[(6条消息) linux内核模块的实现机制,Linux 内核模块工作原理及内核模块编译案例\_你好当代艺术的博客-CSDN博客](https://blog.csdn.net/weixin_36387104/article/details/116611877?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2~default~CTRLIST~Rate-1.pc_relevant_paycolumn_v3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~CTRLIST~Rate-1.pc_relevant_paycolumn_v3&utm_relevant_index=2)

https://blog.csdn.net/weixin\_36387104/article/details/116611877?spm=1001.2101.3001.6650.1&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc\_relevant\_paycolumn\_v3&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc\_relevant\_paycolumn\_v3&utm\_relevant\_index=2

一、Linux内核模块基本原理

Linux 内核模块(LKM)是一些在启动的操作系统内核需要时可以载入内核执行的代码块，不需要时由操作系统卸载。它们扩展了操作系统内核功能却不需要重新编译内核、启动系统。如果没有内核模块，就不得不反复编译生成操作系统的内核镜像来加入新功能，当附加的功能很多时，还会使内核变得臃肿。一个Linux 内核模块主要由以下几个部分组成：

(1) 模块加载函数(必须)：当通过insmod 或modprobe 命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块相关初始化工作。

(2) 模块卸载函数(必须)：当通过rmmod 命令卸载模块时，模块的卸载函数会自动被内核执行，完成与模块加载函数相反的功能。

(3) 模块许可证声明(必须)：模块许可证(LICENCE)声明描述内核模块的许可权限，如果不声明LICENCE，模块被加载时将收到内核被污染的警告。大多数情况下，内核模块应遵循GPL 兼容许可权。Linux2.6 内核模块最常见的是以MODULE\_LICENSE(“Dual BSD/GPL”)语句声明模块

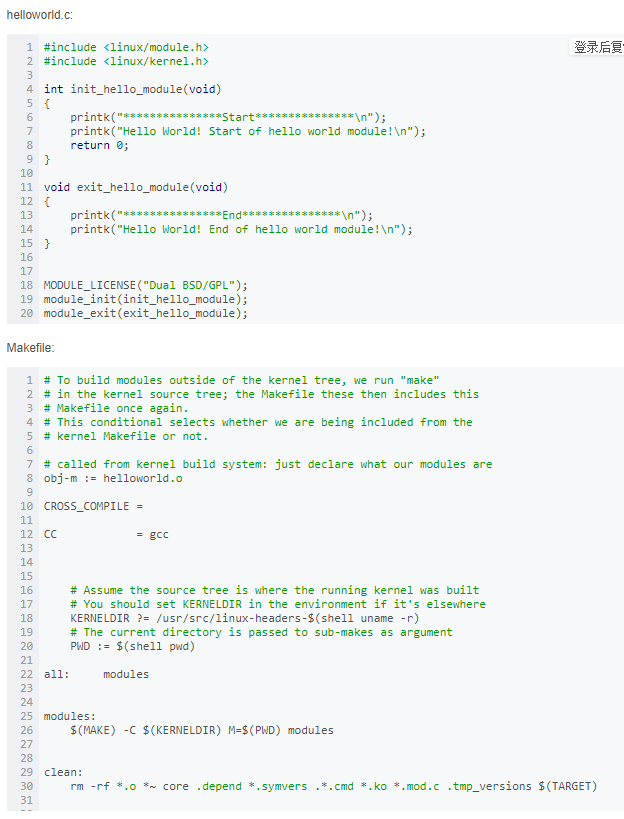
(4) 模块参数(可选)：模块参数是模块被加载的时候可以被传递给他的值，它本身对应模块内部的全局变量。

(5) 模块导出符号(可选)：内核模块可以导出符号(symbol，对应于函数或变量)，这样其他模块可以使用本模块中的变量或函数。

(6) 模块作者等信息声明(可选)。

一个内核模块至少包含两个函数，模块被加载时执行的初始化函数init\_module()和模块被卸载时执行的结束函数cleanup\_module()。在最新内核稳定版本2.6 中，两个函数可以起任意的名字，通过宏module\_init()和module\_exit()注册调用要编译内核模块，把代码嵌进内核空间，首先要获取内核源代码，且版本必需与当前正在运行的版本一致。

