Linux Dynamic Shared Library && LD Linker

.Little Hann 2015-01-29 原文

目录

1. 动态链接的意义

1. . 静态链接对内存和磁盘的浪费很严重，在静态链接中，C语言静态库是很典型的占用空间的例子
2. . 静态链接对程序的更新、部署、发布会造成严重的麻烦

为了解决这些问题，最好的思路就是把程序的模块相互分割开来，形成独立的文件，而不再将它们静态地链接在一起。简单来说，就是不对那些组成程序的目标文件进行链接，等到程序要运行时才进行链接，也就是说，把链接这个过程推迟到了运行时再进行，这就是"动态链接(dynamic linking)"的基本思想

0x1: 动态链接的优点

1. . 多个进程使用到同一个动态链接库文件，只要在内存中映射一份ELF .SO文件即可，有效地减少了进程的内存消耗
2.

```
3.  . 减少物理页面的换入换出(减少page out、page in操作)
4.
5.  . 增加CPU缓存的命中率，因为不同进程间的数据和指令访问都集中在了同一个共享模
    块上
6.
7.  . 使程序的升级更加容易，在升级程序库或共享某个模块时，只要简单地将旧的目标文
    件覆盖掉，而无须将所有的程序再重新链接一遍。当程序下一次运行的时候，新版本的
    目标文件会被自动装载到内存并链接起来，程序就完成了升级的操作
8.
9.  . 程序可扩展性和兼容性
10. 使用动态链接技术，程序在运行时可以动态地选择加载各种程序模块，即插件技术
    (Plug-in)
11.     ) 程序按照一定的规则制定好程序的接口，第三方开发者可以按照这种接口来编写
    符合要求的动态链接文件，该程序可以动态地载入各种由第三方开发的模块，在程序运
    行时动态地链接，实现程序功能的扩展。典型地如php的zend扩展、iis的
    filter/extension、apache的mod模块
12.     ) 动态链接还可以加强程序的兼容性。一个程序在不同的平台运行时可以动态地链
    接到由操作系统提供的动态链接库，这些动态链接库在程序和操作系统之间增加了一个
    中间层，从而消除了程序对不同平台之间依赖的差异性
```

0x2: 动态链接文件的类别

动态链接涉及运行时的链接及多个文件的装载，必须要有操作系统的支持，因为动态链接的情况下，进程的虚拟地址空间的分布会比静态链接的情况下更为复杂，还需要考虑到一些存储管理、内存共享、进程线程等机制的考虑

```
1.  . Linux
2.  在Linux系统中，ELF动态链接文件被称为动态共享对象(DSO Dynamic Shared
    Objects)，一般以".so"为扩展名
3.  常用的C语言库的运行库glibc，它的动态链接形式的版本保存在"/lib/libc.so"、"
    /lib64/libc.so"。整个系统只保留一份C语言库的动态链接文件，而所有的由C语言
    编写的、动态链接的程序都可以在运行时使用它，当程序被装载时，系统的动态链接器
    会将程序所需的所有动态链接库(最基本的就是libc.so)装载到进程的地址空间，并且
    将程序中所有未决议的符号绑定到相应的动态链接库中，并进行重定位工作
4.
5.  . Windows
6.  在Windows系统中，动态链接文件被称为动态链接库(Dynamic Linking
    Library)，一般以".dll"为扩展名
```

Relevant Link:

2. 地址无关代码: PIC

```
1.  . 可执行文件在编译时可以确定自己在进程虚拟地址空间中的位置，因为可执行文件往
    往都是第一个被加载的文件，它可以选择一个固定的位置
2.      ) Linux: 0x08040000
3.      ) Windows: 0x0040000
4.
5.  . 共享对象在编译时不能假设自己在进程虚拟地址空间中的位置
```

0x1: 装载时重定位

Linux和GCC支持2种重定位的方法

```
1.  . 链接时重定位(Link Time Relocation)
2.  -shared -fPIC
3.  在程序链接的时候就将代码中对绝对地址的引用重定位为实际的地址
4.
5.  . 装载时重定位(Load Time Relocation)
6.  -shared
7.  程序模块在编译时目标地址不确定而需要在装载时将模块重定位
```

0x2: 地址无关代码

装载时重定位是解决动态模块中有绝对地址引用的方法之一，但是还存在一个问题，指令部分无法在多个进程间共享，为了解决这个问题，一个基本思想就是把指令中那些需要被修改的部分分离出来，跟数据部分放在一起，这样指令就可以保持不变，而数据部分可以在每个进程中拥有一个副本，这种方案就是地址无关代码(PIC Position-Independent Code)

我们把共享对象模块中的地址引用按照模块内部引用/模块外部引用、指令引用/数据访问分为4类

```
1.  /*
2.  pic.c
3.  */
4.  static int a;
5.  extern int b;
6.  extern void ext();
7.
8.  void bar()
9.  {
10.     //Type2: Inner-module data access(模块内数据访问)
11.     a = ;
12.
13.     //Tyep4: Inter-module data access(模块间数据访问)
14.     b = ;
15. }
16.
17. void foo()
18. {
19.     //Type1: Inner-module call(模块内指令引用)
20.     bar();
21.
22.     //Type3: Inter-module call()
23.     ext();
24. }
```

