下载APP

08 | 动态链接（下）：延迟绑定与动态链接器是什么？

2021-11-10 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 22:56 大小 21.02M

你好，我是海纳。

在上节课里，我们学习了动态链接过程的基本原理。动态链接通过 GOT 表加一层间接的方式，解决了代码中 call 指令对绝对地址的依赖，从而实现了 PIC 的能力。我们同时也讲到了 GOT 表中的地址是由加载器在加载时填充的。

领资料

不过，细心的你也发现了，动态链接带来的代价是性能的牺牲。这里性能的牺牲主要来自于两个方面：

1. 每次对全局符号的访问都要转换为对 GOT 表的访问，然后进行间接寻址，这必然要比直接的地址访问速度慢很多；
2. 动态链接和静态链接的区别是将链接中重定位的过程推迟到程序加载时进行。因此在程序启动的时候，动态链接器需要对整个进程中依赖的 so 进行加载和链接，也就是对进程中所有 GOT 表中的符号进行解析重定位。这样就导致了程序在启动过程中速度的减慢。

我们这节课来看看，如何通过延迟绑定技术，来解决性能下降的问题。延迟绑定不仅仅是用在动态链接中，还被广泛地应用在 Hotspot，V8 等带有即时编译功能的虚拟机中。另外，在游戏行业，修复服务器的错误的同时保证用户不掉线是硬需求，这种不停机进行代码修复的技术被称为热更新技术。学习完这节课后，你不仅能理解动态链接的基本原理，而且也能对热更新的基本原理有所感悟。

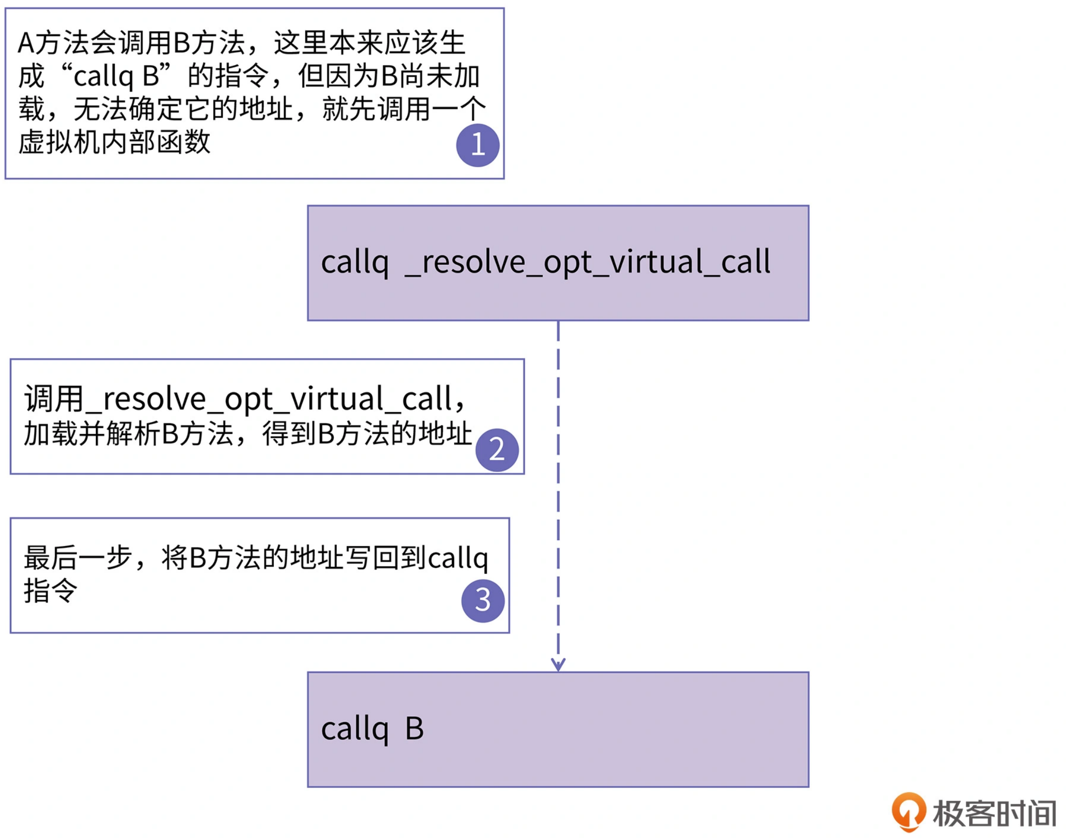
其实，不管是加载时重定位，还是延迟绑定技术，真正发挥作用的是动态链接器。所以这节课我也会给你简单介绍一下动态链接器的基本原理。