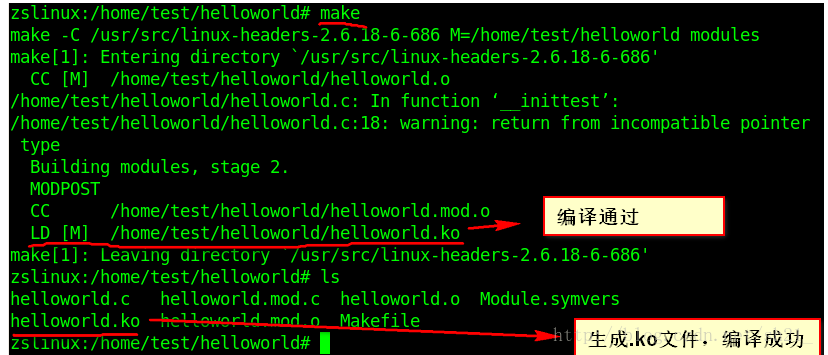
二、编写helloworld.c及其对应的Makefile。



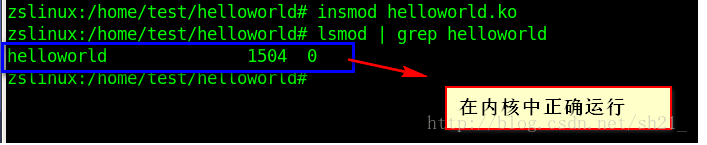
在Makefile中，在obj-m ：= helloworld.o这句中，.o的文件名要与编译的.c文件名一致。 KERNELDIR ？= /usr/src/linux-headers-$(shell uname -r)指示当前linux系统内核的源码位置。

三、编译：

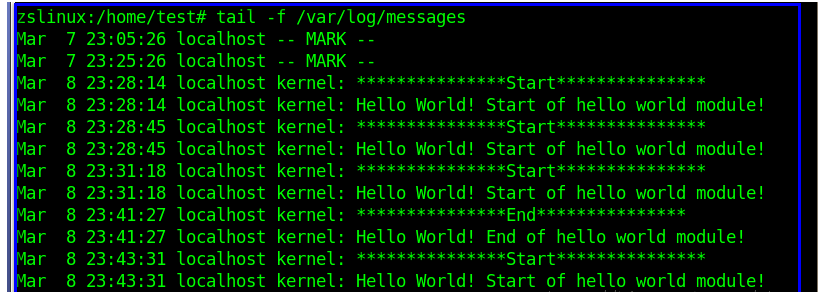
1.在Makefile及helloworld.c所在目录下，直接make，成功后查看当前目录下有无helloworld.ko文件产生，有则内核模块生成成功。

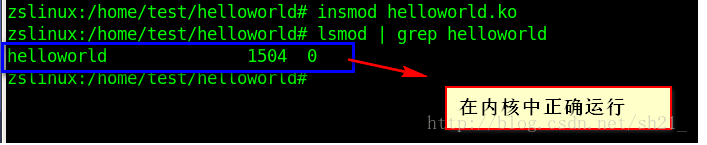


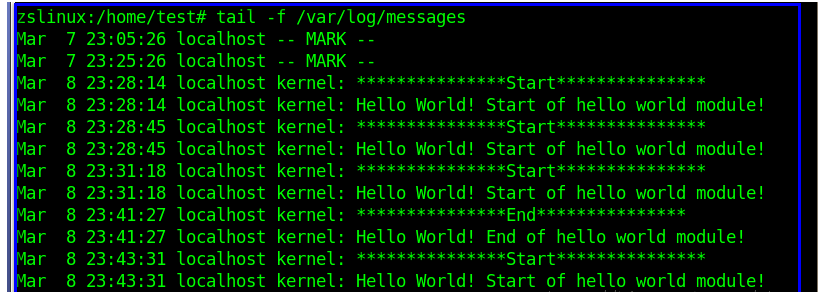
2.使用insmod命令，把此内核模块程序加载到内核中运行。结合lsmod及管道命令，查看内核模块程序在内核中是否正确运行。

