第32卷 第22期 计 算 机 工 程 2006年11月
Vol.32 No.22 Computer Engineering November 2006
·软件技术与数据库· 文章编号：1000—3428(2006)22—0064—03 文献标识码：A 中图分类号：TP316

Linux 动态链接机制研究及应用

张和君，张 跃
(清华大学深圳研究生院嵌入式系统与技术实验室，深圳 518055)

摘 要：结合 ELF 共享库文件格式，研究了 Linux 系统动态链接的机制和原理，论述了实现的关键技术：PIC 代码，全局偏移表 GOT，过程链接表 PLT；分析了动态库的加载映射过程以及符号解析技术。利用 Linux 的动态链接机制实现了共享库重定位的应用。
关键词：Linux；ELF；动态链接；共享库重定位

Research and Application of Dynamic Link Mechanism in Linux

ZHANG Hejun, ZHANG Yue
(Embedded System Lab, Graduate School at Shenzhen, Tsinghua University, Shenzhen 518055)

【Abstract】 The technological details about dynamic link mechanism are researched, with specific ELF shared object file. Critical questions are illustrated which include: position independent code(PIC), global offset table(GOT), procedure link table(PLT), linking and mapping of dynamic libraries, and symbols resolving. The paper suggests the redirect application of shared library with dynamic link technology.
【Key words】 Linux; Executable linking format(ELF); Dynamic link; Shared library redirection

动态链接库和动态链接是重要的软件运行机制，它既是多个进程共享资源的主要方式，也是操作系统向应用程序提供系统服务的主要手段。动态链接主要有 2 个特点：动态的加载，当运行模块在需要的时候才将动态链接库加载映射入运行模块的虚拟地址空间；动态的解析，只有当要调用的函数被调用的时候，才会把这个函数在虚拟内存空间的起始地址解析出来。

除了资源共享这个最基本的目的之外，利用动态链接机制还可实现共享库重定向等重要应用。本文将研究 Linux 下动态链接的实现机制和原理，以及据此实现的共享库重定向技术。

1 Linux 的动态链接机制

Linux下的动态链接库文件是ELF[1]格式的。Linux的动态链接实现机制引入了全局偏移表(GOT)和过程链接表(PLT)，实现了位置无关代码(PIC)[2]。同时，ELF文件的分层管理模型在Linux动态链接机制中也起着重要的作用。

1.1 位置无关代码和 GOT 表

ELF 格式的共享库使用 PIC 技术使代码和数据的引用与地址无关，程序可以被加载到地址空间的任意位置。PIC 在代码中的跳转和分支指令不使用绝对地址。PIC 在 ELF 可执行映像的数据段中建立一个存放所有全局变量指针的全局偏移量表 GOT。

对于模块外部引用的全局变量和全局函数，用 GOT 表的表项内容作为地址来间接寻址；对于本模块内的静态变量和静态函数，用 GOT 表的首地址作为一个基准，用相对于该基准的偏移量来引用，因为不论程序被加载到何种地址空间，模块内的静态变量和静态函数与 GOT 的距离是固定的，并且在链接阶段就可知晓其距离的大小。这样，PIC 使用 GOT 来引用变量和函数的绝对地址，把位置独立的引用重定向到绝对位置。

对于 PIC 代码，代码段内不存在重定位项，实际的重定位项只是在数据段的 GOT 表内。共享目标文件中的重定位类型有 R_386_RELATIVE、R_386_GLOB_DAT 和 R_386_JMP_SLOT，用于在动态链接器加载映射共享库或者模块运行的时候对指针类型的静态数据、全局变量符号地址和全局函数符号地址进行重定位。

1.2 PLT 表

过程链接表用于把位置独立的函数调用重定向到绝对位置。通过 PLT 动态链接的程序支持惰性绑定模式。每个动态链接的程序和共享库都有一个 PLT，PLT 表的每一项都是一小段代码，对应于本运行模块要引用的一个全局函数。程序对某个函数的访问都被调整为对 PLT 入口的访问。

PLT的结构[2]如下(n=1,2,…)：

```
.PLT0:  pushl    4(%ebx)
        jmp      *8(%ebx)
.PLTn:  jmp      *name@GOT(%ebx)
        push     $offset
        jmp      .PLT0@PC
```

