(3 条消息) 内核模块加载过程分析_晓镁的博客 - CSDN 博客

章地址如下：

ELF 文件格式解析

https://blog.csdn.net/zj82448191/article/details/108441447

在用户空间，用 insmod 这样的命令来向内核空间安装一个内核模块，本章将详细讨论模块加载时的内核行为，当我们加载一个模块时，insmod 会首先利用文件系统的接口将其数据读取到用户空间的一段 内存 中，然后通过系统调用 sys_init_module, 让内核去处理加载的整个过程。

一、sys_init_module 函数分析

我们把 sys_init_module 函数分为两个部分，第一部分是调用 load_module(), 完成模块加载最核心的任务，第二部分是模块加载到系统之后的处理。在分析之前我们先介绍两个结构体，这两个结构体在后面的介绍中都用到了。

linux-3.10.70\kernel\module.c

```
struct load_info {
        Elf_Ehdr *hdr;          // ELF文件头
        unsigned long len;      // 文件长度，似乎除了校验的
        Elf_Shdr *sechdrs;      // 节区头部表
        char *secstrings, *strtab;     // section 名称表
        unsigned long symoffs, stroffs;     // 符号表，
        struct _ddebug *debug;
```

```
        unsigned int num_debug;
        bool sig_ok;
#ifdef CONFIG_KALLSYMS
        unsigned long mod_kallsyms_init_off;
#endif
    /*sym 为符号表在secton headers 中的index
     *str 为字符串表在section header 中的index
     */
        struct {
                unsigned int sym, str, mod, vers, infc
        } index;
};
```

linux-3.10.70\include\linux\module.h

```
struct module {
    // 用来记录模块加载过程中不同阶段的状态
        enum module_state state;

        /* Member of list of modules */
        // 可以看到内核使用链表来管理module
        struct list_head list;

        /* Unique handle for this module */
        /* 模块名称 */
        char name[MODULE_NAME_LEN];

        /* Sysfs stuff. */
        struct module_kobject mkobj;
        struct module_attribute *modinfo_attrs;
        const char *version;
        const char *srcversion;
        struct kobject *holders_dir;

        /* Exported symbols */
        // 模块导出符号的起始地址
        const struct kernel_symbol *syms;
        // 模块导出符号的校验码起始地址
        const unsigned long *crcs;
        unsigned int num_syms;

        /* Kernel parameters. */
        // 内核模块参数所在的起始地址
        struct kernel_param *kp;
```

```
        unsigned int num_kp;

        /* GPL-only exported symbols. */
        unsigned int num_gpl_syms;
        const struct kernel_symbol *gpl_syms;
        const unsigned long *gpl_crcs;

        /* symbols that will be GPL-only in the near
        const struct kernel_symbol *gpl_future_syms;
        const unsigned long *gpl_future_crcs;
        unsigned int num_gpl_future_syms;

        /* Startup function. */
        // 这就是我们用module_init(xxx)来声明的入口函数
        int (*init)(void);

        /* If this is non-NULL, vfree after init() ret
        void *module_init;

        /* Here is the actual code + data, vfree'd on
        void *module_core;

        /* Here are the sizes of the init and core se
        unsigned int init_size, core_size;

        /* The size of the executable code in each se
        unsigned int init_text_size, core_text_size;

        /* Size of RO sections of the module (text+ro
        unsigned int init_ro_size, core_ro_size;

        /* Arch-specific module values */
        struct mod_arch_specific arch;

        unsigned int taints;    /* same bits as kernel

#ifdef CONFIG_KALLSYMS
        /*
         * We keep the symbol and string tables for k
         * The core_* fields below are temporary, loa
         * could really be discarded after module ini
         */
        Elf_Sym *symtab, *core_symtab;
        unsigned int num_symtab, core_num_syms;
        char *strtab, *core_strtab;
```

```
        /* Section attributes */
        struct module_sect_attrs *sect_attrs;

        /* Notes attributes */
        struct module_notes_attrs *notes_attrs;
#endif

        /* The command line arguments (may be mangled)
           keeping pointers to this stuff */
        char *args;

#ifdef CONFIG_MODULE_UNLOAD
        /* 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;

        /* Destruction function. */
        void (*exit)(void);

        struct module_ref __percpu *refptr;
#endif
}
```