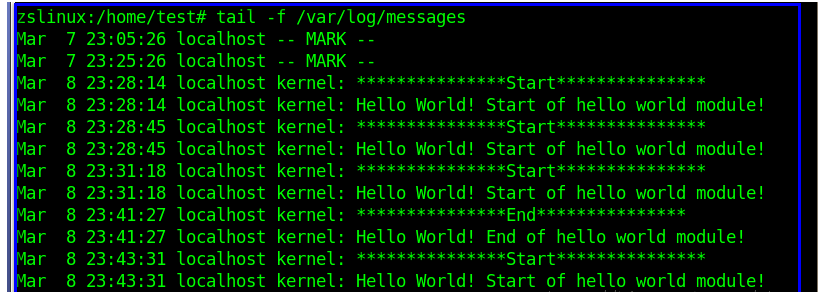


3.查看此内核模块程序打印的信息，另开一个终端，输入tail -n /var/log/messages. 使用rmmod命令把之前加载的内核模块卸载掉，然后再次执行第2步，即可看到此内核模块程序打印的信息。





1. 查看此内核模块程序打印的信息，另开一个终端，输入tail -n /var/log/messages. 使用rmmod命令把之前加载的内核模块卸载掉，然后再次执行第2步，即可看
2. 到此内核模块程序打印的信息。



[(6条消息) ELF文件格式解析\_乱世半仙的博客-CSDN博客\_elf文件格式详解](https://blog.csdn.net/zj82448191/article/details/108441447)

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

前言：由于.KO文件是ELF格式的文件，所以这里先学习一下ELF文件格式

一、ELF文件格式

1.1 ELF文件介绍

ELF(Executable and Linking Format)是一种对象文件的格式，用于定义不同类型的对象文件(Object files)中都放了什么东西、以及都以什么样的格式去放这些东西。

首先我们需要知道对象文件有哪些：

类型 实例

可重定位的对象文件(Relocatable file) .o;.a;.ko

可执行的对象文件(Executable file) vi、gdb、及我们用链接器生成的可执行文件、bash shell 程序

可被共享的对象文件(Shared object file) .so

核心转储文件(Core Dump File) 当进程意外终止时，系统可以将该进程的地址空间的内容及终止时的一些其他信息转储到核心转储文件 core dump

想要知道一个对象文件属于以上类型的哪一种，我们可以使用file +对象文件名命令来查看，例如：

file sum.o sub.o test.o libsub.so test

sum.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

sub.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped

libsub.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped

test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped

1

2

3

4

5

6

那对于 file 命令来说，它又能如何知道这些信息？答案是在ELF对象文件的最前面有一个ELF文件头，里面记载了所适用的处理器、对象文件类型等各种信息。

首先，ELF文件格式提供了两种视图，分别是链接视图和执行视图。

链接视图是以节（section）为单位，执行视图是以段（segment）为单位。链接视图就是在链接时用到的视图，而执行视图则是在执行时用到的视图。上图左侧的视角是从链接来看的，右侧的视角是执行来看的。总个文件可以分为四个部分：

- ELF header： 描述整个文件的组织。

- Program Header Table: 描述文件中的各种segments，用来告诉系统如何创建进程映像的。

- sections 或者 segments：segments是从运行的角度来描述elf文件，sections是从链接的角度来描述elf文件，也就是说，在链接阶段，我们可以忽略program header table来处理此文件，在运行阶段可以忽略section header table来处理此程序（所以很多加固手段删除了section header table）。从图中我们也可以看出，segments与sections是包含的关系，一个segment包含若干个section。

- Section Header Table: 包含了文件各个segction的属性信息，我们都将结合例子来解释。

1

2

3

4

程序头部表（Program Header Table），如果存在的话，告诉系统如何创建进程映像。

节区头部表（Section Header Table）包含了描述文件节区的信息，比如大小、偏移等。

到这里可能大家伙还是不明白节区跟段的区别与联系，这里在说明一下：

一个程序中最重要的部分是段和节，他们是真正的程序体，存储程序执行所需要的数据，程序中有很多段，常见的有代码段和数据段，段是由节组成的。多个节经过链接之后被合并成一个段。

段和节的信息用header来描述，程序头是program header，节头是section header。

程序中段的大小和数量不固定，节也是如此，因此需要一个专门的数据结构来描述他们，这个就是程序头表和节头表，他们用来存储多个程序头和节头，相当于数组的概念。

2.2 ELF文件详细内容解析

这里我们先分析一个.KO文件的内容，包括以下部分

① ELF Header

② section

③ 节区头部表

接下来我们具体分析每个结构体：

2.2.1 ELF Header

我们打开/usr/include/elf.h文件，开始处是一个ELF头部(ELF Header)，用来描述整个文件的组织，这些信息独立于处理器，也独立于文件中的其余内容。