每个 PLT 入口项对应一个 GOT 项，执行函数实际上就是跳转到相应 GOT 项存储的地址，该 GOT 项初始值为 PLTn 项中的 push 指令地址(即 jmp 的下一条指令，所以第 1 次跳转没有任何作用)，待符号解析完成后存放符号的真正地址。动态链接器在装载映射共享库时在 GOT 里设置 2 个特殊值：在 GOT+4(即 GOT[1])设置动态库映射信息数据结构 link_map 地址；在 GOT+8(即 GOT[2])设置动态链接器符号解析函数的地址_dl_runtime_resolve。

PLT 的第 1 个入口 PLT0 是一段访问动态链接器的特殊代码。程序对 PLT 入口的第 1 次访问都转到了 PLT0，最后跳入 GOT[2]存储的地址执行符号解析函数。待完成符号解析

作者简介：张和君(1981－)，男，硕士生，主研方向：嵌入式系统及应用；张 跃，副教授
收稿日期：2005-12-05 **E-mail：**zhang_hj03@mails.tsinghua.edu.cn

后，将符号的实际地址存入相应的 GOT 项，这样以后调用函数时可直接跳到实际的函数地址，不必再执行符号解析函数。

2 ELF 格式的共享库结构

ELF文件采用分层管理模型，从程序装载器的角度看，是段(segments)的集合[1]。ELF文件都有一个ELF文件头，文件头说明文件中程序头部表格(Program Headers Table)和节区头部表格(Section Headers Table)在文件中的分布信息，2 个表格又说明各个段和各个节区在文件中的分布信息。

2.1 ELF 共享目标文件的段

作为共享库，ELF 使用程序头部表格管理文件内容，将文件分成多个段。每个段都有一个类型，并且说明了在运行时要求加载映射的内存地址、大小和权限。可执行文件的段通常包含绝对代码，为了能够让进程正确执行，段必须加载到构造可执行文件时所使用的虚拟地址。但是共享目标文件的代码段是位置无关的，因此段的实际加载虚拟地址可以和文件管理信息中指定的加载地址不同，不影响执行行为。ELF 文件重要的段如表 1 所示。

表 1 程序头部表格中的段类型

类型	取值	说明
PT_LOAD	1	此数组给出一个可加载的段
PT_DYNAMIC	2	给出动态链接的信息表格
PT_INTERP	3	给出一个程序解释器的位置字符串
PT_PHDR	6	程序头部表格的映射地址和大小

2.2 ELF 共享目标文件的动态项

类型为 PT_DYNAMIC 的段是一个非常重要的段，它是一个结构数组，给出动态链接的所有管理信息。每条动态项的结构包含一个类型字段 d_tag、一个虚拟地址 d_ptr 或者整数值 d_val 的联合体 d_un。标志 d_tag 说明该动态项所代表的对象，d_ptr 给出该对象的虚拟地址。对于共享文件，d_ptr 可能与加载时的内存虚拟地址不匹配，因此在加载时要由动态链接器修正为实际的虚拟地址。

可执行文件和共享目标文件中主要的动态项有 DT_NEEDED(动态库依赖表)、DT_PLTGOT(全局偏移表)、DT_HASH(哈希表)、DT_SYMTAB(符号表)、DT_STRTAB(字符串表)、DT_REL(重定位表)、DT_JMPREL(与过程链接表对应的重定位项表)。

3 动态库的装载和解析

操作系统运行程序时，首先将解释器程序即动态链接器 ld.so 映射到一个合适的地址，然后启动 ld.so。ld.so 先完成自己的初始化工作，再从可执行文件的动态库依赖表中指定的路径名查找所需要的库，将其加载映射到内存。

3.1 动态库的加载映射过程

Linux 用一个全局的库映射信息结构 struct link_map 链表来管理和控制所有动态库的加载，动态库的加载过程实际上是映射库文件到内存中，并填充库映射信息结构添加到链表中的过程。结构 struct link_map 描述共享目标文件的加载映射信息，是动态链接器在运行时内部使用的一个结构，通过它保持对已装载的库和库中符号的跟踪。

link_map 使用双向链接中间件“l_next”和“l_prev”链接进程中所有加载的共享库。当动态链接器需要去查找符号的时候，可以向前或向后遍历这个链表，通过访问链表上的每一个库去搜索需要查找的符号。Link_map 链表的入口由每个可执行映像的全局偏移表的第 2 个入口(GOT[1])指向，查找符号时先从 GOT[1]读取 link_map 结点地址，然后沿着 link-map 结点进行搜索。

动态库的加载映射过程主要分 3 步：

(1)动态链接器调用__mmap 函数对动态库的所有 PT_LOAD 可加载段进行整体映射：

```
l_map_start=(ElfW(Addr))__mmap ((void *)0, maplength, prot, MAP_COPY | MAP_FILE, fd, mapoff);
```