1.1 第一部分

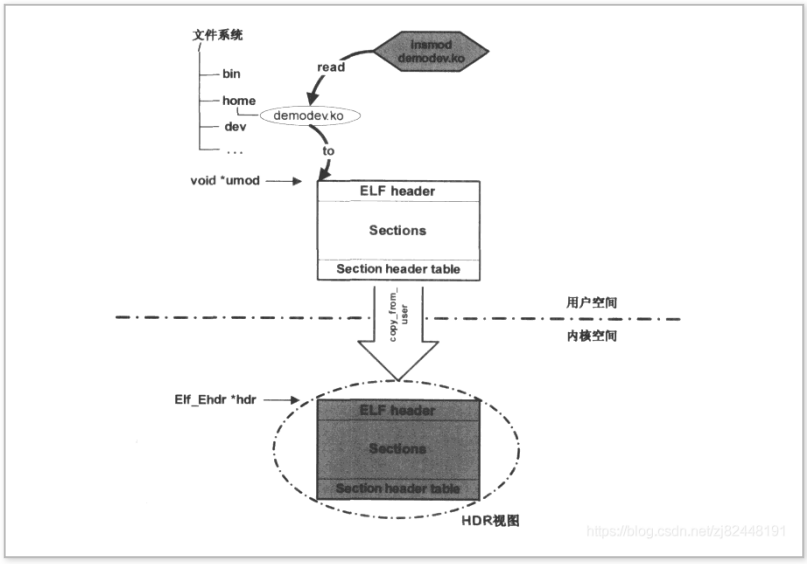
第一部分的内容都是由 load_module 完成的，所以我们这里详细的介绍该函数。

1.1.1 用户空间到内核空间

我们知道 insmod 会首先利用文件系统的接口将其数据读取到用户空间的一段内存中，然后通过系统调用 sys_init_module, 让内核去处理加载的整个过程。调用 sys_init_module 的时候，会把. ko 文件的首地址及大小通过参数传入，然后 load_module 在内核空间中使

用 vmalloc 开辟大小一样的空间，将文件数据都复制到内核空间。从而在内核空间构建出一个 demo.ko 文件的 ELF 静态的内存视图。接下来的操作都将以此视图为基础，我们成这个视图为 HDR 视图。

这部分是我截图出来的别的地方的图，供大家借鉴，从图中我们可以清楚的看到上述的过程。



1.1.2 HDR 视图的第一次修改

我们把文件的数据拷贝到内核空间之后，需要对一些数据进行修改，视图的第一次修改只修改了节点头部表中 sh_addr 该变量，我们知道这个变量表示节区在进程内存中的起始地址。所以在拷贝到内核之后需要重新修改，遍历一次节区头部表，将每个 entry 的 sh_addr 改为

entry【i】.sh_addr = （size_t）hdr+entry【i】.sh_offset

因为 hdr 为文件在内存中的其实位置，sh_offset 为节区的偏移位置。

1.1.3 mod 变量初始化

load_module 函数中定义有一个 struct module 类型的 mod 变量，该变量的初始化是通过模块文件. ko 中的一个节区来实现的，该节区为：.gun.linkonce…thie_module。

