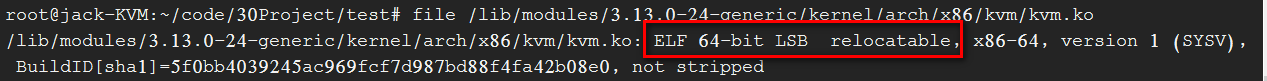
# [Linux下的内核模块机制](https://www.cnblogs.com/ck1020/p/7056179.html)

 Linux的内核模块机制允许开发者动态的向内核添加功能，我们常见的文件系统、驱动程序等都可以通过模块的方式添加到内核而无需对内核重新编译，这在很大程度上减少了操作的复杂度。模块机制使内核预编译时不必包含很多无关功能，把内核做到最精简，后期可以根据需要进行添加。而针对驱动程序，因为涉及到具体的硬件，很难使通用的，且其中可能包含了各个厂商的私密接口，厂商几乎不会允许开发者把源代码公开，这就和linux的许可相悖，模块机制很好的解决了这个冲突，允许驱动程序后期进行添加而不合并到内核。OK，下面结合源代码讨论下模块机制的实现。

类似于普通的可执行文件，模块经过编译后得到.ko文件，其本身也是可重定位目标文件，类似于gcc -c 得到的.o目标文件。对于可重定位的概念，请参考PE文件格式（虽然是windows下的，但是原理类似）。





 既然是重定位文件，在把模块加载到内核的时候就需要进行重定位，回想下用户可执行文件的重定位，一般如果一个程序的可执行文件总能加载到自己的理想位置，所以对于用户可执行文件，一般不怎么需要重定位，而对于动态库文件就不同了，库文件格式是一致的，但是可能需要加载多个库文件，那么有些库文件必然无法加载到自己的理想位置，就需要进行重定位。而内核模块由于和内核共享同一个内核地址空间，更不能保证自己的理想地址不被占用，所以一般情况内核模块也需要进行重定位。在加载到内核时，还有一个重要的工作即使解决模块之间的依赖，模块A中引用了其他模块的函数，那么在加载到内核之前其实模块A并不知道所引用的函数地址，因此只能做一个标记，在加载到内核的时候在根据符号表解决引用问题！这些都是在加载内核的核心系统调用sys\_init\_module完成。

 内核中的数据结构

每一个内核模块在内核中都对应一个数据结构module,所有的模块通过一个链表维护。所以有些恶意模块企图通过从链表摘除结构来达到隐藏模块的目的。部分成员列举如下：



struct module

{

enum module\_state state;

/\* Member of list of modules \*/

struct list\_head list;//所有的模块构成双链表，包头为全局变量modules

/\* Unique handle for this module \*/

char name[MODULE\_NAME\_LEN];//模块名字，唯一，一般存储去掉.ko的部分

/\* Sysfs stuff. \*/

struct module\_kobject mkobj;

struct module\_attribute \*modinfo\_attrs;

const char \*version;

const char \*srcversion;

struct kobject \*holders\_dir;

/\* Exported symbols \*//\*\*/

const struct kernel\_symbol \*syms;//导出符号信息，指向一个kernel\_symbol的数组，有num\_syms个表项。

const unsigned long \*crcs;//同样有num\_syms个表项，不过存储的是符号的校验和

unsigned int num\_syms;

/\* Kernel parameters. \*/

struct kernel\_param \*kp;

unsigned int num\_kp;

/\* GPL-only exported symbols. \*/

unsigned int num\_gpl\_syms;//具体意义同上面符号，但是这里只适用于GPL兼容的模块

const struct kernel\_symbol \*gpl\_syms;

const unsigned long \*gpl\_crcs;

#ifdef CONFIG\_UNUSED\_SYMBOLS

/\* unused exported symbols. \*/

const struct kernel\_symbol \*unused\_syms;

const unsigned long \*unused\_crcs;

unsigned int num\_unused\_syms;

/\* GPL-only, unused exported symbols. \*/

unsigned int num\_unused\_gpl\_syms;

const struct kernel\_symbol \*unused\_gpl\_syms;

const unsigned long \*unused\_gpl\_crcs;

#endif

#ifdef CONFIG\_MODULE\_SIG

/\* Signature was verified. \*/

bool sig\_ok;