返回值 l_map_start 是实际映射的虚拟地址，和段结构成员 p_vaddr 指定的虚拟地址不一定相同，这对于位置无关代码不会产生影响。但是对于数据段和 link_map 结构中其它相关的位置描述信息还要进行修正。共享库映射的内存位置关系如图 1，l_addr 是实际映射地址和原来指定的映射地址的差值，用于其它位置信息的修正，即简单地将原来指定的虚拟地址加上 l_addr 就可以得到实际加载的虚拟地址。

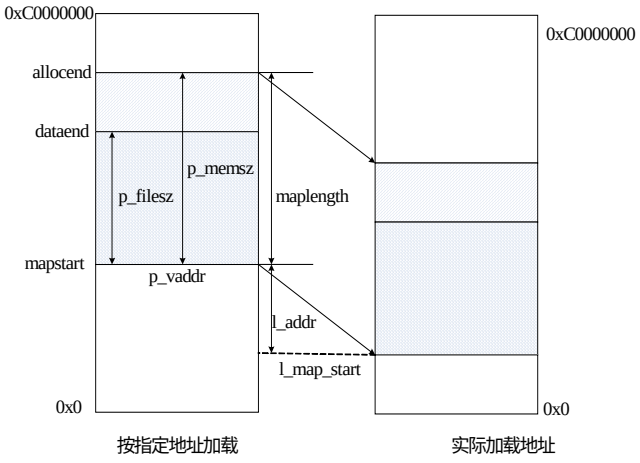

图 1 共享库文件内存加载示意图

(2)共享文件映射完毕，动态链接器处理共享库的 PT_DYNAMIC 动态段，将各项动态链接信息主要是哈希表、符号表、字符串表、重定位表、PLT 重定位项表等地址填写到 link_map 的 l_info 数组结构中。l_info 是 link_map 最重要的字段之一，几乎所有与动态链接管理相关的内容都与 l_info 数组有关。动态链接器还要加载处理当前共享库的所有依赖库。

(3)由于实际的映射地址和指定的虚拟地址有可能不同，因此还要对动态库及其依赖库进行重定位。设置动态库的第 1 个和第 2 个 GOT 表项：

```
Elf32_Addr *got =
(Elf32_Addr *) lmap->l_info[DT_PLTGOT].d_un.d_ptr;
 got[1]=lmap;
 got[2]=&_dl_runtime_resolve;
```

对动态库的所有重定位项进行重定位，在重定位项指定的偏移地址处加上修正值 l_addr。动态项 DT_REL 给出了重定位表的地址，DT_RELSZ 给出重定位表项的数目。

映射完毕后，动态链接器调用共享库(包括所有相关的依赖库)自备的初始化函数进行初始化。

3.2 函数符号的解析过程

当本运行模块要引用其它共享库中的函数时要进行符号解析和重定位。动态链接库在第 1 次调用函数时，经由 PLT 表跳转到 GOT[2]设置的入口_dl_runtime_resolve，又调用执行具体解析工作的 fixup 函数。在跳入前栈上已压入了 2 个

值作为函数 fixup 的参数：(1)要解析的符号在 DT_JMPREL 重定位表中的偏移，间接指明了要解析的符号以及该符号解析到哪个 GOT 项去；(2)动态链接库 struct link_map*指针。函数 fixup 返回时，将函数地址的解析值作为返回地址又跳转到所解析的函数中执行。

函数 fixup 遍历 link_map 链表上的每个库节点，查找要解析的符号。先要根据 reloc_offset 在本运行模块内查找要解析的符号名称：根据 DT_JMPREL 段的段内偏移 reloc_offset 找到要解析符号的重定位项结构信息 reloc；再根据 reloc 的 reloc->r_info 字段找到相应的符号表和字符串表的索引。根据参数 reloc_offset 查找符号名称的过程如图 2。参数 reloc_offset 另一方面也给出了在本运行模块内要重定位的地方，即由重定位项字段 reloc->r_offset 指向的相应 GOT 表项。

