廖威雄: 利用__attribute__((section()))构建初始化函数表与Linux内核init的实现

原创 廖威雄 Linux阅码场 2018-01-11 12:52

本文详细讲解了利用__attribute__((section()))构建初始化函数表，以及Linux内核各级初始化的原理。

作者简介：

廖威雄，2016年本科毕业于暨南大学，目前就职于珠海全志科技股份有限公司从事linux嵌入式系统(Tina Linux)的开发，主要负责文件系统和存储的开发和维护，兼顾linux测试系统的设计和持续集成的维护。

拆书帮珠海百岛分舵的组织长老，二级拆书家，热爱学习，热爱分享。

欢迎投稿：

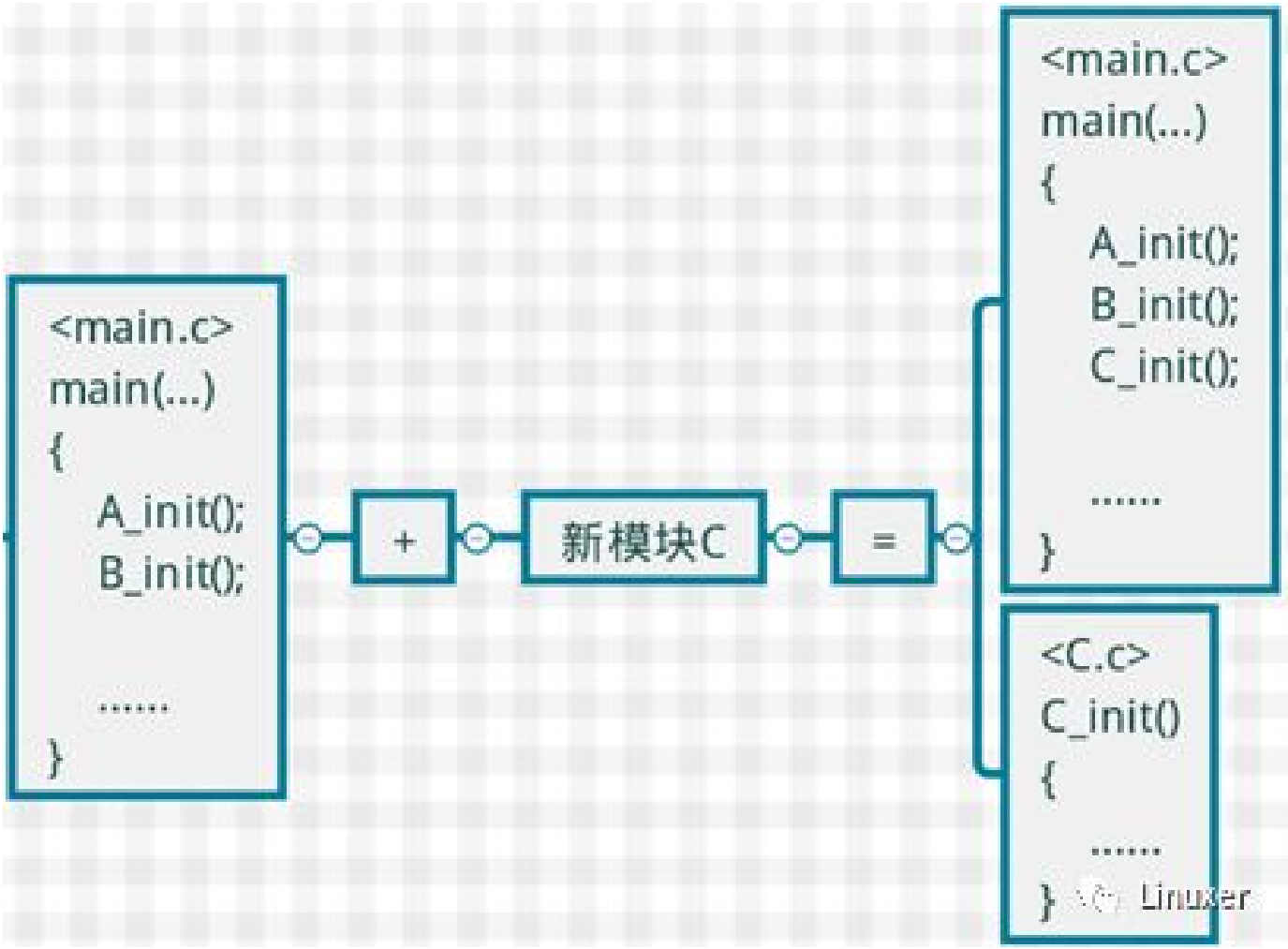
2018年给Linuxer投稿原创Linux技术文章，一经录取，赠送人民邮电出版社任意在售图书，获得读者红包打赏，和公众号站长宋宝华200元微信红包。