在 ELF 文件中为什么会有这个段，其实是模块的编译工具链完成的，如果我们仔细看模块编译后的文件，会发现一个. mod.c 的文件，查看该文件内容会发现如下定义：

```
    struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) =
 .name = KBUILD_MODNAME,
 .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
 .exit = cleanup_module,
#endif
 .arch = MODULE_ARCH_INIT,
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

这里我们只要找到这个段在内存中的其实地址，然后就能赋值给 mod 变量了，内核中 find_sec 函数可以根据某一节区查找到该节区在节区头部表中的标号，进而找到该节区在内存中的开始地址。

mod =（void *）sechdrs[modindex].sh_addr

于是到这里位置，在第一次修改完视图之后，mod 指向了实际的该节区的初始地址，接下来会进行视图的第二次改写，会再次更新 mod 的地址。

1.1.4HDR 视图的第二次改写

这里为什么会有第二次视图的改写，是因为很多节区是不需要加载到文件的内存映像中，所以第二次改写的时候，内核会把所有的节区遍历一遍，若是节区头部表中节区 sh_flgs 变量的值 SHF_ALLOCK，则会根据节区名，将节区分为两部分，CORE 和 INIT。以. init 开头的节区分到 INIT 部分，其余分到 CORE 部分。

在对所有节区进行分类的过程中，会记录下当前节区在该类中的偏移量，保存在节区头部表中 sh_entsize 部分。

enrty【i】.sh_entsize=mod->core_size; 或者 enrty【i】.sh_entsize=mod->init_size;

同时记录该节区分配之后，两个类的空间大小：

mod->core_size += entry[i].sh_size; 或者 mod->init_size +=entry[i].sh_size;

这里分好类之后搬移之前会再做一个工作，是对符号字符表的处理，这里我们介绍一个宏：

CONFIG_KALLSYMS, 这个宏是一个决定内核映像中是否保留所有符号的配置选项，在内核 Kconfig 中，如果没有打开这个宏，这里我们上面都不做，直接进行 HDR 视图的第二次改写，但是如果定义了，内核模块

的符号都会放到 ELF 文件中的一个节区中，但是由于符号表的节区没有 SHF_ALLCOK 标志，所以需要将这个节区也手动分到 CORE 类中。

所有的节区都分完类之后，内核会调用 vmalloc 分配两个类对应的内存空间，基地址分别记录在：mod->module_core 跟 mo->module_init，然后将视图搬移到新申请的内存上，显然，这里搬移成功之后需要再次修改节区头部表中每个节区的 sh_addr 的值，使其指向新的地址。

这里还需要更新一下 mod 变量，使其指向新的地址。

mod = （void*） entry[modindex].sh_addr;

为什么内核会进行第二次分类，这里再次说明一下，因为在模块加载过程结束时，系统会释放掉 HDR 所在的内存区域，在模块是初花完成之后，INIT 类的区域也会被释放掉，由此可见，当一个模块被成功记载切初始化完成后，最终留下的仅仅是 CORE 类的内容，这些数据才是在系统整个运行期间存活的数据。

下面分享第二次搬移过程图：

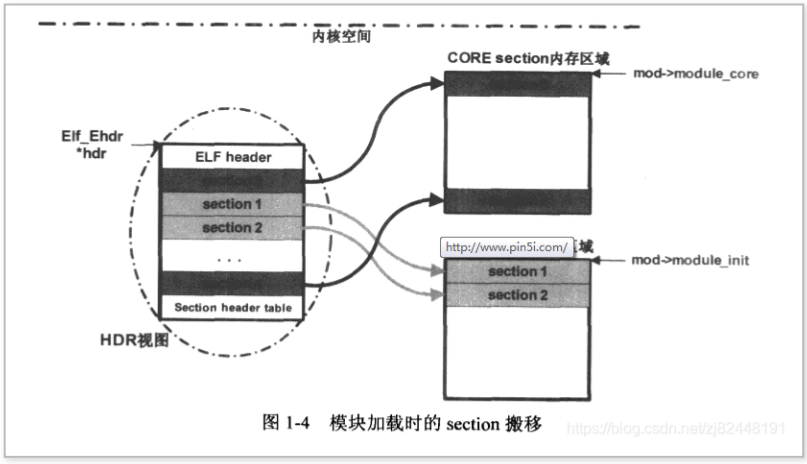


图 1-4 模块加载时的 section 搬移

到此为止，视图的搬移就结束了，但是这个时候工作还没有完成，因为我们还没有解决引用的符号问题。

1.1.5 符号问题

我们知道内核源码中存在大量的 EXPORT_SYMBOL 这样的宏，我们一般只是知道它是想外面导出一个符号，但是不知道其中的原理，这里我们研究一下。

如果没有独立存在的内核模块，EXPORT_SYMBOL 就是去了存在的意义，因为对于静态编译了解的内核映像来说，所有的符号引用都在静态链接的时候完成了，但是内核模块不可避免的必须使用内核提供的基础设施，比如 Printk 函数，作为独立编译的内核模块，要解决这种问题，就必须找到引用的符号在内存的实际地址。

从全局来看，EXPORT_SYMBOL 分为宏定义部分，链接脚本链接器部分和使用导出符号部分，下面来说这三部分：

第一部分

```
<include/linux/module.h>
#define __EXPORT_SYMBOL(sym, sec)                       \
        extern typeof(sym) sym;                         \
        __CRC_SYMBOL(sym, sec)                          \
        static const char __kstrtab_##sym[]             \
        __attribute__((section("__ksymtab_strings"), aligned(1))) \
        = MODULE_SYMBOL_PREFIX #sym;                                \
        static const struct kernel_symbol __ksymtab_##sym   \
        __used                                          \
        __attribute__((section("__ksymtab" sec), unused))   \
        = { (unsigned long)&sym, __kstrtab_##sym }