首先，我们从延迟绑定的最简单的形式，也就是 Hotspot 虚拟机中的运行时重定位技术 patch code 讲起。

patch code 技术

我们知道，在 Java 语言中，类是按需加载的。也就是对于一个 class 文件，只有当 hotspot 第一次使用它的时候，它才会被加载进来。假如我们在即时编译 A 方法的时候要调用 B 方法，但这时 B 方法还没有被加载进来，该怎么办呢？

虚拟机会采用一种叫做 patch code 的技术，在运行时再进行加载。简单地说，就是在生成 call 指令时候，它的目标地址填成一个虚拟机内部的用于解析符号的方法。在 CPU 执行这条 call 语句的时候，就会调用符号解析函数。此时虚拟机就会加载 B 方法所在的类，然后就能确定 B 方法的地址了，这时再把 B 方法的地址写回到 call 指令里。这个过程如下图所示：



这个过程很像是在给原始的代码打补丁，所以人们就把这种方式称为 **patch code 技术**。这就像是在原来的代码安装了一个机关，当 CPU 执行到这个机关时，就会触发一次符号的重定位，然后这个机关就被替换掉了。下一次 CPU 再执行到这个 call 指令的时候，就可以正常地调用到 B 方法了。

上节课，加载器在加载动态库时就把它的 GOT 中的所有符号都解析了，这种方法却把解析符号的过程又往后推到了执行代码时解析。

在 Hotspot 里的 patch code 技术，会直接修改指令参数。不过，运行时修改指令总是一件很危险的事情。所以，动态库真正使用的运行时解析符号技术是延迟绑定技术，它的关键步骤和 patch code 很相似，但却比 patch code 的安全性更好一些，我们一起来看一下。

延迟绑定技术

为了避免在加载时就把 GOT 表中的符号全部解析并重定位，就需要采用计算机领域非常重要的一个思想：Lazy。也就是说，把要做的事情推迟到必须做的时刻。

对于我们当前的问题来说，**将函数地址的重定位工作一直推迟到第一次访问的时候再进行，这就是延迟绑定 (Lazy binding) 的技术**。这样的话，对于整个程序运行过程中没有访问到的全局函数，可以完全避免对这类符号的重定位工作，也就提高了程序的性能。

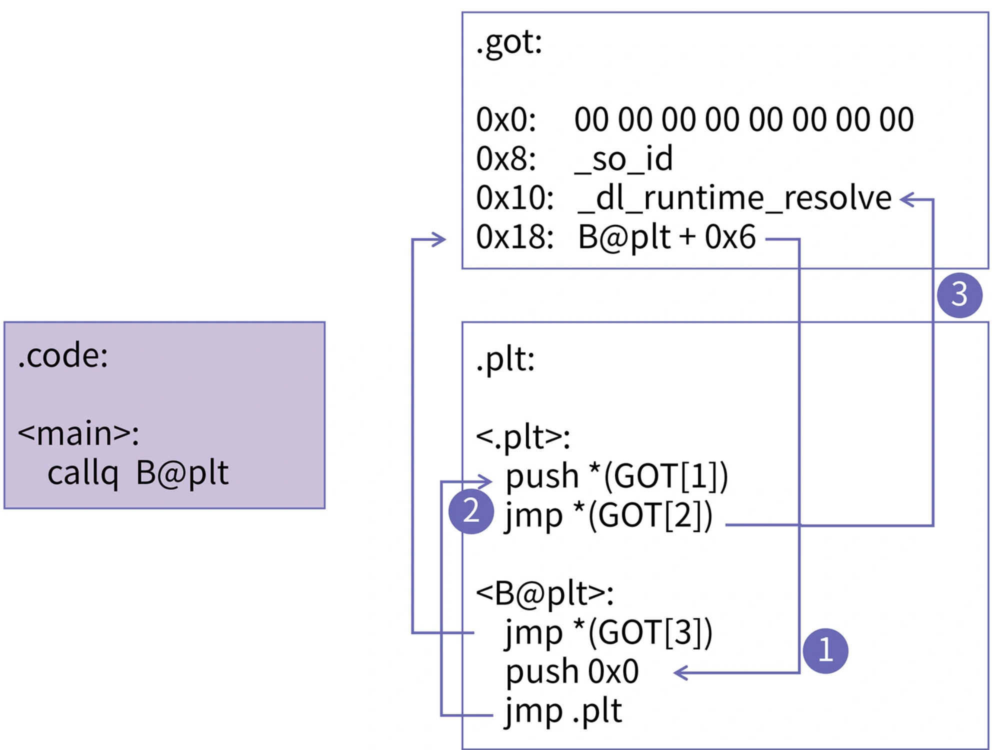
patch code 显然也是一种延迟绑定的技术，但是它要在运行时修改指令参数，这会带来风险。所以动态库的延迟绑定选择了继续使用 GOT 表来进行间接调用，然后 patch 的对象就不再是指令了，而是 GOT 中的一项。