typedef struct

{

unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/

Elf32\_Half e\_type; /\* Object file type \*/

Elf32\_Half e\_machine; /\* Architecture \*/

Elf32\_Word e\_version; /\* Object file version \*/

Elf32\_Addr e\_entry; /\* Entry point virtual address \*/

Elf32\_Off e\_phoff; /\* Program header table file offset \*/

Elf32\_Off e\_shoff; /\* Section header table file offset \*/

Elf32\_Word e\_flags; /\* Processor-specific flags \*/

Elf32\_Half e\_ehsize; /\* ELF header size in bytes \*/

Elf32\_Half e\_phentsize; /\* Program header table entry size \*/

Elf32\_Half e\_phnum; /\* Program header table entry count \*/

Elf32\_Half e\_shentsize; /\* Section header table entry size \*/

Elf32\_Half e\_shnum; /\* Section header table entry count \*/

Elf32\_Half e\_shstrndx; /\* Section header string table index \*/

} Elf32\_Ehdr;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

我们来分析这个结构体：

① e\_ident数组(16 bytes)

给出了ELF的一些标识信息

1) EI\_MAG: 魔数(4 bytes): 标志此文件是一个 ELF 目标文件

1.1) e\_ident[0]: 0x7f

1.2) e\_ident[1]: ‘E’

1.3) e\_ident[2]: ‘L’

1.4) e\_ident[3]: ‘F’

2) EI\_CLASS: 文件的类别(1 byte):

或者说是位宽

2.1) ELFCLASSNONE: 0: 非法类别

2.2) ELFCLASS32: 1: 32位目标

2.3) ELFCLASS64: 2: 64位目标

3) EI\_DATA: 处理器特定数据的数据编码方式

3.1) ELFDATANONE: 0: 非法数据编码

3.2) ELFDATA2LSB: 1: 高位在前

3.3) ELFDATA2MSB: 2: 低位在前

4) EI\_VERSION: ELF 头部的版本号码，此值必须是EV\_CURRENT

5) EI\_PAD: 标记e\_ident 中未使用字节的开始，初始化为0

② e\_type(2 bytes): 目标文件类型

1) ET\_NONE: 0: 未知目标文件格式

2) ET\_REL: 1: 可重定位文件

3) ET\_EXEC: 2: 可执行文件

4) ET\_DYN: 3: 共享目标文件

5) ET\_CORE: 4: Core文件(转储格式0

6) ET\_LOPROC: 0xff00: 特定处理器文件

7) ET\_HIPROC: 0xffff: 特定处理器文件

ET\_LOPROC和ET\_HIPROC之间的取值用来标识与处理器相关的文件格式

③ e\_machine(2 bytes): 文件的目标体系结构类型

1) EM\_NONE: 0: 未指定

2) EM\_M32: 1: AT&T WE 32100

3) EM\_SPARC: 2: SPARC

4) EM\_386: 3: Intel 80386

5) EM\_68K: 4: Motorola 68000

6) EM\_88K: 5: Motorola 88000

7) EM\_860: 7: Intel 80860

8) EM\_MIPS: 8: MIPS RS3000

④ e\_version(4 bytes): 目标文件版本

1) EV\_NONE: 0: 非法版本

2) EV\_CURRENT: 1: 当前版本

⑤ e\_entry(4 bytes): 程序入口的虚拟地址，如果目标文件没有程序入口，可以为0

⑥ e\_phoff(4 bytes): 程序头部表格(Program Header Table)的偏移量(按字节计算)，如果文件没有程序头部表格，可以为0

⑦ e\_shoff(4 bytes): 节区头部表格(Section Header Table)的偏移量(按字节计算)。如果文件没有节区头部表格，可以为0

⑧ e\_flags(4 bytes): 保存与文件相关的，特定于处理器的标志。标志名称采用EF\_machine\_flag的格式

⑨ e\_ehsize(2 bytes): ELF头部的大小(以字节计算)

⑩ e\_phentsize(2 bytes): 程序头部表格的表项大小(按字节计算)

11. e\_phnum(2 bytes): 程序头部表格的表项数目，可以为0

12. e\_shentsize(2 bytes): 节区头部表格的表项大小(按字节计算)

13. e\_shnum(2 bytes): 节区头部表格的表项数目，可以为0

14. e\_shstrndx(2 bytes): 节区头部表格中与节区名称字符串表相关的表项的索引。如果文件没有节区名称字符串表，此参数可以为SHN\_UNDEF