Linuxer-"Linux开发者自己的媒体"第五月稿件和赠书名单

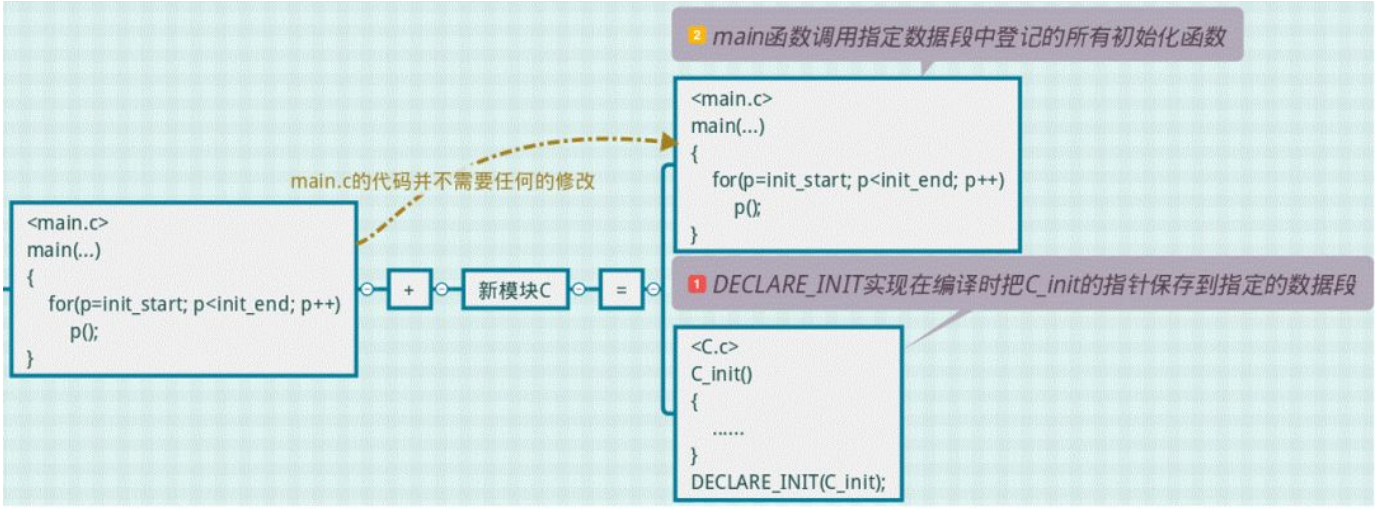
欢迎关注Linuxer

问题导入

传统的应用编写时，每添加一个模块，都需要在main中添加新模块的初始化



使用__attribute__((section()))构建初始化函数表后，由模块告知main：“我要初始化“，添加新模块再也不需要在main代码中显式调用模块初始化接口。



以此实现main与模块之间的隔离，main不再关心有什么模块，模块的删减也不需要修改main。

那么，如何实现这个功能呢？如何实现DECLARE_INIT呢？联想到内核驱动，所有内核驱动的初始化函数表在哪里？为什么添加一个内核驱动不需要修改初始化函数表？

下文会从 构建初始化函数表的原理分析、分析内核module_init实现、演练练习 的3个角度给小伙伴分享。

构建初始化函数表的原理分析

__attribute__((section("name")))是gcc编译器支持的一个编译特性（arm编译器也支持此特性），实现在编译时把某个函数/数据放到name的数据段中。因此实现原理就很简单了：

1. 模块通过__attribute__((section("name")))的实现，在编译时把初始化的接口放到name数据段中
2. main在执行初始化时并不需要知道有什么模块需要初始化，只需要把name数据段中的所有初始化接口执行一遍即可

首先： gcc -c test.c -o test.o

此时编译过程中处理了__atribute__((section(XXX)))，把标记的变量/函数放到了test.o的XXX的数据段，可用 readelf命令查询。