值得注意的是，当编译器在编译pic.c时，它并不能确定变量b、函数ext()是模块外部还是模块内部的，因为它们有可能被定义在同一个共享对象的其他目标文件中，所以编译器只能把它们都当作模

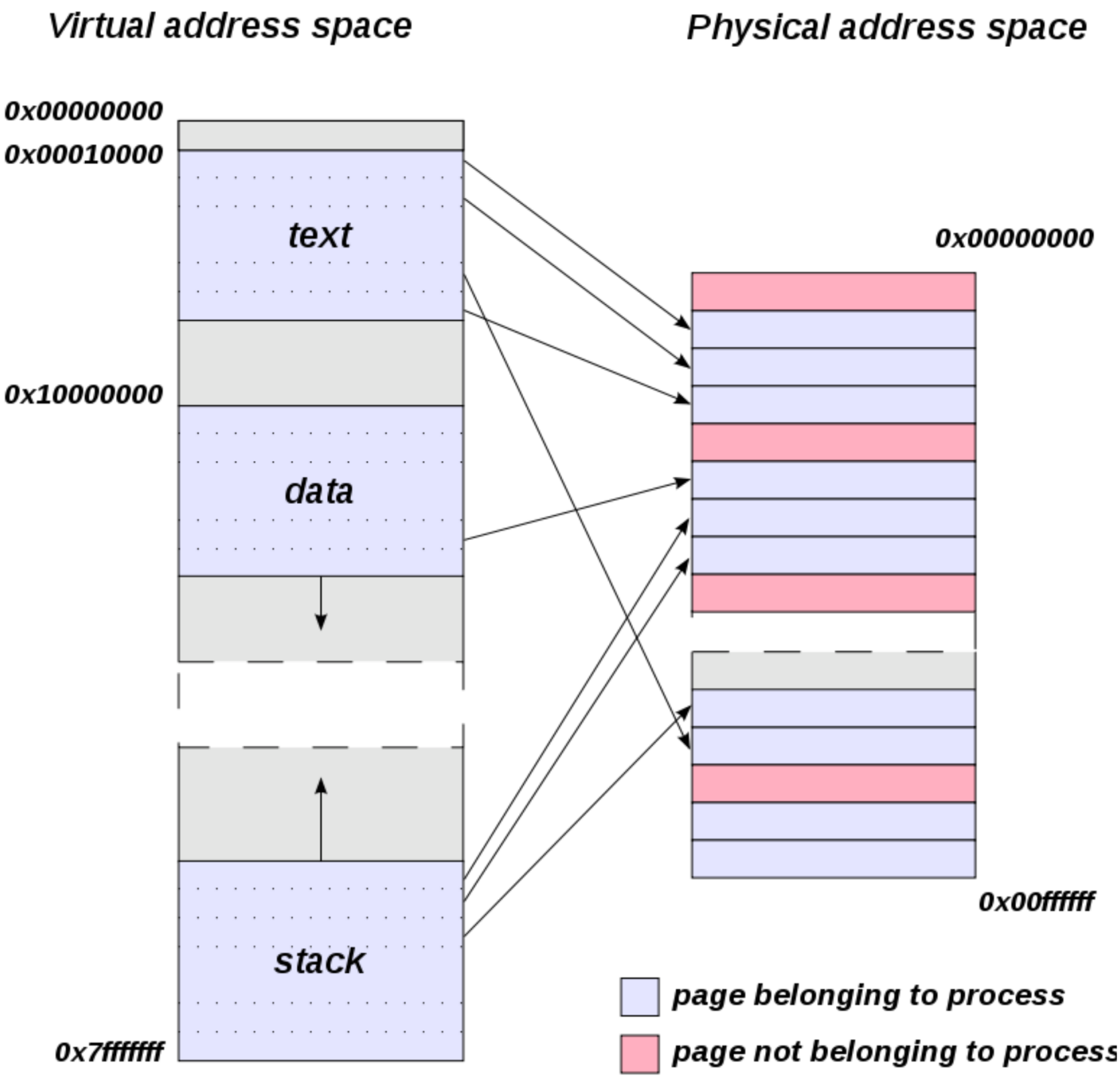
块外部的函数和变量来处理

Type1: Inner-module call(模块内指令引用)

这是最简单的一种情况，被调用的函数与调用者都处于同一个模块，它们之间的相对位置是固定的，对于现代操作系统来说，模块内部跳转、函数调用都可以是"相对地址调用"、或者是"基于寄存器的相对调用"，所以对于这种指令是不需要重定位的，只要模块内的相对位置不变，则模块内的指令调用就是地址无关的

Type2: Inner-module data access(模块内数据访问)

我们知道，一个模块前面一般是若干个页的代码，后面紧跟着若干个页的数据，这些页之间的相对位置是固定的，所以只需要相对于当前指令加上"固定的偏移量"就可以访问到模块内部数据了



Type3: Inter-module call()

GOT实现指令地址无关的方式和GOT实现模块间数据访问的方式类似，唯一不同的是，GOT中的项保存的是目标函数的地址，当模块要调用目标函数时，可以通过GOT中的项进行间接跳转

Tyep4: Inter-module data access(模块间数据访问)