理想情况下，我们把 GOT 中的待解析符号的地方都填成动态符号解析的函数就可以了，当 CPU 执行到这个函数的时候，就会跳转进去解析符号，然后把 GOT 表的这一项填成符号的真正的地址。如下图所示：

```
.got:

0x0:    00 00 00 00 00 00 00 00
0x8:    _so_id
0x10:   _dl_runtime_resolve
0x18:   _dl_runtime_resolve

.code:

<main>:
  callq  *(GOT[2])
```

但是动态解析符号的函数 _dl_runtime_resolve 依赖两个参数，一个是当前动态库的 ID，另一个是要解析的符号在 GOT 表中的序号。动态库的 ID 存储在 GOT 的 0x8 偏移的位置，而要解析的符号序号却不容易得到。

为了解决传递参数的问题，动态链接又引入了过程链接表（Procedure Linkage Table，PLT），将动态解析符号的过程做成了三级跳。如下图所示：



在图中，我用序号①、②、③和它们旁边的箭头分别给你标注出了三级跳的路径。如果你仔细观察的话，你还会发现这张图与上一张图的主要变化就是引入了.plt 段，在代码段里，main 函数对 B 函数的调用转成了对"B@plt"的调用，"B@plt"函数只有三条指令。

它的第一条指令 jmp *(GOT[3]) 是一个间接跳转，跳转的目标是 GOT 表偏移为 0x18 的位置，正常情况下，这个位置应该放的是 B 函数的真实地址。但现在填入的是指向了 B@plt + 0x6 的位置，这是为了传递参数给 _dl_runtime_resolve 函数。B@plt+0x6 的位置其实就是 B@plt 函数的第二条指令，它的作用是将函数参数入栈，然后执行第三条指令 jmp .plt 再准备第二个参数。

我们再回到图中看看，在序号①箭头的位置，也就是第一级跳转，它的目的是把参数 0 入栈。由于 GOT 表的 0x0，0x8，0x10 的位置都被占用了，所以参数 0 代表的就是 0x18 位置，这就是 B 函数的真实地址应该存放的地方。

然后在序号②箭头的位置，发生了第二级跳转，这一次是为了把动态库的 ID 号压栈传参。

最后在序号③箭头的位置，继续进行第三级跳转，这一次跳转才真正地调用到了_dl_runtime_resolve。调用完这个方法以后，B 函数的真实地址就会被填入 GOT 表中了。

上述过程由于传参的需要而变成了多级跳转，但如果抛开因为传参而产生的两级跳转，你会发现它的基本结构与 patch code 技术如出一辙。

这样的跳转虽然麻烦，但有一个非常重要的优点，就是运行期间不会修改代码段的指令，所有的修改只涉及了 GOT 这个位于数据段的表里。我们在第 3 节课就已经介绍过，.code 和.plt 会被加载到内存的代码段 (code segment)，它的权限是可读可执行，但不可写；上节课也讲了.got 会被加载进数据段，它的权限是可读可写。我们现在介绍的**多级跳转的延迟绑定技术的整个重定位过程最终只会修改 GOT 的 0x18 这一个位置，其他位置都不必发生变化。**