最后：ld -T <ldscript> test.o -otest.bin

链接时，test.o的XXX数据段（输入段），最终保存在test.bin的XXX数据段（输出段），如此在bin中构建了初始化函数表。

由于自定义了一个数据段，而默认链接脚本缺少自定义的数据段的声明，因此并不能使用默认的链接脚本。

ld链接命令有两个关键的选项：

ld -T <script>：指定链接时的链接脚本
ld --verbose：打印出默认的链接脚本

在我们下文的演练中，我们首先通过”ld --verbose”获取默认链接脚本，然后修改链接脚本，添加自定义的段，最后在链接应用时通过“-T<script>” 指定我们修改后的链接脚本。

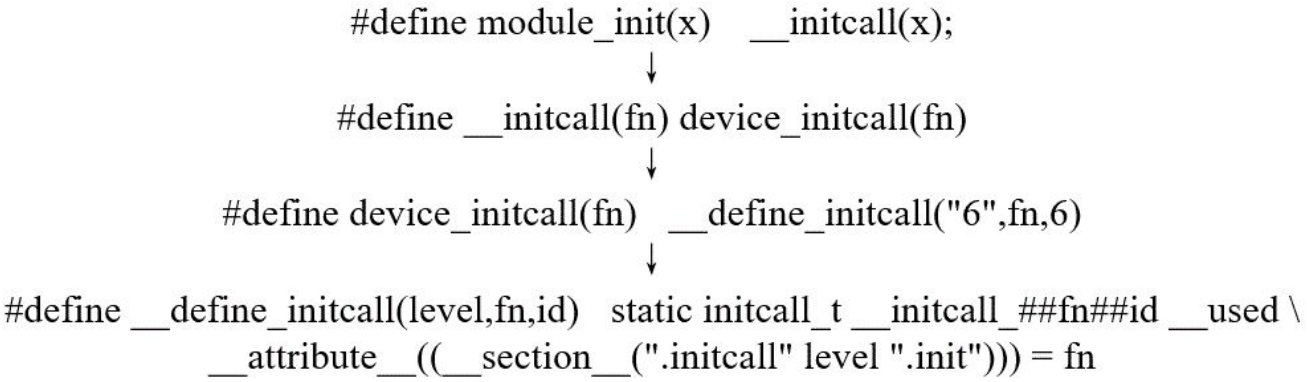
下文，我们首先分析内核module_init的实现，最后进行应用程序的演练练习。

分析内核module_init实现

内核驱动的初始化函数表在哪里？为什么添加一个内核驱动不需要修改初始化函数表？为什么所有驱动都需要module_init？

1. module_init的定义

module_init定义在<include/linux/init.h>。代码如下：



代码中使用的“_section_”，是一层层的宏，为了简化，把其等效理解为“section”。

分析上述代码，我们发现module_init由__attribute__((section(“name”)))实现，把初始化函数地址保存到名为".initcall6.init" 的数据段中。

2. 链接内核使用自定义的链接脚本

我们看到内核目录最上层的Makefile，存在如下代码：

```
# Rule to link vmlinux - also used during CONFIG_KALLSYMS
# May be overridden by arch/$(ARCH)/Makefile
quiet_cmd_vmlinux__ ?= LD      $@
cmd_vmlinux__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
     -T $(vmlinux-lds) $(vmlinux-init)                    \
     --start-group $(vmlinux-main) --end-group            \
     $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)
```

本文的关注点在于：-T $(vmlinux-lds)，通过“ld -T <script>”使用了定制的链接脚本。定制的链接脚本在哪里呢？在Makefile存在如下代码：

```
vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
```

我们以”ARCH=arm“ 为例，查看链接脚本：arch/arm/kernel/vmlinux.lds：

```
.init.data : {
    *(.init.data) *(.meminit.data) *(.init.rodata) *(.meminit.rodata) . = ALIGN(32);
__dtb_start = .; *(.dtb.init.rodata) __dtb_end = .;
    . = ALIGN(16); __setup_start = .; *(.init.setup) __setup_end = .;
    __initcall_start = .; *(.initcallearly.init) __initcall0_start = .; *(.initcall0.init)
*(.initcall0s.init) __initcall1_start = .; *(.initcall1.init) *(.initcall1s.init) __initcall2_start = .;
*(.initcall2.init) *(.initcall2s.init) __initcall3_start = .; *(.initcall3.init) *(.initcall3s.init)
__initcall4_start = .; *(.initcall4.init) *(.initcall4s.init) __initcall5_start = .; *(.initcall5.init)
*(.initcall5s.init) __initcallrootfs_start = .; *(.initcallrootfs.init) *(.initcallrootfss.init)
__initcall6_start = .; *(.initcall6.init) *(.initcall6s.init) __initcall7_start = .; *(.initcall7.init)
*(.initcall7s.init) __initcall_end = .;
    __con_initcall_start = .; *(.con_initcall.init) __con_initcall_end = .;
    __security_initcall_start = .; *(.security_initcall.init) __security_initcall_end = .;
    . = ALIGN(4); __initramfs_start = .; *(.init.ramfs) . = ALIGN(8); *(.init.ramfs.info)
  }
```