模块间的数据访问比模块内部稍微麻烦一点，因为模块间的数据访问目标地址要等到装载时才能确定。而我们要达到代码地址无关的目的，最基本的思想就是把和地址相关的部分放到数据段中，ELF的做法是在数据段里建立一个指向这些变量的指针数组，也被称为全局偏移表(global offset table GOT)，当代码需要引用到该全局变量时，可以通过GOT中相对应的项进行间接引用。链接器在装载动态模块的时候会查找每个变量所在的地址，然后填充GOT中的各个项，以确保每个指针所指向的地址正确，由于GOT本身是放在数据段的，所以它可以在模块装载时被修改，并且每个进程都可以有独立的副本，相互不受影响。
综上所述，地址无关代码的实现方式如下

```
1.  . 模块内部
2.      ) 指令跳转、调用: 相对跳转和调用
3.      ) 数据访问: 相对地址访问
4.  . 模块外部
5.      ) 指令跳转、调用: 间接跳转和调用(GOT)
6.      ) 数据访问: 间接访问(GOT)
```

使用GCC产生地址无关代码很简单，只需要使用"-fPIC"参数即可

区分一个DSO是否为PIC的方法很简单，输入以下指令

```
1.  readelf -d hook.so | grep TEXTREL
2.  /*
3.  1. PIC
4.  PIC的DSO是不会包含任何代码段重定位表的，TEXTREL表示代码段重定位表地址
5.
6.  2. 非PIC
7.  本条指令有任何输出，则hook.so就不是PIC
8.  */
```

地址无关代码技术除了可以用在共享对象上面，它也可以用于可执行文件，一个以地址无关方式编译的可执行文件被称作地址无关可执行文件(PIE Position-Independent Executable)，与GCC的"-fPIC"类似，产生PIE的参数为"-fPIE"

0x3: PIC

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

0X4: 全局偏移表(GOT)

```
1.  . 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
2.  . 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真实的绝对位置，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

```
1.  . R_386_RELATIVE
2.  . R_386_GLOB_DAT
3.  . 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

0x5: 过程链接表(PLT)

过程链接表(PLT)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，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

```
1.  . GOT+(即 GOT[]): 设置动态库映射信息数据结构link_map地址
2.  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。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
3.  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
4.  //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
5.
6.  . GOT+(即 GOT[]): 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[]存储的地址执行符号解析函数。待完成符号解析后，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

动态库的加载映射过程主要分3步

```
. 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

. 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
```

Relevant Link:

```
http://zhiwei.li/text/2009/04/elf%E7%9A%84got%E5%92%8Cplt%E4%BB%A5%E5%8F%8Apic/#comment-4235
http://www.programlife.net/linux-got-plt.html
```

3. 延迟版定(PLT Procedure Linkage Table)

我们知道，动态链接比静态链接慢的主要原因有如下几个

```
. 动态链接下对于全局和静态的数据访问都要进行复杂的的GOT定位，然后间接寻址，对于模块间的调用也要先定位GOT，然后再进行间接跳转
. 动态链接的链接工作是在运行时完成的，动态链接器会寻找并装载所需要的共享对象，然后进行符号查找地址重定位工作等
```

0x1: 延迟绑定的实现

在动态链接下，程序模块间包含了大量的函数引用，所以在程序开始执行前，动态链接器会耗费大量时间用于解决模块间的函数引用的符号查找以及重定位。但是需要明白的是，在一个程序运行过程中，可能很多函数在程序执行完时都不会被用到，例如一些错误处理函数或者是一些很少运行到的代码逻辑流支，如果一开始就把所有函数都链接好实际上是一种浪费，所以ELF采用了一种延迟绑定(Lazy Binding)技术，即当函数第一次被用到时才进行绑定(符号查找、重定位等)，如果这个函数没有被用到则不进行绑定。

采用了延迟绑定技术后，程序开始运行时，模块间的函数调用全都没有进行绑定，而是需要用到时才由动态链接器来负责绑定

ELF使用PLT(Procedure Linkage Table)的方法来实现，在Glibc中，实现延迟绑定功能的函数名叫"_dl_runtime_resolve()"

在开始学习PLT技术之前，我们来总结一下ELF中这种技术的核心思想

1. 不管是模块间的指令调用、还是跨模块的全局静态变量的引用，ELF使用了GOT间接跳转来实现，本质上是使用了"中间层技术"来屏蔽可能存在的外部模块引入的不确定性，中间层技术是实现兼容的一种很好的思考方式

PLT为了实现延迟绑定，在GOT的基础之上又增加了一层间接跳转，调用函数并不直接通过GOT跳转，而是通过一个叫做PLT项的结构来进行跳转。每个外部函数在PLT中有一个相应的项

4. 动态链接相关结构

在动态链接情况下，可执行文件的装载与静态链接的情况基本一样

1. ．操作系统读取可执行文件的头部，检查文件的合法性
2. ．从头部中的"Program Header"中读取每个"Segment"的虚拟地址、文件地址和属性，并将它们映射到进程虚拟空间的相对位置
3. ．在静态链接情况下，这个时候操作系统就可以把控制权交给可执行文件的入口地址，然后程序开始执行

但是在动态链接情况下，操作系统不能在装载完可执行文件之后就把控制权交给可执行文件，因为可执行文件依赖于很多动态共享对象(DSO)，这个时候，可执行文件对于很多外部符号的引用还处于无效地址的状态，即还没有跟相应的共享对象中的实际位置链接起来，所以在映射完可执行文件之后，操作系统会先启动一个动态链接器(Dynamic Linker)

在Linux中，动态链接器ld.so实际上也是一个共享对象

1. ．操作系统同样通过映射的方式将它加载到进程的地址空间中

2. ．操作系统在加载完动态链接器之后，就将控制权交给动态链接器的入口地址(与可执行文件一样，共享对象也有入口地址)
3. ．当动态链接器得到控制权之后，它开始执行一系列自身的初始化操作，然后根据当前的环境参数，开始对可执行文件进行动态链接工作
4. ．当所有动态链接工作完成之后，动态链接器会将控制权转交到可执行文件的入口地址，程序开始正式执行

0x1: .interp段

值得注意的是，动态链接器的位置既不是系统配置决定、也不是由环境参数决定，而是由ELF文件自身决定。在动态链接的ELF可执行文件中，有一个专门的段叫作 ".interp段"(interpreter(解释器)段)