当执行完了重定位过程以后，CPU 再一次运行到 main 里的 call 指令时，就能通过一次跳转就调用到真正的 B 函数了，这时的 GOT 已经与上节课所讲的加载时重定位后的 GOT 一模一样了。如图所示：

```
.got:

0x0:   00 00 00 00 00 00 00 00
0x8:   _so_id
0x10:  _dl_runtime_resolve
0x18:  B’ s real address
```

```
.code:

<main>:
  callq  B@plt
```

```
.plt:

<.plt>:
  push *(GOT[1])
  jmp *(GOT[2])

<B@plt>:
  jmp *(GOT[3])
  push 0x0
  jmp .plt
```

在这个图里，重定位完以后，只有红色字体的代码和数据是起作用的，.plt 段里的其他代码就被“短路”掉了。这时，GOT 表的结构就与上节课所讲的加载时重定位的情况完全一样了。**只有用到的符号才会被重定位，这就是延迟绑定技术**。未被用到的符号在加载时被重定位，这是一种浪费，而延迟绑定技术避免了这种浪费。为了加深理解，我结合一个具体例子向你展示延迟绑定是怎么实现的。

延迟绑定技术的具体实现

下面我们还是根据上节课的例子来看一下延迟绑定技术的具体实现。

复制代码

```
// foo.c

static int static_var;
int global_var;
extern int extern_var;
extern int extern_func();

static int static_func() {
    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 }
```

我们将这个例子编译成 libfoo.so ，编译命令是：

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

这里跟上节课编译的区别是，去掉了 -fno-plt 的编译选项，这样可以打开 PLT 表的生成。上一节课里，我们只需要关注 PIC 技术的实现，因此需要通过 -fno-plt 的选项来关闭 PLT 表的生成。

我们先通过反汇编先来看一下 demo 函数的汇编指令：

```
1 00000000000006a0 <demo>:
2  ...
3  6fd:    e8 7e fe ff ff          callq  580 <global_func@plt>
4  702:    01 45 fc                add    %eax,-0x4(%rbp)
5  705:    b8 00 00 00 00          mov    $0x0,%eax
6  70a:    e8 81 fe ff ff          callq  590 <extern_func@plt>
7  70f:    01 45 fc                add    %eax,-0x4(%rbp)
8  712:    8b 45 fc                mov    -0x4(%rbp),%eax
9  715:    c9                      leaveq
10 716:    c3                      retq
```

从汇编中你可以看到，对函数 global_func 和 extern_func 的调用都变成了对 global_func@plt 和 extern_func@plt 的调用。继续查看这两个带 @plt 后缀的函数，其对应的 VMA 分别是 0x580 和 0x590，所以接着看这两个位置的汇编代码。

复制代码

```
1 Disassembly of section .plt:
2
3 0000000000000570 <.plt>:
4  570:   ff 35 92 0a 20 00       pushq  0x200a92(%rip)        # 201008 <_GLOBAL
5  576:   ff 25 94 0a 20 00       jmpq   *0x200a94(%rip)        # 201010 <_GLOBA
6  57c:   0f 1f 40 00             nopl   0x0(%rax)
7
8 0000000000000580 <global_func@plt>:
9  580:   ff 25 92 0a 20 00       jmpq   *0x200a92(%rip)        # 201018 <global
10 586:   68 00 00 00 00          pushq  $0x0
11 58b:   e9 e0 ff ff ff          jmpq   570 <.plt>
12
13 0000000000000590 <extern_func@plt>:
14 590:   ff 25 8a 0a 20 00       jmpq   *0x200a8a(%rip)        # 201020 <extern
15 596:   68 01 00 00 00          pushq  $0x1
16 59b:   e9 d0 ff ff ff          jmpq   570 <.plt>
```