#endif

/\* symbols that will be GPL-only in the near future. \*/

const struct kernel\_symbol \*gpl\_future\_syms;

const unsigned long \*gpl\_future\_crcs;

unsigned int num\_gpl\_future\_syms;

/\* Exception table \*/

unsigned int num\_exentries;

struct exception\_table\_entry \*extable;

/\* Startup function. \*/

int (\*init)(void);//模块初始化函数指针

/\* If this is non-NULL, vfree after init() returns \*/

void \*module\_init;/如果该函数不为空，则init结束后就可以调用进行适当释放

/\* Here is the actual code + data, vfree'd on unload. \*/

void \*module\_core;//核心数据和代码部分，在卸载的时候会调用

/\* Here are the sizes of the init and core sections \*/

unsigned int init\_size, core\_size;//对应于上面的init和core函数，决定各自占用的大小

/\* The size of the executable code in each section. \*/

unsigned int init\_text\_size, core\_text\_size;

/\* Size of RO sections of the module (text+rodata) \*/

unsigned int init\_ro\_size, core\_ro\_size;

。。。。。。

#ifdef CONFIG\_MODULE\_UNLOAD

　　　　/\*模块间的依赖关系记录\*/

/\* What modules depend on me? \*/

struct list\_head source\_list;

/\* What modules do I depend on? \*/

struct list\_head target\_list;

/\* Who is waiting for us to be unloaded \*/

struct task\_struct \*waiter;//等待队列，记录那些进程等待模块被卸载

/\* Destruction function. \*/

void (\*exit)(void);//卸载退出函数，模块中定义的exit函数

。。。。。。

};



依赖关系

模块间的依赖关系通过两个节点source\_list和target\_list记录，前者记录那些模块依赖于本模块，后者记录本模块依赖于那些模块。节点通过module\_use记录，module\_use如下

struct module\_use {

struct list\_head source\_list;

struct list\_head target\_list;

struct module \*source, \*target;

};

每个module\_use记录一个映射关系，注意这里把source和target放在一个一个结构里，因为一个关系需要在源模块和目标模块都做记录。如果模块A依赖于模块B，则生成一个module\_use结构，其中source\_list字段链入模块B的module结构的source\_list链表，而source指针指向模块A的module结构。而target\_list加入到模块A中的target\_list链表，target指针指向模块B的模块结构，参考下面代码。



static int add\_module\_usage(struct module \*a, struct module \*b)

{

struct module\_use \*use;

pr\_debug("Allocating new usage for %s.\n", a->name);

use = kmalloc(sizeof(\*use), GFP\_ATOMIC);

if (!use) {

printk(KERN\_WARNING "%s: out of memory loading\n", a->name);

return -ENOMEM;

}

use->source = a;

use->target = b;

list\_add(&use->source\_list, &b->source\_list);

list\_add(&use->target\_list, &a->target\_list);

return 0;

}



符号信息

内核模块几乎不会作为完全独立的存在，均需要引用其他模块的函数，而这一机制就是由符号机制保证的。参考前面的module数据结构，在

const struct kernel\_symbol \*syms;//导出符号信息，指向一个kernel\_symbol的数组，有num\_syms个表项。

const unsigned long \*crcs;//同样有num\_syms个表项，不过存储的是符号的校验和

unsigned int num\_syms;

syms指针指向一个符号数组，也可以称之为符号表，不过是局部的符号表。看下kernel\_symbol结构

struct kernel\_symbol

{

unsigned long value;

const char \*name;

};

结构很简单，value记录符号地址，而name自然就是符号名字了。原理很简单，借助于find\_symbol函数看下内核如果解决位引用的符号



const struct kernel\_symbol \*find\_symbol(const char \*name,

struct module \*\*owner,

const unsigned long \*\*crc,

bool gplok,

bool warn)

{

struct find\_symbol\_arg fsa;

fsa.name = name;

fsa.gplok = gplok;

fsa.warn = warn;

if (each\_symbol\_section(find\_symbol\_in\_section, &fsa)) {

if (owner)

\*owner = fsa.owner;

if (crc)

\*crc = fsa.crc;

return fsa.sym;

}

pr\_debug("Failed to find symbol %s\n", name);

return NULL;

}



首先把参数信息封装成一个find\_symbol\_arg结构，然后调用了each\_symbol\_section，并传入了在section中查找symbol的函数find\_symbol\_in\_section



bool each\_symbol\_section(bool (\*fn)(const struct symsearch \*arr,

struct module \*owner,

void \*data),

void \*data)