```
objdump -s main
```

```
main:     file format elf64-x86-64

Contents of section .interp:
 400200 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400210 7838362d 36342e73 6f2e3200           x86-64.so.2.
```

".interp"里保存的就是一个字符串，表明可执行文件所需要的动态链接器的路径，在Linux中，操作系统在对可执行文件进行加载的时候，会去寻找装载该可执行文件所需要的相应的动态链接器，即".interp"段指定的路径的共享对象

动态链接器在Linux下是Glibc的一部分，也是属于系统库级别的，它的版本号往往跟系统中的Glibc库版本号一致，当系统中的Glibc库更新或者安装其他版本的时候，/lib64/ld-linux.so.2这个软链接就是指向到新的动态链接器，而可执行文件本身不需要修改".interp"段中的动态链接器的路径来适应系统的升级，这又是利用中间层思想带来的兼容性的一个例子

0x2: .dynamic段

动态链接器ELF中最重要的结构应该是".dynamic"段，这个段里面保存了动态链接器所需要的基本信息，例如依赖哪些共享对象、动态链接符号表的位置动态链接重定位表的位置、共享对象初始化代码的地址等

linux-2.6.32.63\include\linux\elf.h

```
typedef struct dynamic
{
    Elf32_Sword d_tag;
    union
    {
        Elf32_Sword d_val;
        Elf32_Addr d_ptr;
    } d_un;
```

```
} Elf32_Dyn;

typedef struct
{
    /*
    entry tag value : 类型值
        #define DT_NULL         0
        #define DT_NEEDED    1
        #define DT_PLTRELSZ    2
        #define DT_PLTGOT    3
        #define DT_HASH         4 : 动态链接哈希表地址，d_ptr表示".hash"的地址
        #define DT_STRTAB    5 : 动态链接字符串表的地址，d_ptr表示".dynstr"的地址
        #define DT_SYMTAB    6 : 动态链接符号表的地址，d_ptr表示".dynsym"的地址
        #define DT_RELA         7 : 动态链接重定位表地址
        #define DT_RELASZ    8
        #define DT_RELAENT    9
        #define DT_STRSZ    10 : 动态链接字符串表大小，d_val表示大小
        #define DT_SYMENT    11
        #define DT_INIT         12 : 初始化代码地址
        #define DT_FINI         13 : 结束代码地址
        #define DT_SONAME    14 : 本共享对象的"SO-NAME"
        #define DT_RPATH     15 : 动态链接共享对象搜索路径
        #define DT_SYMBOLIC    16
        #define DT_REL             17
        #define DT_RELSZ    18
        #define DT_RELENT    19 : 动态重读位表入口数量
        #define DT_PLTREL    20
        #define DT_DEBUG    21
        #define DT_TEXTREL    22
        #define DT_JMPREL    23
        #define DT_ENCODING    32
        #define OLD_DT_LOOS    0x60000000
        #define DT_LOOS         0x6000000d
        #define DT_HIOS         0x6ffff000
        #define DT_VALRNGLO    0x6ffffd00
        #define DT_VALRNGHI    0x6ffffdff
        #define DT_ADDRRNGLO    0x6ffffe00
        #define DT_ADDRRNGHI    0x6fffffff
        #define DT_VERSYM    0x6ffffff0
        #define DT_RELACOUNT    0x6ffffff9
        #define DT_RELCOUNT    0x6ffffffa
        #define DT_FLAGS_1    0x6ffffffb
        #define DT_VERDEF    0x6ffffffc
        #define    DT_VERDEFNUM    0x6ffffffd
        #define DT_VERNEED    0x6ffffffe
        #define    DT_VERNEEDNUM    0x6fffffff
        #define OLD_DT_HIOS     0x6fffffff
        #define DT_LOPROC    0x70000000
        #define DT_HIPROC    0x7fffffff
    */
    Elf64_Sxword d_tag;
    union
    {
```