这段汇编是对 libfoo.so 中.plt 段的反汇编。从这里我们可以看出来，PLT 表的每一项其实都是一段相似的 stub 代码构成，这个 stub 共三条指令，这三条指令和我们上面的图中所画的是完全一样的。

从反汇编的结果来看，global_func@plt 的第一行是一个间接跳转，跳转的目标地址存储在 0x201018 这个位置，通过 objdump 我们可以找到这个位置位于.got.plt 段里。这个命令我们已经很熟悉了，你可以自己动手试一下。从名字中可以看出，.got.plt 段跟.got 段是一样的，存放的是 GOT 表，只不过.got.plt 里边的 GOT 表是为 PLT 表准备的。

在这里 0x201018 的位置存放的值是 0x586。这就跳回到 global_func@plt 里继续执行了，这是我们上面所分析的一级跳，是为了传递参数给符号解析函数的。最终经过传参，跳转，控制流才终于进入到 dl_runtime_resolve 中解析符号并做重定位。

最后，我们再总结一下 GOT 表中的各个表项的含义。

1. GOT.PLT[0]位置被加载器保留，它里面存放的是.dynamic 段的地址，这里我们不用关心。

2. GOT.PLT[1]位置存放的是当前 so 的 ID，这个 ID 是加载器在加载当前动态库文件的时候分配的。
3. GOT.PLT[2]位置存放的是动态链接函数的入口地址，一般是动态链接器中的 _dl_runtime_resovle 函数。这个函数的作用是找到需要查找的符号地址，并最终回填到 GOT.PLT 表的对应位置。

然后再回顾一下延迟绑定的整个过程。

1. 当 demo 函数想要调用 global_func 的时候，程序调用先进入 global_func@plt 中；
2. 在 global_func@plt 中，会先执行 jmpq *GOT.PLT[3] ，此时 GOT.PLT[3] 里存放的是 global_func@plt 项中的第二条指令，因此控制流继续返回到 global_func@plt 中进行执行；
3. 接下会把数值 0x0 进行压栈，这个数值代表了 global_func 的 ID。然后 jmp 到 PLT[0] 的表项中进行执行；
4. 在 PLT[0] 中，继续将 GOT.PLT[1] 的值也就是库文件的 ID 进行压栈，然后通过 GOT.PLT[2] 跳转到 _dl_runtime_resolve 函数中；
5. _dl_runtime_resolve 则根据存在栈上的函数 ID 和 so 的 ID 进行全局搜索，找到对应的函数地址之后就可以将其重新填充到 GOT.PLT[3] 中，这个时候延迟加载的整个过程就完成了；
6. 当下一次调用 global_func 的时候，CPU 就可以通过 global_func@plt 中第一条指令 jmpq *GOT.PLT[3] 直接跳转到 global_func 的真实地址中。

到这里，我们对动态链接中 PIC 技术和延迟加载技术进行了深入的分析。这个过程中我们几次提到动态链接器，但一直没有展开说，接下来我们就来揭开动态链接器的神秘面纱。

Loader 的加载机制

虽然我们已经搞清楚了链接的全部流程。不过还缺了最后一环，就是可执行文件和共享库文件是如何被加载的？

在 Linux 下，编译一个最简单的可执行程序，通过 ldd a.out 命令你会发现有一个特殊的共享库文件：ld-linux-x86-64.so。从名字上可以看出，这个 ld-linux.so 跟链接器 ld 应该是存在某种联系的。

动态链接会把不同模块之间，符号重定位的操作，推迟到程序运行的时候，而 ld-linux.so 就负责这个工作。所以**我们经常称 ld.so 为动态链接器，又因为它还负责加载动态库文件，所以我们有时也叫它 loader，或者加载器。**

我们知道，一个完全静态链接的可执行文件则不需要动态链接器的辅助，所以内核加载完之后可以直接跳转到用户代码的入口中进行执行。内核加载的过程主要是打开文件，初始化进程空间，读磁盘加载文件数据等等，这部分工作不是我们关心的重点，所以就不再分析了。