这里我们展示读取到的头部信息：

2.2.2 节区

节区满足以下条件：

目标文件中的每个节区都有对应的节区头部描述它，反过来，有节区头部不意味着有节区

每个节区占用文件中一个连续字节区域(这个区域可能长度为0)

文件中的节区不能重叠，不允许一个字节存在于两个节区中的情况发生

目标文件中可能包含非活动空间(INACTIVE SPACE)。这些区域不属于任何头部和节区，其内容未指定

2.2.3 节区头部表

typedef struct

{

Elf32\_Word sh\_name; /\* Section name (string tbl index) \*/

Elf32\_Word sh\_type; /\* Section type \*/

Elf32\_Word sh\_flags; /\* Section flags \*/

Elf32\_Addr sh\_addr; /\* Section virtual addr at execution \*/

Elf32\_Off sh\_offset; /\* Section file offset \*/

Elf32\_Word sh\_size; /\* Section size in bytes \*/

Elf32\_Word sh\_link; /\* Link to another section \*/

Elf32\_Word sh\_info; /\* Additional section information \*/

Elf32\_Word sh\_addralign; /\* Section alignment \*/

Elf32\_Word sh\_entsize; /\* Entry size if section holds table \*/

} Elf32\_Shdr;

1

2

3

4

5

6

7

8

9

10

11

12

13

sh\_name(4 bytes): 节区名称，是"节区头部字符串表节区"(Section Header String Table Section)的索引。名字是一个NULL结尾的字符串。

所谓"节区头部字符串表节区"，就是一段连续的保存每个节区名字的ascii字符的地址空间

sh\_type(4 bytes): 为节区的内容和语义进行分类

1) SHT\_NULL: 0: 此值标志节区头部是非活动的，没有对应的节区。此节区头部中的其他成员取值无意义

2) SHT\_PROGBITS: 1: 此节区包含程序定义的信息，其格式和含义都由程序来解释

3) SHT\_SYMTAB: 2: 此节区包含一个符号表。目前目标文件对每种类型的节区都只能包含一个，不过这个限制将来可能发生变化，通常情况下，SHT\_SYMTAB节区提供用于链接编辑(指ld而言)的符号，尽管也可用来实现动态

链接。

4) SHT\_STRTAB: 3: 此节区包含字符串表。目标文件可能包含多个字符串表节区。

5) SHT\_RELA: 4: 此节区包含重定位表项，其中可能会有补齐内容(addend)，例如32位目标文件中的Elf32\_Rela类型。目标文件可能拥有多个重定位节区

6) SHT\_HASH: 5: 此节区包含符号哈希表。所有参与动态链接的目标都必须包含一个符号哈希表。目前，一个目标文件只能包含一个哈希表，不过此限制将来可能会解除。

7) SHT\_DYNAMIC: 6: 此节区包含动态链接的信息。目前一个目标文件中只能包含一个动态节区，将来可能会取消这一限制。

8) SHT\_NOTE: 7: 此节区包含以某种方式来标记文件的信息。

9) SHT\_NOBITS: 8: 这种类型的节区不占用文件中的空间，其他方面和SHT\_PROGBITS相似。尽管此节区不包含任何字节，成员sh\_offset中还是会包含概念性的文件偏移

10) SHT\_REL: 9: 此节区包含重定位表项，其中没有补齐(addends)，例如32位目标文件中的Elf32\_rel类型。目标文件中可以拥有多个重定位节区

11) SHT\_SHLIB: 10: 此节区被保留，不过其语义是未规定的。包含此类型节区的程序与ABI不兼容。

12) SHT\_DYNSYM: 11: 作为一个完整的符号表，它可能包含很多对动态链接而言不必要的符号。因此，目标文件也可以包含一个SHT\_DYNSYM节区，其中保存动态链接符号的一个最小集合，以节省空间

13) SHT\_LOPROC(0x70000000)~SHT\_HIPROC(0x7FFFFFFF): 这一段(包括两个边界)，是保留给处理器专用语义的

14) SHT\_LOUSER(0X80000000): 此值给出保留给应用程序的索引下界

15) SHT\_HIUSER(0X8FFFFFFF): 此值给出保留给应用程序的索引上界

sh\_flags(4 bytes): sh\_flags字段定义了一个节区中包含的内容是否可以修改、是否可以执行等信息。如果一个标志位被设置，则该位取值为1。未定义的各位都设置为0(这是一种bitmap位图表示法)