```
62.             Elf64_Xword d_val;
63.             Elf64_Addr d_ptr;
64.         } d_un;
65.     } Elf64_Dyn;
```

从作用上来说，".dynamic"段里保存的信息类似于ELF文件头，使用readelf -d hook.so可以查看".dynamic"段的内容

Linux还提供了一个指令来查看一个程序主模块、或者一个共享库依赖于哪些共享库: ldd programe

0x3: 动态符号表

为了完成动态链接，最关键的是所依赖的符号和相关文件的信息。为了表示动态链接这些模块之间的符号导入导出关系，ELF专门有一个叫作动态符号表(dynamic symbol table)的段，这个段的段名通常为".dynsym"(dynamic symbol)。与".symtab"类似，动态符号表也需要一些辅助的表，比如用于保存符号名的字符串表，即动态符号字符串表".dynstr"(dynamic string table)，由于在动态链接下，我们需要在程序运行时查中啊符号，为了加快符号的查找过程，往往还有辅助的符号哈希表".hash"

0x4: 动态链接重定位表

动态链接下，无论是可执行文件还是共享对象，只要它依赖于其他共享对象，也就是说有导入的符号时，那么它的代码或数据中就会有对于导入符号的引用，在编译时这些导入符号的地址未知，在静态链接中，这些未知的地址引用在最终链接时会被重定位修正，但是在动态链接中，导入符号的地址在运行时才确定，所以需要在运行时将这些导入符号的引用修正，即需要动态重定位

```
1.  . ".rel.dyn"
2.  对数据引用的修正，它所修正的位置位于".got"以及数据段
3.
4.  . ".rel.plt"
5.  对函数引用的修正，它所修正的位置位于".got.plt"
```

0x5: 动态链接时进程堆栈初始化信息

进程初始化的时候，堆栈里保存了关于进程执行环境和命令行参数等信息，除此之外，堆栈里还保存了动态链接器所需要的一些辅助信息数组(auxiliary vevtor)

linux-2.6.32.63\include\linux\elf.h

```
1.  typedef struct
2.  {
3.      /*
4.      Entry type
5.          #define AT_NULL 0 : 表示辅助信息数组结束
```

```
            #define AT_IGNORE 1
            #define AT_EXECFD 2 : 表示可执行文件的文件句柄
            #define AT_PHDR 3 : 可执行文件中"程序头表(program header)"在进程中的地址
            #define AT_PHENT 4 : 可执行文件中程序头表每一个入口(entry)的大小
            #define AT_PHNUM 5 : 可执行文件头中程序头表中入口(entry)的数量
            #define AT_PAGESZ 6
            #define AT_BASE 7 : 表示动态链接器本身的装载地址
            #define AT_FLAGS 8
            #define AT_ENTRY 9 : 可执行文件入口地址，即启动地址
            #define AT_NOTELF 10
            #define AT_UID 11
            #define AT_EUID 12
            #define AT_GID 13
            #define AT_EGID 14
            #define AT_CLKTCK 17
        */
        u64 a_type;
        union
        {
            u64 a_val;                  /* Integer value */

        } a_un;
    } Elf64_auxv_t;
```

事实上，辅助信息位于环境变量指针的后面

```
    #include <stdio.h>
    #include <elf.h>

    int main(int argc, char* argv[])
    {
        int* p = (int*)argv;
        int i;
        Elf32_auxv_t* aux;

        printf());

        ; i < *(p - ); i++)
        {
            printf());
        }

        p += i;
        p++;//skip 0

        printf("Environment: \n");
        while(*p)
        {
            printf("%s\n", *p);
            p++;
        }

        p++;//skip 0
```

```
28.
29.         printf("Auxiliary Vectors: \n");
30.         aux = (Elf32_auxv_t*)p;
31.         while(aux->a_type != AT_NULL)
32.         {
33.             printf("Type: %02d Value: %x\n", aux->a_type,
    aux->a_un.a_val);
34.             aux++;
35.         }
36.
37.         ;
38.     }
```

5. 动态链接的步骤和实现

动态链接基本上分为3步

```
1.  . 启动动态链接器本身(自举)
2.  . 装载所有需要的共享对象
3.  . 重定位、初始化
```

0x1: 动态链接器自举

我们知道，对于Linux程序中的普通共享对象(DSO)文件来说

```
1.  . 普通DSO的重定位工作由动态链接器来完成
2.  . 普通DSO依赖的其他共享对象由动态链接器负责链接和装载
```

而对于动态链接器对应的DSO文件来说

```
1.  . 动态链接器本身不可以依赖于其他任何共享对象
2.  编写动态链接器时保证不使用任何系统库、运行库
3.
4.  . 动态链接器本身所需要的全局和静态变量的重定位工作由它本身完成
```

动态链接器必须在启动时有一段很精巧的代码可以完成这项艰巨的工作同时又不能用到全局和静态变量。这种具有一定限制条件的启动代码往往被称为"自举(Boosttrap)"

```
1.  . 动态链接器入口地址就是自举代码的入口，当操作系统将进程控制权交给动态链接器
    时，动态链接器的自举代码即开始执行
2.  . 自举代码会找到自己的GOT。而GOT的第一个入口保存的即是".dynamic"段的偏移地
    址，由此获得了动态链接器本身的".dynamic"段
3.  . 通过".dynamic"段中的信息，自举代码便可以获得动态链接器本身的重定位表和符
    号表等，从而得到动态链接器本身的重定位入口，先将它们全部重定位
4.  . 从这一步开始动态链接器代码中才可以开始使用自己的全局变量和静态变量
```

0x2: 装载共享对象

完成基本自举后，动态链接器将可执行文件和链接器自身的符号表都合并到一个符号表中，我们称之为"全局符号表(Global Symbol Table)"。然后链接器开始寻找可执行文件所依赖的共享对象，在".dynamic"段中，有一种类型的入口是DT_NEEDED，它标识了该可执行文件(或共享对象)所依赖的共享对象。由此