而对于一个需要动态链接的可执行文件 a.out，当我们在 Linux 的 shell 终端里边敲了./a.out 的命令后，内核会先准备好可执行文件需要的环境，然后依次把 a.out 和 ld-linux.so 加载到内存中，下一步就是跳转到 ld-linux.so 的入口函数中。

进入 ld-linux.so 以后，与上文所讲的内核的文件加载过程就有区别了。它已经不是内核态执行，而是用户态执行了。ld-linux.so 的源码实际上是在 glibc 里边，主要实现都是在 glibc 的 elf 文件夹下。

ld-linux.so 做的事情主要有这么几件：第一是启动动态链接器；第二是根据可执行文件的动态链接信息，寻找并加载可执行文件依赖的.so 文件；第三步是跟静态链接器一样，对所有的符号进行解析和重定位；最后会根据 so 的情况来依次执行各个 so 的 init 函数。

启动动态链接器

在第一点中，你可能会问，加载跟启动动态链接器的事情不是已经在内核里边做过了么？这里启动动态链接器是在做什么呢？

我们知道，动态链接器的作用是用来对可执行文件中需要动态链接的这些全局符号进行重定位解析，填写 GOT 表等，这时候你会发现，ld-linux.so 本身也是一个共享文件，那它自己的动态链接的过程是谁来进行呢？

答案就是自己。**ld-linux.so 在启动之后，首先需要完成自己的符号解析和重定位的过程，这个过程叫做动态链接器的自举 (Bootstrap)**。ld-linux.so 中的整个自举过程的代码是需要非常小心翼翼的，因为此时 ld-linux.so 本身的 GOT/PLT 信息都未完成，所以在自举过

程中的代码不能使用全局符号和外部符号，稍有不慎就会导致整个程序崩溃。你可以到 elf/rtld.c 中看一下这块代码，主要逻辑在 _dl_start 函数里。

加载依赖共享文件

完成自举后，ld-linux.so 就可以放心的使用各种全局符号和外部符号了。接下来第二步是根据可执行文件的.dynamic 段信息依次加载程序依赖的共享库文件。**程序的共享库依赖关系往往是一个图的关系，所以这里在加载共享库的过程也相当于是图遍历的过程，这里往往采用的是广度优先搜索的算法来遍历。**

在第 6 节课，我们讲过静态链接，在链接的过程中需要维护一个全局的符号表，遍历.o 文件的时候不断收集文件中的符号并且合并到全局符号表中。

同样的，ld-linux.so 在加载共享文件的过程中也会维护一个全局符号表，每次加载新的共享文件后，将共享文件中的符号信息合并到全局符号表中。这个时候，问题来了：如果两个不同的 so，如 libfoo1.so 与 libfoo2.so 都定义了一个 foo 函数，那 ld-linux.so 加载这两个 so 的时候会发生什么？

在静态链接的过程中，如果不同的.o 里边定义了相同的符号，这时链接器会报出 redefine 的错误。而 ld-linux.so 的执行策略则是不同的，ld-linux.so 在碰到相同的符号时，只会将第一次碰到的符号添加到全局符号表中，而后续碰到重名的符号就被自动忽略。

这样导致的结果是，不同 so 的同名函数，在运行时能看到的只有加载顺序在前的函数定义。所以对于上面的问题而言，如果 libfoo1.so 依赖在前，那么最终运行时只能看到 libfoo1.so 的 foo 函数，即使是 libfoo2.so 里的函数调用 foo，调用的也是 libfoo1.so 里的 foo，而不是自己 so 的 foo。由此我们在开发过程中一定需要注意不同 so 中符号重名的问题，否则就会碰到意想不到的问题。