1) SHF\_WRITE: 0x1: 节区包含进程执行过程中将可写的数据

2) SHF\_ALLOC: 0x2: 此节区在进程执行过程中占用内存。某些控制节区并不出现于目标文件的内存映像中，对于那些节区，此位应设置为0

3) SHF\_EXECINSTR: 0x4: 节区包含可执行的机器指令

4) SHF\_MASKPROC: 0xF0000000: 所有包含于此掩码中的四位都用于处理器专用的语义

sh\_addr(4 bytes): 如果节区将出现在进程的内存映像中，此成员给出节区的第一个字节应处的位置。否则，此字段为0

sh\_offset(4 bytes): 此成员的取值给出节区的第一个字节与文件头之间的偏移。不过，SHT\_NOBITS类型的节区不占用文件的空间，因此其sh\_offset成员给出的是其概念性的偏移

sh\_size(4 bytes): 此成员给出节区的长度(字节数)。除非节区的类型是SHT\_NOBITS，否则节区占用文件中的sh\_size 字节。类型为SHT\_NOBITS的节区长度可能非零，不过却不占用文件中的空间

sh\_link(4 bytes): 此成员给出节区头部表索引链接。其具体的解释依赖于节区类型

根据节区类型的不同，sh\_link和sh\_info 的具体含义也有所不同

sh\_type 　　　　sh\_link 　　　　　　sh\_info

SHT\_DYNAMIC 　　　　此节区中条目所用到的字符串表格的节区头部索引 0

SHT\_HASH 　　　　　 此哈希表所适用的符号表的节区头部索引 　 0

SHT\_REL、SHT\_RELA 　　　相关符号表的节区头部索引 　　　　　重定位所适用的节区的节区头部索引

SHT\_SYMTAB、SHT\_DYNSYM 相关联的字符串表的节区头部索引 　　最后一个局部符号(绑定 STB\_LOCAL)的符号表索引值加一

其它 　　　　　　SHN\_UNDEF 　　　　　　　0

sh\_info(4 bytes): 此成员给出附加信息，其解释依赖于节区类型

sh\_addralign(4 bytes): 某些节区带有地址对齐约束。例如，如果一个节区保存一个doubleword，那么系统必须保证整个节区能够按双字对齐。sh\_addr对sh\_addralign取模，结果必须为0。目前仅允许取值为0和2的幂

次数。数值0和1表示节区没有对齐约束

sh\_entsize(4 bytes): 某些节区中包含固定大小的项目，如符号表。对于这类节区，此成员给出每个表项的长度字节数。如果节区中并不包含固定长度表项的表格，此成员取值为0

这里我们展示读取到的节区头部表：

2.2.4 部分节区说明

1 .rel.xxxxx

对应xxxxx section的relocate表，用于符号重定位,比如说 .rel.text 就是 .text 的重定向表所在的节区

Relocation section '.rel.text' at offset 0xe52c contains 67 entries:

Offset Info Type Sym.Value Sym. Name

00000050 00000e02 R\_ARM\_ABS32 00000000 .rodata.str1.4

00000054 00005802 R\_ARM\_ABS32 00000000 memcpy

00000058 00006902 R\_ARM\_ABS32 00000000 printk

0000005c 00000e02 R\_ARM\_ABS32 00000000 .rodata.str1.4

00000120 00006c02 R\_ARM\_ABS32 00000000 \_ctype

00000124 00007502 R\_ARM\_ABS32 00000000 strlen

000001c8 00007502 R\_ARM\_ABS32 00000000 strlen

000001cc 00006402 R\_ARM\_ABS32 00000000 match\_int

000001d0 00006c02 R\_ARM\_ABS32 00000000 \_ctype

000001d4 00007302 R\_ARM\_ABS32 00000000 match\_hex

000001fc 0000521c R\_ARM\_CALL 00000128 parse\_interger

0000023c 00006302 R\_ARM\_ABS32 00000000 kallsyms\_lookup\_name

00000290 0000741c R\_ARM\_CALL 00000060 str\_trim

0000029c 0000521c R\_ARM\_CALL 00000128 parse\_interger

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

2 .symtab 模块中所有的符号记录都在这里面

eadelf -s ssp.ko | more

Symbol table '.symtab' contains 53529 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000000 0 SECTION LOCAL DEFAULT 1