```
1. ．链接器可以列出可执行文件所需要的所有共享对象，并将这些共享对象的名字放入到一个装载集合中
2. ．然后链接器开始从集合里取一个所需要的共享对象的名字，找到相应的文件后打开该文件，读取相应的ELF文件头和".dynamic"段，然后将它相应的代码段和数据段映射到进程空间中
3. ．如果这个ELF共享对象还依赖于其他共享对象，那么将所依赖的共享对象的名字放到装载集合中，如果循环知道所有依赖的共享对象都被装载进来为止
4. ．链接器对共享对象的遍历过程本质上是一个图的遍历过程，链接器可能会使用深度优先、或者广度优先的顺序来进行
5. ．当一个新的共享对象被装载进来的时候，它的符号表会被合并到全局符号表中，所以当所有的共享对象都被装载进来的时候，全局符号表里面将包含进程中所有动态链接锁需要的符号
```

符号优先级

在动态链接器按照各个模块之间的依赖关系，对它们进行装载并且将它们的符号"合并"到全局符号表时，会发生两个不同的模块定义了一个同名的符号
编写示例代码模拟这个场景

```
1. ．定义一个简单的输出demo函数(同名)
2. /*
3. a1.c
4. */
5. #include <stdio.h>
6.
7. void a()
8. {
9.     printf("a1.c\n");
10. }
11.
12. /*
13. a2.c
14. */
15. #include <stdio.h>
16.
17. void a()
18. {
19.     printf("a2.c\n");
20. }
21. 可以看到，a1.c、a2.c中都定义了名为"a"的函数
22.
23. ．显式指定依赖关系
24. /*
25. 代码中调用a()
26. b1.c
27. */
```

```
void a();

void b1()
{
    a();
}

/*
代码中调用a()
b2.c
*/
void a();

void b2()
{
    a();
}

. 根据依赖关系进行编译
//b1.so依赖a1.so
gcc -fPIC -shared a1.c -o a1.so
gcc -fPIC -shared b1.c a1.so -o b1.so

//b2.so依赖a2.so
gcc -fPIC -shared a2.c -o a2.so
gcc -fPIC -shared b2.c a2.so -o b2.so

. 同时引入b1.so、b2.so，模拟同名符号冲突的情况
/*
main.c
*/
#include <stdio.h>

void b1();
void b2();

int main()
{
    b1();
    b2();
    ;
}
gcc main.c b1.so b2.so -o main -Xlinker -rpath ./
./main
```

```
[root@localhost test]# ./main
a1.c
a1.c
```

当动态链接器对main程序进行动态链接时，b1.so、b2.so、a1.so、a2.so都会被装载到进程的地址空间中，并且它们的符号都会被"合并"到全局符号表中，当发生同名符号重合的情况时，这种现象叫做"全局符号介入(global symbol interpose)"

Linux下的动态链接器的处理规则是这样的

```
1.  . 当一个符号需要被加入到全局符号表时，如果相同的符号名已经存在，则后载入的符号被忽略
2.  . 按照广度优先的顺序进行加载
3.  . 如果优先引入的函数(符号)没有显式地进行调用链获取，并在完成自身逻辑后调用调用链上的下一个函数，则后引入的函数(符号)将被忽略
4.  #define FN(ptr,type,name,args)  ptr = (type (*)args)dlsym (REAL_LIBC, name)
```

0x3: 重定位和初始化

当完成动态链接器的装载、普通共享对象的装载之后，链接器开始重新遍历可执行文件和每个共享对象的重定位表，将它们的GOT/PLT中的每个需要重定位的位置进行修正
重定位完成后就，如果某个共享对象有".init"段，那么动态链接器会执行".init"段中的代码，用以实现共享对象特有的初始化过程，例如共享对象中的C++全局/静态对象的构造就是通过"init"段来初始化

当完成了重定位和初始化后，所有的准备工作就宣告完成了，所需要的共享对象也都已经装载并且链接完成了，这个时候进程的控制权就由动态链接器转交给程序的入口并且开始执行

Relevant Link:

6. Linux动态链接器实现

Linux动态链接器本身是一个共享对象，它的路径是"/lib/ld-linux.so.2、/lib64/ld-linux-x86-64.so.2"。共享对象本质上也是一个ELF文件，包含ELF文件头(包括e_entry、段表等)，而动态链接器是个非常特殊的共享对象，它不仅是个共享对象，还是一个可执行程序，可以直接在命令行下运行

```
1.  /lib64/ld-linux-x86-.so.
```

```
[root@localhost 3789]# /lib64/ld-linux-x86-64.so.2
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]
You have invoked `ld.so', the helper program for shared library executables.
This program usually lives in the file `/lib/ld.so', and special directives
in executable files using ELF shared libraries tell the system's program
loader to load the helper program from this file.  This helper program loads
the shared libraries needed by the program executable, prepares the program
to run, and runs it.  You may invoke this helper program directly from the
command line to load and run an ELF executable file; this is like executing
that file itself, but always uses this helper program from the file you
specified, instead of the helper program file specified in the executable
file you run.  This is mostly of use for maintainers to test new versions
of this helper program; chances are you did not intend to run this program.