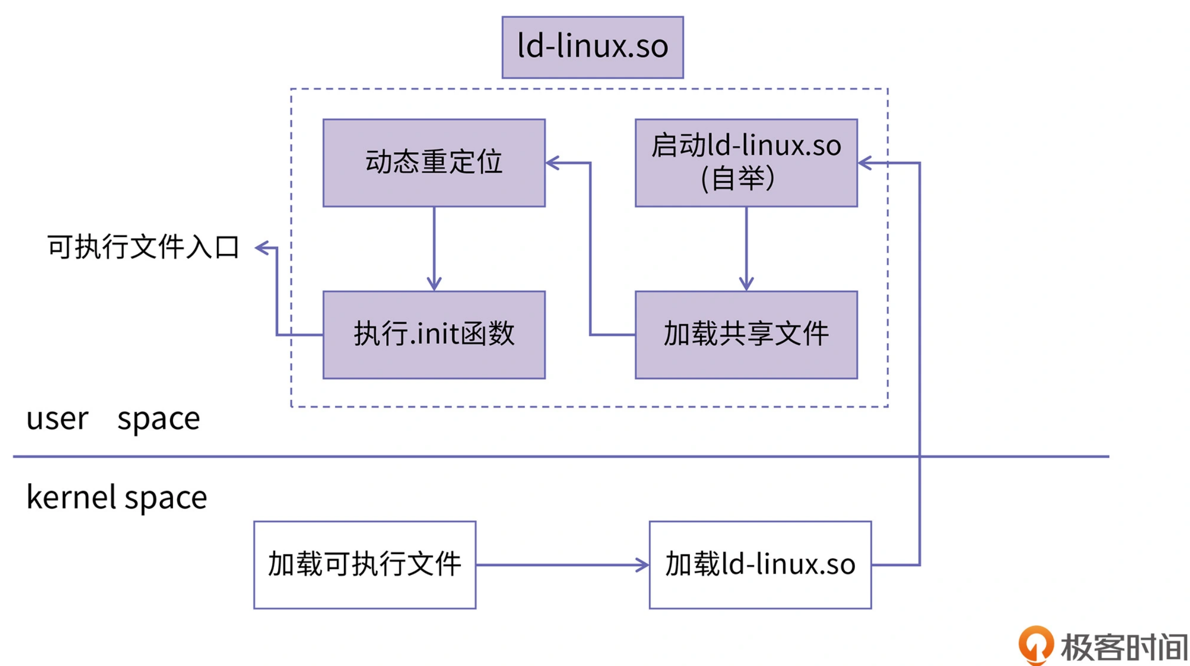
符号重定位与解析

在完成了共享文件的加载之后，全局符号表的信息就收集完成了，这时 ld-linux.so 就可以根据全局符号表和重定位表的信息依次对各个 so 和可执行文件进行重定位修正了。**这个过程跟静态链接中重定位的过程类似**，你可以自己去分析一下。

init 函数调用

最后，有的 so 文件还会有.init 段，进行一些初始化函数的调用，例如 so 中全局变量的对象构造函数，或者用户自己生成在.init 段的初始化函数等。这些都会由 ld-linux.so 在最后的阶段进行一次调用。当这些完成之后，ld-linux.so 就会结束自己的使命，最终将程序的控制流转到可执行文件的入口函数中进行。

整个 Loader 加载动态链接的可执行文件流程如下图所示：



总结

我们通过三节课的学习，弄明白了“将符号转成地址”这个工作是由谁、在何时、如何完成的。

编译器在把源代码翻译成汇编指令的过程中，由于不知道其他编译单元的符号的真实地址，在引用这些符号的时候只能使用占位符（通常是 0）来代替。这些占位符由链接器填充。当链接器把所有的符号的位置都确定好以后，再把真实地址回填到占位符里，这个过程就是重定位。

重定位的时机有三个，分别是编译期重定位（第 6 节课），加载期（第 7 节课）和这节课介绍的运行时重定位。

这节课我们先介绍了 patch code 技术，它被采用了即时编译的语言虚拟机广泛地使用。它可以做到运行时解析符号。它的主要原理是把 call 指令的目标地址填成用于解析符号的函数地址，当 CPU 执行到这个 call 指令时就会转去解析函数，然后把 call 指令的目标地址替换成符号的真实地址。

patch code 技术有一个缺点，那就是在运行期要修改代码段的数据，这为系统带来了风险。动态链接库则引入了.plt 和.got 段，通过间接调用来解决这个问题。在运行时，符号解析函数只需要修改 GOT 的内容就可以了，代码段是不会发生任何变化的。

当然，因为要向符号解析函数传递参数，所以动态库的.plt 设计成了三级跳转的结构，看上去虽然很复杂，但我们只需要牢牢记住.plt 最终的目标还是调用到符号解析函数，然后重写 GOT 表的内容即可。