#define EXPORT_SYMBOL(sym)                              \
        __EXPORT_SYMBOL(sym, "")

#define EXPORT_SYMBOL_GPL(sym)                              \
        __EXPORT_SYMBOL(sym, "_gpl")

#define EXPORT_SYMBOL_GPL_FUTURE(sym)                       \
        __EXPORT_SYMBOL(sym, "_gpl_future")
```


从上面的宏定义我们可以看到，EXPORT_SYMBOL 的宏定义实际上就是定义了两个变量，这里我们使用

EXPORT_SYMBOL (my_exp_function) 作为一个例子。导出这个函数，相当于定义了两个变量如下：

```
    static  const  char * _kstrtab_my_exp_function = '

static  const struct kernel_symbol __kstrtab_my_exp_fu
{(unsigned long )&my_exp_function, _kstrtab_my_exp_fun

这里介绍一下kernel_symbol结构体
  struct kernel_symbol{
                        unsigned long value;
                        const char* name;
                }
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

通过以上定义，我们可以知道，导出符号其实就是将符号名称及其地址通过 kernel_symbol 这个结构体告诉外部。

从上面的宏定义中还可以看到第二个信息，就是指针变

量_kstrtab_my_exp_function 将会被放到_ksymtab_strings 节区中，__kstrtab_my_exp_function 变量会放到_ksymtab 节区中。

第二部分

这一部分是链接脚本及链接器，我们这里粘出虚拟机的连接脚本。

<arch/x86/kernel/vmlinux.lds>

```
__ksymtab : AT(ADDR(__ksymtab) - 0xC0000000)
{ __start___ksymtab = .; *(__ksymtab) __stop___ksymtab = .; }

__ksymtab_gpl : AT(ADDR(__ksymtab_gpl) - 0xC0000000)
{ __start___ksymtab_gpl = .; *(__ksymtab_gpl) __stop___ksymtab_gpl = .; }

__ksymtab_gpl_future : AT(ADDR(__ksymtab_gpl_future) - 0xC0000000)
{
__start___ksymtab_gpl_future = .;
*(__ksymtab_gpl_future) __stop___ksymtab_gpl_future = .;
}
```



```
__kcrctab : AT(ADDR(__kcrctab) - 0xC0000000)
{ __start___kcrctab = .; *(__kcrctab) __stop___kcrctab = .; }

__kcrctab_gpl : AT(ADDR(__kcrctab_gpl) - 0xC0000000)
{ __start___kcrctab_gpl = .; *(__kcrctab_gpl) __stop___kcrctab_gpl = .; }

__kcrctab_gpl_future : AT(ADDR(__kcrctab_gpl_future) - 0xC0000000)
{ __start___kcrctab_gpl_future = .; *(__kcrctab_gpl_future) __stop___kcrctab_gpl_future = .; }

__ksymtab_strings : AT(ADDR(__ksymtab_strings) - 0xC0000000)
{ *(__ksymtab_strings) }
```


从链接脚本中我们可以看到，定义了好多变量，我们这里只看 EXPORT_SYMBOL 类型的，即：

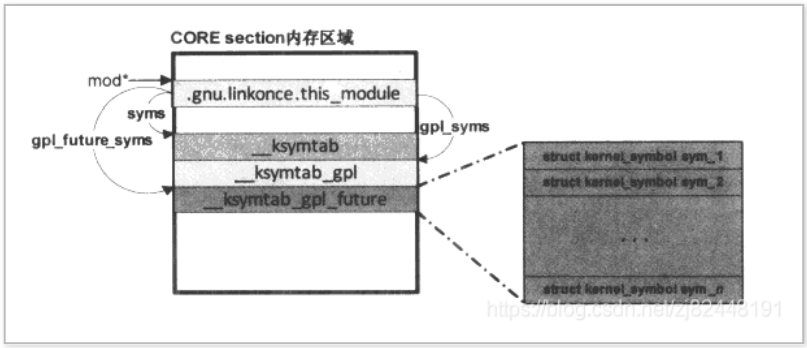
__start__ksymtab 跟__stop__ksymtab

__start__kcrctab 跟__stop__kcrctab

我们这里先不讲解第三部分，先研究一下模块的导出符号。

模块导出的符号经过编译工具链生成这些导出符号的节区，这些节区都带有 SHF_ALLOC 标志，所以在模块加载过程中会搬移到 core 类的空间中。如果没有导出符号，则不会产生写个节区。

