x86.get_pc_thunk.bx>
8   4f1:   81 c3 0f 1b 00 00      add    $0x1b0f,%ebx
9   4f7:   c7 83 14 00 00 00 01   movl   $0x1,0x14(%ebx)
10  ...
```

这里可以看到，32 位系统是通过一个 call stub 来获取的 pc 的值。因为 call 指令本身会做的一个操作是将 return address 压栈，而在 __x86.get_pc_thunk.bx 这个 stub 里边，则将当前栈顶的值 (%esp) 取出来放到 %ebx 寄存器中，那么此时 %ebx 里存放的就是 ret 之后的 pc 的值了。这个设计利用了 call 指令的会将下一条指令地址压栈的思路，非常巧妙的获取了 pc 的值，还是很有意思的。

2. **静态函数的访问方式**

静态函数和静态变量一样，都是不能被外部访问的，所以我们也可以推测它的寻址方式和静态变量一样，那么这里我就不再详细讲解验证过程了，请你自己动手验证。

3. **外部变量的访问**

接着来看对 extern_var 的访问。demo 中对 extern_var 的访问是 0x69f 和 0x6a6 两条指令。0x69f 先将 extern_var 的地址 mov 到 rax 寄存器中，然后 0x6a6 则将具体的数据 0x3 写到 extern_var 表示的内存地址中。

可以得到这条指令中使用的实际地址地址是 0x6a6 + 0x20094a = 0x200ff0，继续通过 objdump 来查看对应位置的内容。

```
1 $objdump -D libfoo.so
2  Disassembly of section .got:
3
4 0000000000200fc0 <.got>:
5         ...
```

这里就是我们刚才讲过的 GOT 了。其中存放的是该模块需要访问的所有外部符号的地址。这样可以使得对外部符号的访问转换为对 GOT 表的访问。由于 GOT 表的相对偏移在同一个 so 中肯定是不变的，所以对 GOT 的访问可以使用相对寻址完成。

GOT 中指向的是调用目标的在各自进程中的虚拟地址，我们是通过 GOT 表间接访问的方式，将对外部符号地址的直接依赖消除了。

每个进程都有自己的私有 GOT 段，GOT 中记录了当前的 so 文件所引用的所有外部符号。这些外部符号都需要进行解析和重定位。这个工作由 loader 负责，其为符号分配并记录地址，然后将这些地址回写进 GOT 表。这个过程的原理和上节课所讲的两阶段重定位过程几乎一致，区别仅仅是 linke 操作的是文件中的地址，而 loader 操作的是内存地址。

4. **外部函数访问**

例子中 0x6eb 位置是对 extern_func 的调用处，同外部数据访问类似，这里也是采用了 GOT 表的间接访问的方式，GOT 表 0x200fe0 的位置存放的是 extern_func 的运行时地址，也需要在启动时进行重定位。

5. **全局变量和全局函数的访问**

从例子中可以看到，对于全局变量和全局函数的访问的处理方式，与外部变量和外部函数的访问方式是保持一致的，都是采用 GOT 的方式，因此在这里我就不再详细解释了，你可以去上面的例子中看一下。

总结

为了节约内存，让进程间可以共享代码，人们把可以被共享的代码都抽出来，放到一个文件中，多个进程共享这个文件就可以了。这个可共享的文件就是动态库文件。动态库文件中的符号要在加载时才被解析，所以这种技术就叫动态链接技术。

动态库文件被加载进内存以后，在物理内存只有一份，多个进程都可以将它映射进自己的虚拟地址空间。各个进程在映射时可以将动态库的代码段映射到任意的位置。

如果两个共享库之间有引用关系的话，引用者和被引用者之间的相对位置就不能确定了，这时就需要引入地址无关代码技术。对于内部函数或数据访问，因为其相对偏移是固定

的，所以可以通过相对偏移寻址的方式来生成代码；对于外部和全局函数或数据访问，则通过 GOT 表的方式，利用间接跳转将对绝对地址的访问转换为对 GOT 表的相对偏移寻址，由此得到了地址无关的代码。

地址无关的代码除了可以在 so 中使用，同样可以在可执行文件中使用，可以通过 -pie 选项使得 gcc 编译地址无关的可执行文件。地址文件的可执行文件可以被加载到内存的任意位置执行，这会使得缓冲区溢出的难度增加（你可以结合第 4 节课思考一下原因），但代价是通过 GOT 访问地址会多一次访存，性能会下降。

通过今天这节课，我们对动态链接和其中地址无关代码技术有了整体的认知，但在这里面仍然可以看到引入动态链接带来的一些问题。下节课，我们会进一步探讨动态链接的优化，以及动态链接器与 loader 的实现。

思考题

我们在第 3 节课讲过二进制文件的.text 段会被加载进内存的代码段 (code segment)，请你想一想，.got 段加载进内存的什么位置是比较合理的？

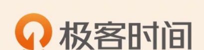
吊打面试官

- 谈谈静态库和动态库的区别?