我们这两节课的内容都是动态链接，而真正负责动态链接的是 ld-linux.so，它被称为动态链接器，但因为它还负责加载文件工作，所以也被人称为加载器或者 loader。它的工作流程主要有启动，加载，重定位和 init 四个步骤。

链接与加载还有很多细节，但我已经带你建立起了基本的知识框架。如果对链接和加载还有更浓厚的兴趣，你可以参考《程序员的自我修养 》，《链接器和加载器 》等书，以便了解更多的相关结构和算法。

思考题

相信你已经完全理解了动态链接器的时机和原理了，那么请你思考一下：在生成一个动态库文件的时候，我们一定要加 shared 选项，但 -fPIC 选项是必然要加的吗？有没有不需要用这个选项的情况呢？如果没有，为什么？如果有的话，又是什么情况呢？欢迎你在留言区分享你的想法和收获，我在留言区等你。

吊打面试官

- 一个命令执行程序，在它的main函数执行之前，发生了哪些事情呢?

首先，shell在执行一个新的命令时，第一步会使用fork创建一个进程，第二步，调用execve来运行新的程序，execve的作用是处理命令行参数，处理环境变量，设置好进程空间，比如堆空间，主线程栈空间等等;

第三步是动态链接器加载程序所依赖的动态库，如果动态库需要在加载时重定位，那么这一步加载器就会解析动态库的符号并做重定位，如果是运行时重定位，这一步只需要做最基本的动态库文件注册即可。这些工作都完成以后，就可以跳到main函数中去执行了。

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

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。

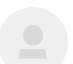
上一篇 07 | 动态链接（上）：地址无关代码是如何生成的？

下一篇 09 | 深入理解堆：malloc和内存池是怎么回事？

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 (4)

写留言



shenglin

2021-11-18

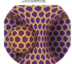
Disassembly of section .got.plt:

0000000000201000 <_GLOBAL_OFFSET_TABLE_>:
  201000: 08 0e or %cl,(%rsi)
  201002: 20 00 and %al,(%rax)...

展开

作者回复: 这里可能是我没讲清楚哈。GOT表里的是一个地址，跳转发生在plt的第一行，你观察一下，plt中的那条jmp指令，它跳转的目标是从GOT里获取的。所以GOT里是数据，不是指令。你再看一下0x201020那个位置是不是0x0656? :)

共 2 条评论 > 1



keepgoing

2021-11-11

三节课的编译链接内容感觉收获不小，尝试把知识串连一下加深印象：
编译器：生成每个编译单元的机器码，生成重定位表，符号地址0占位
链接器：合并目标文件，静态符号分配地址，重定位表寻找静态符号填址
加载器（加载过程）：重定位表找动态符号，根据符号所在动态库，把符号偏移定位在各

个动态库的GOT表中。对于各个动态库中的符号，进行地址分配，填回GOT表中。...

展开

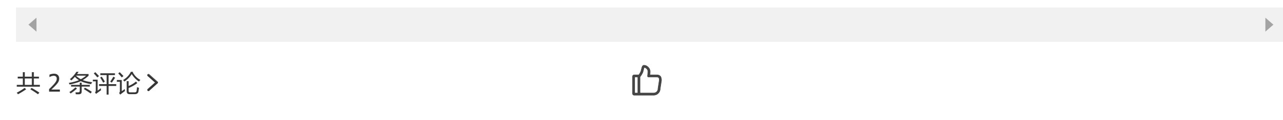
kylin

2021-11-10

海纳老师，您好，我有一个疑问，加载时动态链接的话，每个.so文件（动态链接库）是不是都有一个自己的GOT表？

展开

作者回复: 有不需要的情况，可以想一下got产生的动机是什么？它要解决什么问题？有没有哪种情况不存在这个问题的？



共 2 条评论

kylin

2021-11-10

-fPIC 选项有时候是不需要的。
 在动态链接库不需要依赖其它动态库的时候，就不需要位置无关代码，所以我认为在这种特殊场景下是不需要打开-fPIC的。

作者回复: very good