{

struct module \*mod;

static const struct symsearch arr[] = {

{ \_\_start\_\_\_ksymtab, \_\_stop\_\_\_ksymtab, \_\_start\_\_\_kcrctab,

NOT\_GPL\_ONLY, false },

{ \_\_start\_\_\_ksymtab\_gpl, \_\_stop\_\_\_ksymtab\_gpl,

\_\_start\_\_\_kcrctab\_gpl,

GPL\_ONLY, false },

{ \_\_start\_\_\_ksymtab\_gpl\_future, \_\_stop\_\_\_ksymtab\_gpl\_future,

\_\_start\_\_\_kcrctab\_gpl\_future,

WILL\_BE\_GPL\_ONLY, false },

#ifdef CONFIG\_UNUSED\_SYMBOLS

{ \_\_start\_\_\_ksymtab\_unused, \_\_stop\_\_\_ksymtab\_unused,

\_\_start\_\_\_kcrctab\_unused,

NOT\_GPL\_ONLY, true },

{ \_\_start\_\_\_ksymtab\_unused\_gpl, \_\_stop\_\_\_ksymtab\_unused\_gpl,

\_\_start\_\_\_kcrctab\_unused\_gpl,

GPL\_ONLY, true },

#endif

};

if (each\_symbol\_in\_section(arr, ARRAY\_SIZE(arr), NULL, fn, data))

return true;

list\_for\_each\_entry\_rcu(mod, &modules, list) {

struct symsearch arr[] = {

{ mod->syms, mod->syms + mod->num\_syms, mod->crcs,

NOT\_GPL\_ONLY, false },

{ mod->gpl\_syms, mod->gpl\_syms + mod->num\_gpl\_syms,

mod->gpl\_crcs,

GPL\_ONLY, false },

{ mod->gpl\_future\_syms,

mod->gpl\_future\_syms + mod->num\_gpl\_future\_syms,

mod->gpl\_future\_crcs,

WILL\_BE\_GPL\_ONLY, false },

#ifdef CONFIG\_UNUSED\_SYMBOLS

{ mod->unused\_syms,

mod->unused\_syms + mod->num\_unused\_syms,

mod->unused\_crcs,

NOT\_GPL\_ONLY, true },

{ mod->unused\_gpl\_syms,

mod->unused\_gpl\_syms + mod->num\_unused\_gpl\_syms,

mod->unused\_gpl\_crcs,

GPL\_ONLY, true },

#endif

};

if (mod->state == MODULE\_STATE\_UNFORMED)

continue;

if (each\_symbol\_in\_section(arr, ARRAY\_SIZE(arr), mod, fn, data))

return true;

}

return false;

}



首先考虑的自然是内核自身的符号，根据优先顺序，定义了一个数组，内核中的导出符号记录在全局的结构中，顺序分别是\_\_start\_\_\_ksymtab、\_\_start\_\_\_ksymtab\_gpl、\_\_start\_\_\_ksymtab\_gpl\_future、\_\_start\_\_\_ksymtab\_unused、\_\_start\_\_\_ksymtab\_unused\_gpl。然后调用each\_symbol\_in\_section进行遍历数组，针对每一个项，调用find\_symbol\_in\_section进行查找。如果内核中的符号没有包含指定符号，则需要查找其他加载模块的符号表，这就是局部符号表，方法类似，不过是表指针记录在module结构中而不是全局的。不在赘述。看下find\_symbol\_in\_section



static bool find\_symbol\_in\_section(const struct symsearch \*syms,

struct module \*owner,

void \*data)

{

struct find\_symbol\_arg \*fsa = data;

struct kernel\_symbol \*sym;

sym = bsearch(fsa->name, syms->start, syms->stop - syms->start,

sizeof(struct kernel\_symbol), cmp\_name);

if (sym != NULL && check\_symbol(syms, owner, sym - syms->start, data))

return true;

return false;

}



该函数是根据是个符号表的起始和结束区间对符号进行查找，具体查找工作有bsearch完成，通过二分查找key,即符号名字。算法挺简单，我们也看下



void \*bsearch(const void \*key, const void \*base, size\_t num, size\_t size,

int (\*cmp)(const void \*key, const void \*elt))