显然，如果内核别的模块想使用一个内核模块导出的符号，就要对这些导出符号的地址有所了解，所以会记录下来这些符号的地址，这里内核通对 HDR 视图的查找，获取 t_ksymtab 节区在 core 空间的地址，然后将这个地址记录到 mod->syms 中，这样内核通过该变量就能找到模块导出符号的所有信息。具体图如下：



到这里位置，我们总结一下，我们知道了不管是内核还是内核模块，对于导出符号，都有相应的指针变量保存了这些符号节区在内存空间中的地址，所以当我们需要使用某个符号的时候，是不是就是在这些区域里寻找呢？我们接着分析

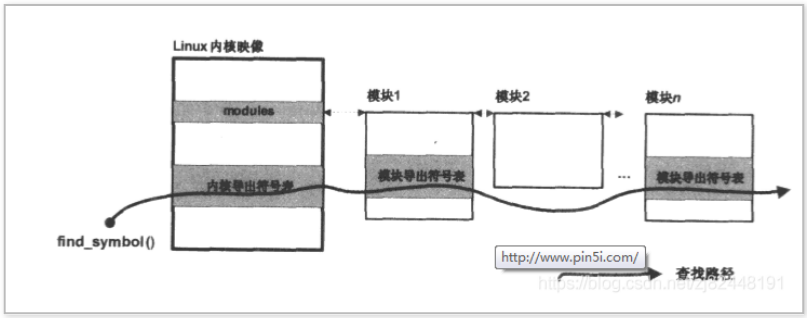
第三部分

这部分是解决那些 “” 未解决的引用 “” 问题，所谓的 “”“未解决的引用” 就是模块工具链编译模块生成. KO 文件的时候，对于模块中调用的一些函数，例如 printk 函数，链接工具无法在该模块的所有文件中找到该引用的指令码（因为这个函数是内核实现，指令码存在于内核的目标文件中），所以就会将这

个引用定义未解决，对他的处理一直延续到内核加载时，至于怎么解决，我们接着分析。

内核中有一个查找符号的函数 find_symbol，由于我这里不想过多的分析代码，所以我把这个函数的功能给大家说一下，该函数的功能就是内核与内核模块的符号节区中寻找 name 一样的符号，然后找到地址之后就可以引用了。

至于找到路径，是先从内核映像的导出符号表寻找 找不到则去模块链表中一个一个的模块导出符号表中寻找：



在内核模块的加载过程中，会有一个专门的函数解决那些 “” 未解决的引用 “” 问题，这个函数会给每个我们模块引用的符号找到它保存在节区中的对应的 struct kerne_synbol 变量，该变量之前我们介绍过，保存了符号名称与地址，但是找到这个地址就真的找到符号存在的地址了吗？？？ 我们思考一下，之前我们内核模块加载的时候，我们只是搬移了这些节区，并没有对这些节区中的地址做修改，这些符号的地址还是编译时候的地址，所以我们即使找到了，也不是采用这些地址。负责会出现大问题。linux 内核对这一问题的解决引入了重定位。

1.1.6 重定位

重定位的作用主要是解决静态链接时与动态加载时实际符号不一致的问题，上一节结束部分提到的模块导出的符号地址，就是需要重定位的一个典型的例子。重定位的部分先不讲解了 。有兴趣的可以自己研究

1.2 第二部分

load_module 函数完成了模块加载的所有困难的东西，接着返回到 sys_init_module，之后做的操作就很简单了。

1. 调用模块的是初始化函数。
这里调用 mod 函数中的 init 指向的函数，这之前已经分析过了，就不在叙述，需要知道的是，模块可以不提供初始化函数，如果提供了，则在初始化函数执行之后，需要修改模块的状态位 MODULE_STATE_LIVE;

2. 释放 INIT 类的 section 占用的空间
调用 vfree 来释放 mod->module_init,，这里再提一下，模块加载之后也会释放掉视图所在部分的内存，不过这部分是在 load_module 函数最后面执行的。

全文完