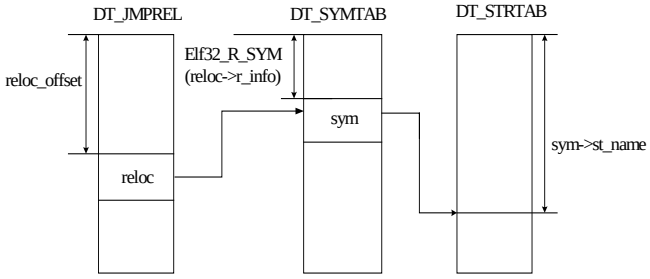

图 2 由重定位偏移查找要解析的符号名称

函数 fixup 遍历各个动态库的每个符号表，查找与给定解析符号字符串相符的符号表项 sym，最后重定位：

```
*( reloc->r_offset + lmap->l_addr) =
sym->st_value+search_lmap->l_addr;
```

加上修正值 l_addr 是为了得到运行时的重定位地址或符号地址，其中 lmap 和 search_lmap 分别是本运行模块和当前搜索库的 link_map 指针。

4 共享库的重定向

利用动态链接机制还可以实现共享库的重定向等重要应用[3]。动态库重定向通过某种操作在目标进程中装载指定的共享库，再配合以函数重定向等技术，使我们能观察、捕获、改变或控制目标进程的行为。通过共享库重定向可以为程序在运行期打补丁，目标程序或服务不需要重新启动，非常方便和安全。

实现共享库重定向要对目标进程进行控制操作。Linux 下通过系统调用ptrace，一个进程可以动态地读写另一个进程的内存和寄存器[4]，从而可以跟踪、调试和控制运行中的进程。Linux的可执行映像在用户空间的映射总是从 0x8048000 开始，因为gcc编译器的默认链接脚本从该地址开始链接各个目标文件。因此可以用ptrace从该地址读取目标进程的可执行映像，进一步找到目标进程中的link_map链表，就可以任意解析目标进程内的可执行映像和所有共享库中的函数符号。实现原理和步骤：

(1)调用 ptrace(PTRACE_ATTACH, pid, NULL, NULL)关联到目标进程。

(2)查找目标进程内装载共享库的函数_dl_open。只有使用目标进程内的库装载函数_dl_open，才能将自己的共享库装载到目标进程的地址空间中。函数_ld_open 在 ld.so 动态库中，Linux 下每个程序执行时，都先要把 ld.so 映射到自己的进程地址空间中。

(3)运行函数_dl_open 装载自己的共享库。在目标进程的系统堆栈上的运行场景设置相应的寄存器内容作为参数和运行地址：

```
regs.eax=const char *libname;
regs.ecx=const void *caller (NULL);
regs.edx=int mode (RTLD_LAZY);
regs.eip=_dl_open;
```

当目标进程恢复运行，恢复进入内核空间时，保存运行场景，执行函数_dl_open，装载我们自己的共享库到目标进程的地址空间中。

(4)函数重定向。

查找原来执行映像中旧函数的重定位地址和共享库中新函数的地址，用新函数的地址覆盖原来 GOT 表项的内容，就实现了函数的重定向。函数重定向前后的跳转关系如图 3，重定向之后当执行函数 printf 时，实际执行的是新共享库中的函数。

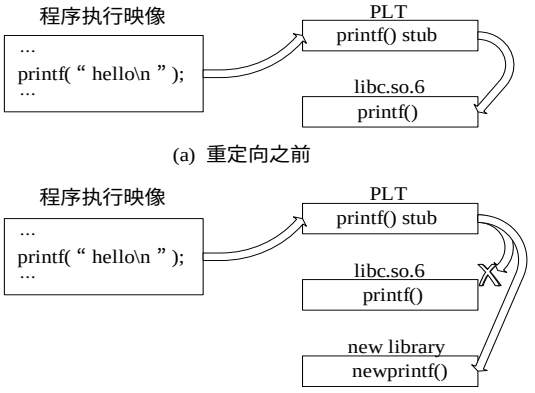

图 3 函数重定向前后的跳转关系

5 结论

Linux 动态链接机制的运行特点类似于操作系统对物理内存的 COW(copy on wirte)操作，在达到共享资源的目的的同时，极大地提高了程序运行的性能。根据 Linux 动态链接机制实现的共享库重定向对 Linux 软件的升级和维护和服务器级的应用有着重要的意义，其安全问题的研究仍将是今后的重点。

参考文献

1 Tool Interface Standard. Executable and Linktable Format Specification(Version 1.2)[S]. 1995.

2 Levine J R. Linkers and Loaders[M]. San Francisco: Morgan-Kauffman, 1999.

3 bh-europe-01-clowes.ppt[Z]. http://www.blackhat.co.m/presentations/bh-europe-01/shaun-clowes, 2002.

4 毛德操, 胡希明. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2001.