在上述代码中，我们聚焦于两个地方：

__initcall6_start = .; ： 由__initcall6_start指向当前地址
*(.initcall6.init) ： 所有.o文件的.initcall6.init数据段放到当前位置

如此，“__initcall6_start”指向“.initcall6.init”数据段的开始地址，在应用代码中就可通过“__initcall6_start”访问数据段“.initcall6.init”。

是不是如此呢？我们再聚焦到文件<init/main.c>中。

“.initcall.init”数据段的使用

在<init/main.c>中，有如下代码：

```
static initcall_t *initcall_levels[] __initdata = {
      __initcall0_start,
      __initcall1_start,
      __initcall2_start,
      __initcall3_start,
      __initcall4_start,
      __initcall5_start,
      __initcall6_start,
      __initcall7_start,
      __initcall_end,
};
......
int __init_or_module do_one_initcall(initcall_t fn)
{
        ......
    if (initcall_debug)
        ret = do_one_initcall_debug(fn);
    else
        ret = fn();
        ......
}
......
static void __init do_initcall_level(int level)
{
    ......
    for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
        do_one_initcall(*fn);
}
```

按0-7的初始化级别，依次调用各个级别的初始化函数表，而驱动module_init的初始化级别为6。在“for (fn = initcall_levels[level]; fn <initcall_levels[level+1]; fn++)”的for循环调用中，实现了遍历当前初始化级别的所有初始化函数。

module_init的实现总结

通过上述的代码追踪，我们发现module_init的实现有以下关键步骤：

1. 通过module_init的宏，在编译时，把初始化函数放到了数据段：.initcall6.init
2. 在链接成内核的时候，链接脚本规定好了.initcall6.init的数据段以及指向数据段地址的变量：_initcall6_start
3. 在init/main.c中的for循环，通过_initcall6_start的指针，调用了所有注册的驱动模块的初始化接口
4. 最后通过Kconfig/Makefile选择编译的驱动，实现只要编译了驱动代码，则自动把驱动的初始化函数构建到统一的驱动初始化函数表

演练练习

分析了内核使用__attribute__((section(“name”)))构建的驱动初始化函数表，我们接下来练习如何在应用中构建自己的初始化函数表。

下文的练习参考了：https://my.oschina.net/u/180497/blog/177206

1. 应用代码

我们的练习代码（section.c）如下：

```
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>

typedef void (*init_call)(void);

/*
 * These two variables are defined in link script.
 */
extern init_call _init_start;
extern init_call _init_end;

#define _init __attribute__((unused, section(".myinit")))
#define DECLARE_INIT(func) init_call _fn_##func _init = func

static void A_init(void)
{
    write(1, "A_init\n", sizeof("A_init\n"));
}
DECLARE_INIT(A_init);

static void B_init(void)
{
    printf("B_init\n");
}
DECLARE_INIT(B_init);

static void C_init(void)
{
    printf("C_init\n");
}
DECLARE_INIT(C_init);
/*
```

```
/*
 * DECLARE_INIT like below:
 *  static init_call _fn_A_init __attribute__((unused, section(".myinit"))) = A_init;
 *  static init_call _fn_C_init __attribute__((unused, section(".myinit"))) = C_init;
 *  static init_call _fn_B_init __attribute__((unused, section(".myinit"))) = B_init;
 */


void do_initcalls(void)
{
    init_call *init_ptr = &_init_start;
    for (; init_ptr < &_init_end; init_ptr++) {
        printf("init address: %p\n", init_ptr);
        (*init_ptr)();
    }
}


int main(void)
{
    do_initcalls();
    return 0;
}
```

在代码中，我们做了3件事：

a. 使用__attribute__((section()))定义了宏：DECLARE_INIT，此宏把函数放置到初始化函数表

b. 使用DELCARE_INIT的宏，声明了3个模块初始化函数：A_init/B_init/C_init

c. 在main中通过调用do_initcalls函数，依次调用编译时构建的初始化函数。其中，“_init_start”和“_init_end”的变量在链接脚本中定义。