{

size\_t start = 0, end = num;

int result;

while (start < end) {

size\_t mid = start + (end - start) / 2;

result = cmp(key, base + mid \* size);

if (result < 0)

end = mid;

else if (result > 0)

start = mid + 1;

else

return (void \*)base + mid \* size;

}

return NULL;

}



 找到一个结果就调用cmp进行比较，cmp为开始传递进来的比较函数，本质还是调用strcmp函数。有这里可以看出，符号表种符号是有顺序的，即通过首字母进行排列，首字母相同则按照第二个字母，以此类推。这样在找到symbol后会对其进行校验，如果没有找到就直接返回false了……

使用未导出的函数

1、定义函数指针

2、声明函数

3、查找符号表

以内核中的lookup\_swap\_cache函数为例，函数在内核中未导出，不能直接使用。通过查找符号表，把函数地址强制转化成函数指针，可以为我们所用。

1、定义函数指针

函数原始定义如下：struct page \*lookup\_swap\_cache(swp\_entry\_t swp)

typedef struct page\* (\*LOOKUPSWAPCACHE)(swp\_entry\_t);

2、声明函数

LOOKUPSWAPCACHE lookup\_swap\_cache\_chen;

3、赋值

lookup\_swap\_cache\_chen=(LOOKUPSWAPCACHE)kallsyms\_lookup\_name("lookup\_swap\_ cache");

这样就可以在自己的模块中使用了lookup\_swap\_cache\_chen函数了。需要包含头文件#include <linux/kallsyms.h>

用户空间信息

一下信息摘自：http://www.blogbus.com/wanderer-zjhit-logs/172382425.html

 内核符号表（kernel symbol table）变量名或者函数名组成，每一项是符号和地址的序对，就像域名和ip地址，格式如下：

[root@rx6600 boot]# head System.map  
000000000479c4a0 A phys\_start  
a000000000000600 A \_\_start\_gate\_mckinley\_e9\_patchlist  
a000000000000604 A \_\_end\_gate\_mckinley\_e9\_patchlist  
a000000000000604 A \_\_end\_gate\_vtop\_patchlist  
a000000000000604 A \_\_start\_gate\_fsyscall\_patchlist  
a000000000000604 A \_\_start\_gate\_vtop\_patchlist  
对于系统的oop消息、或者通过gdb的调试消息，都需要根据该对照表，将内核熟悉的函数地址转化为用户熟悉的函数名称，便于用户进行故障定位、运行监控。  
 内核符号表存储位置  
 System.map  
磁盘中真实存在的文件，存储内核中静态编译的函数和变量地址，每个新编译内核对应一个System.map文件，当klogd输出内核消息时，会通过/boot/System.map来将函数、变量地址转换为名称，方便用户理解。该文件对应不同的编译内核有对应的实现文件。  
/proc/kallsyms  
内核启动时候创建,共oops时定位错误，文件大小总为0，包含当前内核导出的、可供使用的变量或者函数  
相似点：都是内核函数、变量的符号表，结构一致；对于可导出的内核变量、函数，其运行时在物理内存中的位置是一样的。  
区别：  
两者侧重点不同，System.map文件面向内核，对于内核中的没有导出的变量或者函数名，比如kthread\_create\_list链表头指针，也有其相应的内核地址，该文件一般是只读的、固定大小的，没有动态添加模块中的变量、函数名；而System.map在内核启动过程中创建，并实时更新，反映的是系统的当前最新情况，其内部也包含内核或者是已加载模块导出的函数、变量名称。所以和System.map文件有差别，且文件动态变化，大小不固定。

* 注：/proc/kmsg文件保存了内核从最开始启动到正常运行时的所有内核输出消息，是内核在运行过程中通过printk输出的。
* 如果klogd启动，klogd读取/proc/kmsg文件的内容，然后通过syslogd程序，写到/var/log/messages文件中，当然，syslogd可以通过syslogd.conf文件进行配置。利用dmesg，其实也是读取/proc/kmsg文件内容，然后显示到终端。
* dmesg和klogd都是利用了System.map文件将内核地址转化为对应的函数名称，方便用户调试。
* 在内核运行出现问题时，一般由于引用了一个无效指针造成的oops错误，如果在应用层，一般应用程序不可能从段错误（即引用无效地址）中恢复，但是由于内核稳定性比较高，一般只是会将该内核模块杀死，并使系统维持在一个稳定状态；如果出现更严重情况，即内核出现panic，就会宕机重启。