  --list                list all dependencies and how they are resolved
  --verify              verify that given object really is a dynamically linked
                        object we can handle
  --library-path PATH   use given PATH instead of content of the environment
                        variable LD_LIBRARY_PATH
  --inhibit-rpath LIST  ignore RUNPATH and RPATH information in object names
```

```
                        in LIST
 --audit LIST           use objects named in LIST as auditors
```

Linux的ELF动态链接器是glibc的一部分，它的源代码位于glibc的源代码的ELF目录下

\glibc-2.18\sysdeps\i386\dl-machine.h

```
/*
    Initial entry point code for the dynamic linker.
       The C function `_dl_start' is the real entry point;
       its return value is the user program's entry point.
 */

#define RTLD_START asm ("\n\
    .text\n\
    .align \n\
:    movl (%esp), %ebx\n\
    ret\n\
    .align \n\
.globl _start\n\
.globl _dl_start_user\n\
_start:\n\
    # Note that _dl_start gets the parameter in %eax.\n\
    movl %esp, %eax\n\
    call _dl_start\n\
_dl_start_user:\n\
    # Save the user entry point address in %edi.\n\
    movl %eax, %edi\n\
    # Point %ebx at the GOT.\n\
    call 0b\n\
    addl $_GLOBAL_OFFSET_TABLE_, %ebx\n\
    # See if we were run as a command with the executable file\n\
    # name as an extra leading argument.\n\
    movl _dl_skip_args@GOTOFF(%ebx), %eax\n\
    # Pop the original argument count.\n\
    popl %edx\n\
    # Adjust the stack pointer to skip _dl_skip_args words.\n\
    leal (%esp,%eax,), %esp\n\
    # Subtract _dl_skip_args from argc.\n\
    subl %eax, %edx\n\
    # Push argc back on the stack.\n\
    push %edx\n\
    # The special initializer gets called with the stack just\n\
    # as the application's entry point will see it; it can\n\
    # switch stacks if it moves these contents over.\n\
" RTLD_START_SPECIAL_INIT "\n\
    # Load the parameters again.\n\
    # (eax, edx, ecx, *--esp) = (_dl_loaded, argc, argv, envp)\n\
    movl _rtld_local@GOTOFF(%ebx), %eax\n\
    leal (%esp,%edx,), %esi\n\
    leal (%esp), %ecx\n\
    movl %esp, %ebp\n\
    # Make sure _dl_init  byte aligned stack.\n\
    andl $-, %esp\n\
```

```
48.     pushl %eax\n\
49.     pushl %eax\n\
50.     pushl %ebp\n\
51.     pushl %esi\n\
52.     # Clear %ebp, so that even constructors have terminated
    backchain.\n\
53.     xorl %ebp, %ebp\n\
54.     # Call the function to run the initializers.\n\
55.     call _dl_init_internal@PLT\n\
56.     # Pass our finalizer function to the user in %edx, as per ELF
    ABI.\n\
57.     leal _dl_fini@GOTOFF(%ebx), %edx\n\
58.     # Restore %esp _start expects.\n\
59.     movl (%esp), %esp\n\
60.     # Jump to the user's entry point.\n\
61.     jmp *%edi\n\
62.     .previous\n\
63. ");
```

执行流程如下

```
1. . _start()调用_dl_start()函数
2. . _dl_start()首先对ld-x.y.z.so进行重定位，因为ld-x.y.z.so自身是动态链接
   器，它必须自己完成重定位，即"自举"
3. . 完成自举后就可以调用其他函数、并且访问全局变量了
4. . 调用_dl_start_final()收集一些基本的运行数值，进入_dl_sysdep_start()
5. . _dl_sysdep_start()进行了一些平台相关的处理之后就进入了_dl_main()，这是
   动态链接器的主函数
```

Relevant Link:

```
1. http://mirror.hust.edu.cn/gnu/glibc/
```

7. 显式运行时链接

支持动态链接的系统大部分都支持一种更加灵活的模块加载方式，即"显式运行时链接(explicit run-time linking)(运行时加载)"。让程序自己在运行时控制加载指定的模块，并且可以在不需要该模块时将其卸载

在Linux中，从文件本身的格式上来看，动态库实际上和共享对象库没有区别，主要的区别是

```
. 共享对象是由动态链接器在程序启动之前负责装载和链接的，这一系列步骤都由动态链接器自动完成，对于程序本身是透明的

. 动态库的装载是通过一些列的动态链接器提供的API按成的
/*
#include <dlfcn.h>
/lib.lindl.so.2
*/
```

0x1: dlopen()

打开一个动态链接库，将其加载到进程的地址空间，并返回动态链接库的句柄，完成初始化过程