2. 链接脚本

通过命令”ld --verbose”获取默认链接脚本：

```
GNU ld (GNU Binutils for Ubuntu) 2.24
  支持的仿真：
   elf_x86_64
   ……
使用内部链接脚本：
==================================================
XXXXXXXX （缺省链接脚本）


==================================================
```

我们截取分割线”=====“之间的链接脚本保存为：ldscript.lds

在.bss的数据段前添加了自定义的数据段：

```
_init_start = .;
.myinit : { *(.myinit) }
_init_end = .;
```

”_init_start“和”_init_end“是我们用于识别数据段开始和结束的在链接脚本中定义的变量，而.myinit则是数据段的名称，其中：

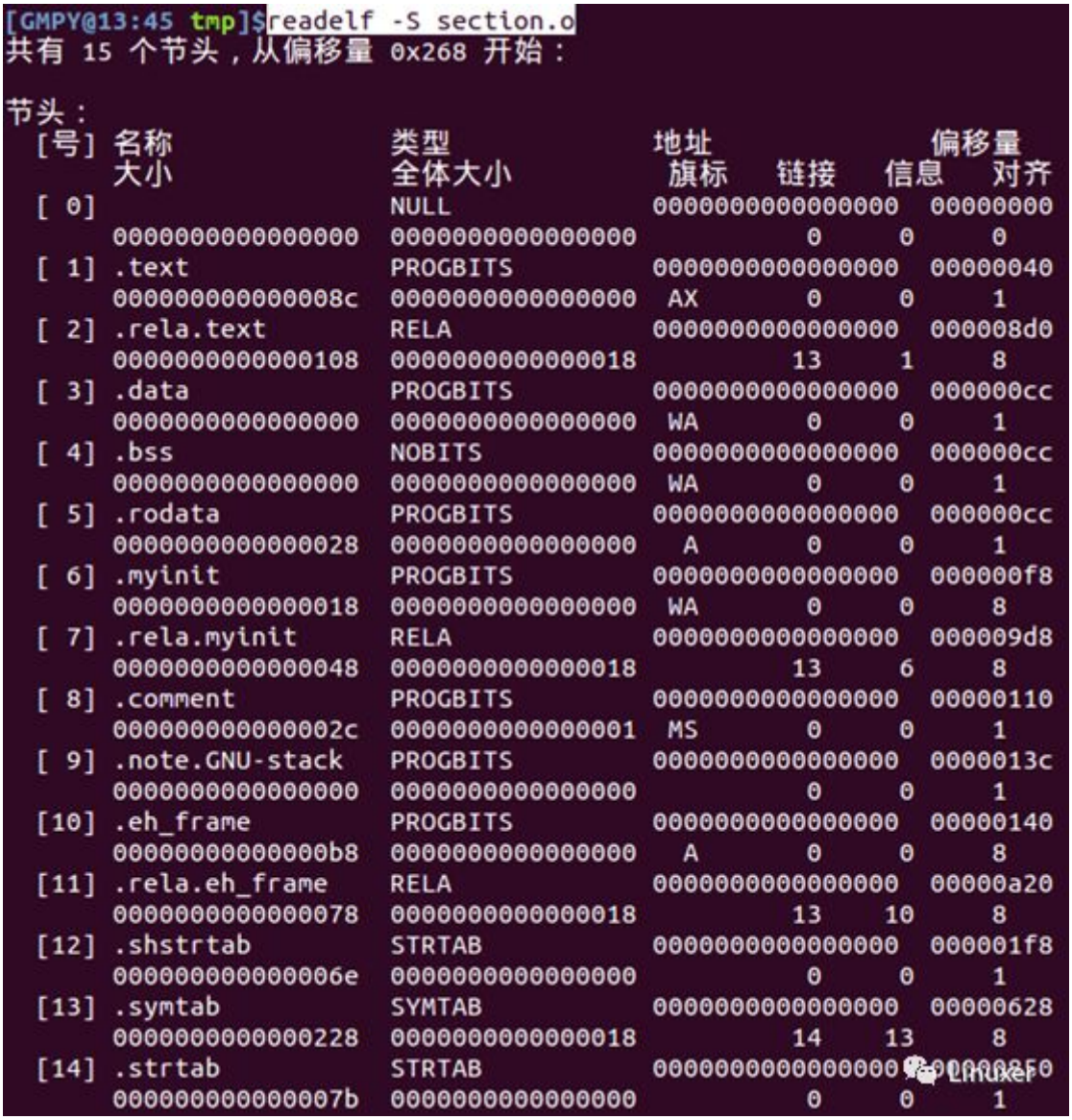
.myinit : { *(.myinit) }：表示.o中的.myinit数据段（输入段）保存到bin中的.myinit数据段（输出段）中