静态库是在编译阶段就和可执行文件打包链接在一起的，它可以看成是中间文件的简单集合，保留了符号，只有在静态链接的过程，才会真正地做地址分配和重定位。

而动态库在编译阶段，它的代码并不会被合并进可执行文件中，在运行时才会被加载进内存，它被加载进内存的地址是不固定的，所以每次加载完成以后，才能为它的符号分配真实的内存地址，然后再把地址回填到引用它的GOT中。动态库的一个优点是可以在多个进程间共享，从而可以减少内存的重复。

高频面试真题



好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 静态链接：变量与内存地址是如何映射的？

下一篇 08 | 动态链接（下）：延迟绑定与动态链接器是什么？

训练营推荐

Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等大厂真题

18 大知识点专项练习

大厂面试流程解析

可复用的面试方法

面试前要做的准备

精选留言 (8)

写留言

kylin

2021-11-09

老师，请问当两个进程仅仅共享同一个动态库的话，您文中说过：

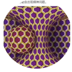
“正是虚拟地址技术让我们在进程间共享动态库变得容易，我们只需要在虚拟空间里设置一下到物理地址的映射即可完成共享。”

因为动态库在不同进程的虚拟内存不一样，所以每个进程只要知道动态库的起始地址就...

展开

作者回复: 注意看文章，我们讲了A.exe和B.exe的smap的结果，那里能看到动态库是独立的，它不会再和其他动态库合并了。这就意味着一个库的数据段和代码段在内存里也是靠在一起的。相对偏移和文件中的相对偏移是一样的。所以它就可以提前决定这个偏移是多少。

1



keepgoing

2021-11-08

老师我在尝试把这节课的知识和上节课联系起来，梳理出几个小问题想请教一下：

1. 一个程序编译不管存不存在动态库一起编译，都会进行一遍编译->静态链接，我理解前两个阶段是运行时无关的，但如果存在动态库，静态编译时的动态库符号是怎么进行区分的呢，区分之后也是跟链接之前的符号一样先用0标记存在重定位表里吗？...

展开

作者回复: 很好。看不懂的就大胆提问，不用不好意思，不存在愚蠢的问题。第一个大问题，在链接时，可执行程序的每一个符号都要被解决，有些是在其它中间文件中，这种就是静态链接；有些是在动态库里，这种就不需要解决了，只要在GOT表里留好位置就行了。所以编译一个依赖动态库的应用程序时，既有静态链接，也有动态链接。这不矛盾。第二个大问题的第1小问，两个因素都起作用的。首先，动态库之间的相对位移是在运行时才能知道的，如果整个程序中只有一个动态库，且这个动态库不再依赖其他库了，那显然不需要PIC。其次，如果不是多个进程需要共享代码的话，那我们把偏移放到代码里也没问题。这两个条件都要满足，这才是要使用PIC的原因。第2个问题，进程没有GOT表，GOT表都是跟着动态库的，你可以再读一个03，然后自己动手看看readelf -l的结果和运行起来以后smap的结果。所以第3个问题也就回答了。

共 3 条评论 >　　1

我是内存

2021-11-18

688: c7 05 92 09 20 00 01 movl $0x1,0x200992(%rip) # 201024 <static_var>
68f: 00 00 00

文中这里rip=0x68f，加上0x200992=201021，不是注释中的0x201024那个地址呀。
而且用objdump -d libfoo.so也看不到0x201024，我在自己的机器上也看不到本机反汇编...

展开

共 1 条评论 >

我是内存

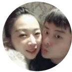
2021-11-18

你好，我对文中说明使用GOT时候的例子有一个疑问。

文中的背景是：

1）进程1映射foo在0x1000处，调用foo的指令在0x2000，但是填入的是call 0x3000(%ri...

展开


鸮
2021-11-15

老师，有个问题请教下，动态链接技术减少了内存损耗，so文件只在内存中存在一份。比如a先启动，加载了lib.c，然后b又启动，也来加载lib.c，那b是如何知道 lib.c的是否在内存中，并且知道llib.c的真实物理地址的。思考题 ：got表每个进程独一份，记录了需要被重定向的数据，就是把不确定的数据全放在got表，加载进内存后，进行重定位，此时要回写got，将真实的地址写回，因此是 可读可写的，应该在 数据段。

展开

作者回复: 看第十课，mmap的私有文件映射能找到这个问题的答案。

kylin
2021-11-09

GOT是不同进程私有的，并且可以读写，应该在数据段

作者回复: Good

好吃不贵
2021-11-09

GOT表是存在数据段中的。
通过readelf的Segment sections的mapping可以看到。
03 .tdata .init_array .ctors .dtors .data.rel.ro .dynamic .got .got.plt .data .bss

展开

作者回复: Good，动手看一下最直接。不过，还是要稍想一下，为什么要这么做？

2021-11-08

老师，请教个问题啊，GOT这块之前也去学习搞过，但这个东西不用就忘记，原理类的有必要去了解吗，学习到那种程度，还是说知道有这个东西就可以了

展开

作者回复: 这节课的核心就是地址无关代码产生的动机是什么。我们学习计算机知识，一定不能死记硬背。你不必记忆got是什么，只要理解了pic产生的原因，got这种东西就是自然而然的。掌握了这个方案，你下次在工作中遇到类似的问题就知道往哪个方向去研究了。死记硬背的东西很快就忘了，那种学习方法我是非常反对的。