```
void * dlopen( const char * pathname, int mode);
. pathname: 被加载动态库的路径
值得注意的是:
如果pathname传入是0，则dlopen返回的是全局符号表的句柄，也就是说我们可以在运行时找到全局符号表里面的任何一个符号，并且可以执行它们，这类似于高级语言中的反射(relection)特性
全局符号表包括了程序的可执行文件本身、被动态链接器加载到进程中的所有共享模块、运行时通过dlopen打开并且使用了RTLD_GLOBAL方式的模块中的符号

. mode:
mode是打开方式，其值有多个，不同操作系统上实现的功能有所不同，在linux下，按功能可分为三类:
    2.1 解析方式
        ) RTLD_LAZY: 在dlopen返回前，对于动态库中的未定义的符号不执行解析(只对函数引用有效，对于变量引用总是立即解析)
        ) RTLD_NOW: 需要在dlopen返回前，解析出所有未定义符号，如果解析不出来，在dlopen会返回NULL，错误为: : undefined symbol: xxxx.....
    2.2 作用范围: 可与解析方式通过"|"组合使用
        ) RTLD_GLOBAL: 动态库中定义的符号可被其后打开的其它库解析
        ) RTLD_LOCAL: 与RTLD_GLOBAL作用相反，动态库中定义的符号不能被其后打开的其它库重定位。如果没有指明是RTLD_GLOBAL还是RTLD_LOCAL，则缺省为RTLD_LOCAL
    2.3 作用方式
        ) RTLD_NODELETE: 在dlclose()期间不卸载库，并且在以后使用dlopen()重新加载库时不初始化库中的静态变量。这个flag不是POSIX-2001标准
        ) RTLD_NOLOAD: 不加载库。可用于测试库是否已加载(dlopen()返回NULL说明未加载，否则说明已加载)，也可用于改变已加载库的flag，如: 先前加载库的flag为RTLD_LOCAL，用dlopen(RTLD_NOLOAD|RTLD_GLOBAL)后flag将变成RTLD_GLOBAL。这个flag不是POSIX-2001标准
        ) RTLD_DEEPBIND: 在搜索全局符号前先搜索库内的符号，避免同名符号的冲突。这个flag不是POSIX-2001标准
```

dlopen会尝试以一定的顺序去查找动态库文件

```
. 查找环境变量LD_LIBRARY_PATH指定的一些列目录
. 查找由/etc/ld.so.cache指定的共享库路径
. /lib/、/usr/lib
```

0x2: dlsym()

根据动态链接库操作句柄与符号，返回符号对应的地址

```
#include <dlfcn.h>
void * dlsym(void *handle, constchar *symbol)
```

符号优先级

```
. 全局符号的优先级
我们知道，在全局符号的引入中，不管是动态链接器在程序启动时引入依赖的动态共享库、LD_LIBRARY_PATH指定引入的动态共享库、还是程序在运行时中调用dlopen引入动态共享库。当发生多个同名符号冲突时，都遵循"先来后到"的原则，即先装入的符号优先，这种优先级方式称为"装载序列(Load Ordering)"

. dlsym()动态符号的优先级
dlsym()对符号的查找优先级分为两种
        ) 装载序列:
        在全局符号表中进行遵循装载序列的搜索: 在dlopen的时候pathname传入0

        ) 依赖序列:
        从目标共享模块开始进行遵循依赖序列的搜索: 在dlopen的时候pathname传入目标模块路径，在dlsym的时候传入dlopen返回的句柄指针
        所谓依赖序列就是以被dlopen打开的那个共享对象为根节点，对它所有依赖的共享对象进行广度优先遍历，找到了就返回，如果没找到就继续找，直到找到符号为止
```

我们可以使用下面的代码来帮助我们理解这个原理

```
/*
hook.c
*/
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

int strcmp(const char *s1, const char *s2)
{
    //这个hook函数只是简单地打印一句话
    printf("oops!!! hack function invoked\n");
}
gcc -fPIC -shared -o hook.so hook.c -ldl
cp hook.so /lib64/

/*
main.c
*/
#include <stdio.h>
#include <dlfcn.h>

```

```c
int main(int argc, char **argv)
{
    void *handle1, *handle2;
    int (*cosine1)(const char *, const char *);
    int (*cosine2)(const char *, const char *);
    char *error;

    //返回hook.so的模块句柄
    handle1 = dlopen ("hook.so", RTLD_LAZY);
    //返回全局符号表
    handle2 = dlopen (, RTLD_LAZY);

    if (!handle1 | !handle2)
    {
        fprintf (stderr, "%s\n", dlerror());
        ;
    }

    //从刚才打开的句柄中搜索符号
    cosine1 = dlsym(handle1, "strcmp");

    //从全局符号表中搜索符号
    cosine2 = dlsym(handle2, "strcmp");

    if ((error = dlerror()) != NULL)
    {
        fprintf (stderr, "%s\n", error);
        ;
    }

    //采用依赖序列(dependency ordering)优先级进行符号搜索，优先执行动态引入的hook.so的函数
    printf ("%f\n", (*cosine1)("aaa", "bbb"));

    //采用装载序列(load ordering)优先级啊进行符号搜索，动态引入的hook.so被忽略
    printf ("%f\n", (*cosine2)("aaa", "bbb"));
    dlclose(handle1);
    dlclose(handle2);
    ;
}
gcc -rdynamic -o main main.c -ldl
```