2: 00000000 0 SECTION LOCAL DEFAULT 3

3: 00000000 0 SECTION LOCAL DEFAULT 5

4: 00000000 0 SECTION LOCAL DEFAULT 7

5: 00000000 0 SECTION LOCAL DEFAULT 8

6: 00000000 0 SECTION LOCAL DEFAULT 10

7: 00000000 0 SECTION LOCAL DEFAULT 13

8: 00000000 0 SECTION LOCAL DEFAULT 23

9: 00000000 0 SECTION LOCAL DEFAULT 24

10: 00000000 0 SECTION LOCAL DEFAULT 31

11: 00000000 0 NOTYPE LOCAL DEFAULT 1 $a

12: 00000088 0 NOTYPE LOCAL DEFAULT 1 $d

13: 00000000 0 NOTYPE LOCAL DEFAULT 28 $d

14: 00000098 0 NOTYPE LOCAL DEFAULT 1 $a

15: 00000150 0 NOTYPE LOCAL DEFAULT 1 $d

16: 00000164 0 NOTYPE LOCAL DEFAULT 1 $a

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

3…shstrtab

节区名称字符串表，所有的节区名称字符串都放在这里

readelf -p 35 ssp.ko

String dump of section '.shstrtab':

[ 1] .symtab

[ 9] .strtab

[ 11] .shstrtab

[ 1b] .rel.text

[ 25] .rel.text.unlikely

[ 38] .rel.init.text

[ 47] .rel.rodata

[ 53] .rodata.str1.4

[ 62] .rel.pv\_table

[ 70] .ARM.extab.text.unlikely

[ 89] .rel.ARM.exidx.text.unlikely

[ a6] .ARM.extab.init.text

[ bb] .rel.ARM.exidx.init.text

[ d4] .ARM.extab.exit.text

[ e9] .rel.ARM.exidx.exit.text

[ 102] .modinfo

[ 10b] .ARM.extab

[ 116] .rel.data

[ 120] .rel.gnu.linkonce.this\_module

[ 13e] .rel.ARM.exidx

[ 14d] .note.gnu.build-id

[ 160] .bss

[ 165] .comment

[ 16e] .note.GNU-stack

[ 17e] .ARM.attributes

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

4 .strtab

符号名称字符串表，这里记录了所有的符号的名称字符串,其格式即字符串表格式，和节区头部名称字符串表格式相同。

5 .gnu.linkonce.this\_module

struct module 实例所在的节区

————————————————

版权声明：本文为CSDN博主「乱世半仙」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/zj82448191/article/details/108441447>

[(6条消息) 内核模块加载过程分析\_晓镁的博客-CSDN博客](https://blog.csdn.net/realmardrid/article/details/113357857?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-1.pc_relevant_paycolumn_v3&spm=1001.2101.3001.4242.2&utm_relevant_index=4)

<https://blog.csdn.net/realmardrid/article/details/113357857?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-1.pc_relevant_paycolumn_v3&spm=1001.2101.3001.4242.2&utm_relevant_index=4>

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，似乎除了校验的时候用了一下，再也没用过

Elf\_Shdr \*sechdrs; // 节区头部表

char \*secstrings, \*strtab; // section 名称表， 字符名称表

unsigned long symoffs, stroffs; // 符号表，字符串表在最终core section中的偏移

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, info, pcpu;

} 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 future. \*/

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() returns \*/

void \*module\_init;

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

void \*module\_core;

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

unsigned int init\_size, core\_size;

/\* 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;

/\* Arch-specific module values \*/

struct mod\_arch\_specific arch;

unsigned int taints; /\* same bits as kernel:tainted \*/

#ifdef CONFIG\_KALLSYMS

/\*

\* We keep the symbol and string tables for kallsyms.

\* The core\_\* fields below are temporary, loader-only (they

\* could really be discarded after module init).

\*/

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). People like

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。

在这里插入图片描述

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：

在这里插入图片描述

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

在这里插入图片描述

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 = "my\_exp\_function";

static const struct kernel\_symbol \_\_kstrtab\_my\_exp\_function =

{(unsigned long )&my\_exp\_function, \_kstrtab\_my\_exp\_function }

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。

在这里插入图片描述

在这里插入图片描述

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：

在这里插入图片描述

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：

在这里插入图片描述

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。