前期准备充足，下面进行编译、链接、执行的演示

3. 编译

执行：gcc -c section.c -o section.o 编译应用源码。

执行：readelf -S section.o 查看段信息，截图如下：

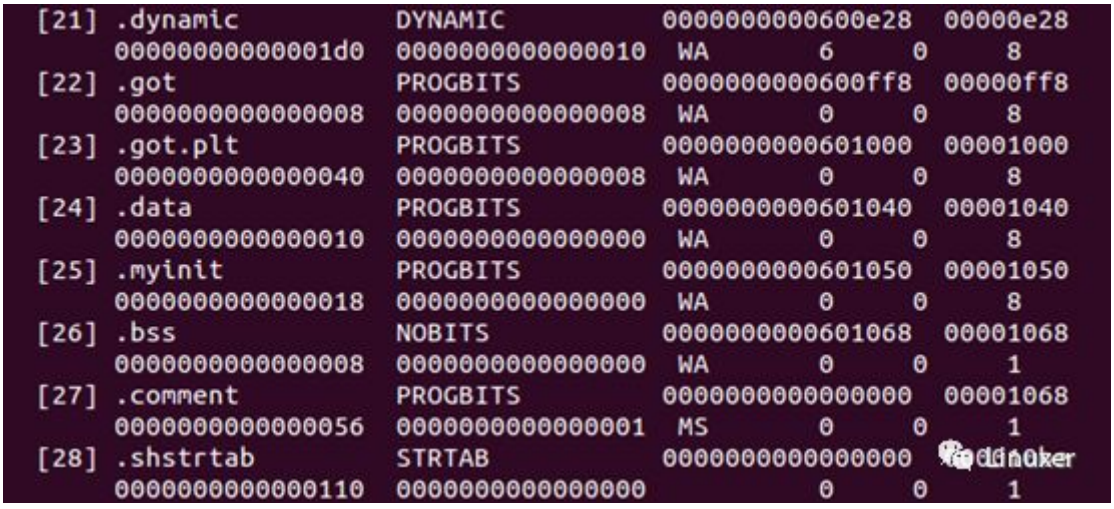


可以看到，段[6]是我们自定义的数据段

4. 链接

执行：gcc -T ldscript.lds section.o -o section 链接成可执行的bin文件

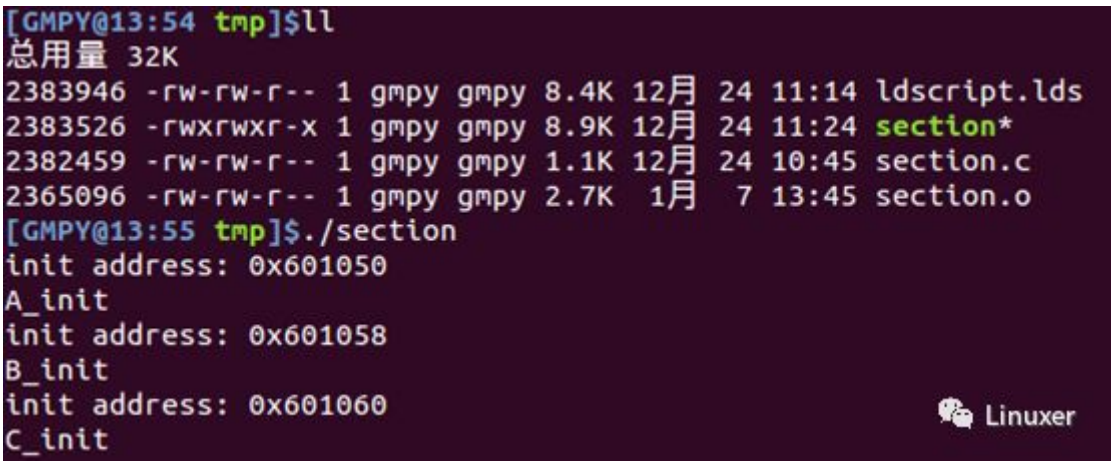
执行：readelf -S section 查看bin文件的段分布情况，部分截图如下：



在我链接成的可执行bin中，在[25]段中存在我们自定义的段

5. 执行

执行结果：



本文后面跟着的一篇文章是关于这篇文章对应的高清思维导图。