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。

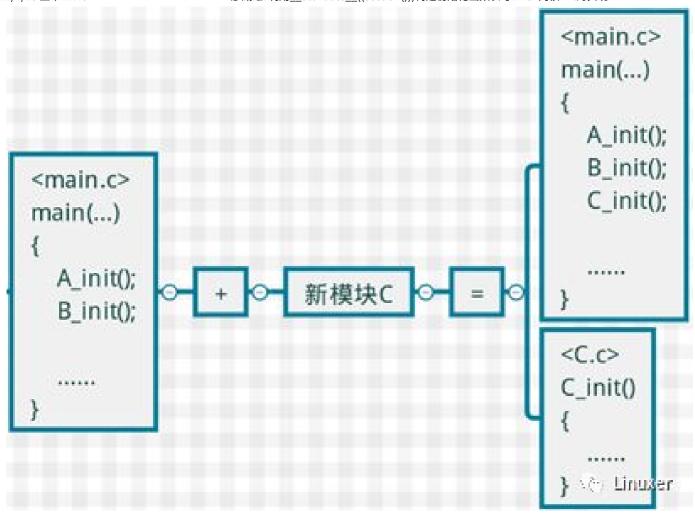
Linuxer-"Linux开发者自己的媒体"第五月稿件和赠书名单

欢迎关注Linuxer

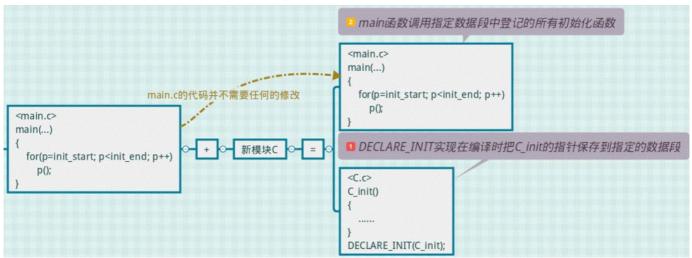


问题导入

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:

- 模块通过\_\_attribute\_\_((section("name")))的实现,在编译时把初始化的接口 1. 放到name数据段中
- 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 2. 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: Id -T < Idscript > 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:

Id -T <script>: 指定链接时的链接脚本 Id --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?

module init的定义

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

```
\#define module init(x) initcall(x);
                     #define initcall(fn) device initcall(fn)
             #define device_initcall(fn) __define_initcall("6",fn,6)
#define define initcall(level,fn,id) static initcall t initcall ##fn##id used \
               attribute (( section (".initcall" level ".init"))) = fn
```

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);
      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
- 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 3. 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++) {</pre>
     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:

- 使用 attribute ((section()))定义了宏: DECLARE INIT, 此宏把函数放置到 a. 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"Id --verbose"获取默认链接脚本:

```
GNU Id (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:

22/2/25 上十	11.45	/参风/组、个小用attinbut	e((Section()))构建初始化图	X1X-J LITIUX P 3/XI
[GMPY@: 共有 1	13:45 tmp]\$ <mark>readelf</mark> 5 个节头,从偏移量	-S section.o 0x268 开始:		
节头:				
	名称	类型	地址	偏移量
[ ]]	天小	全体大小	旗标 链接 信	
[0]		NULL	00000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
[1]	.text	PROGBITS	0000000000000000	00000040
	000000000000008c	0000000000000000	AX 0 0	1
[2]	.rela.text	RELA	00000000000000000	000008d0
	0000000000000108	0000000000000018	13 1	8
[ 3]	.data	PROGBITS	00000000000000000	000000cc
	0000000000000000	0000000000000000	WA 0 0	1
[ 4]	.bss	NOBITS	00000000000000000	000000cc
	0000000000000000	0000000000000000	WA 0 0	1
[ 5]	.rodata	PROGBITS	00000000000000000	000000cc
	0000000000000028	0000000000000000	A 0 0	1
[ 6]		PROGBITS	00000000000000000	000000f8
	0000000000000018	0000000000000000	WA 0 0	8
[ 7]	.rela.myinit	RELA	00000000000000000	8be0000
200	0000000000000048	0000000000000018	13 6	8
[ 8]	.comment	PROGBITS	00000000000000000	00000110
20.00	0000000000000002c	00000000000000001	MS 0 0	1
[ 9]	.note.GNU-stack	PROGBITS	00000000000000000	0000013c
	0000000000000000	00000000000000000	0 0	1
[10]		PROGBITS	00000000000000000	00000140
2000	0000000000000b8	00000000000000000	A 0 0	8
[11]		RELA	00000000000000000	00000a20
12000000	0000000000000078	0000000000000018	13 10	8
[12]	.shstrtab	STRTAB	0000000000000000	000001f8
4000	0000000000000006e	00000000000000000	0 0	1
[13]		SYMTAB	00000000000000000	00000628
-	0000000000000228	0000000000000018	14 13	8
[14]	.strtab	STRTAB	0000000000000000	
	0000000000000007b	00000000000000000	0 0	1

可以看到,段[6]是我们自定义的数据段

#### 链接 4.

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:

[21]	.dynamic	DYNAMIC	0000000000600e28	00000e28
	00000000000001d0	00000000000000010	WA 6	0 8
[22]	.got	PROGBITS	0000000000600ff8	00000ff8
	8000000000000000	0000000000000008	WA 0	0 8
[23]	.got.plt	PROGBITS	0000000000601000	00001000
	00000000000000040	0000000000000008	WA 0	0 8
[24]	.data	PROGBITS	0000000000601040	00001040
	00000000000000010	0000000000000000	WA 0	0 8
[25]	.myinit	PROGBITS	0000000000601050	00001050
	0000000000000018	0000000000000000	WA 0	0 8
[26]	.bss	NOBITS	0000000000601068	00001068
	8000000000000000	0000000000000000	WA 0	0 1
[27]	.comment	PROGBITS	0000000000000000	00001068
	0000000000000056	00000000000000001	MS 0	0 1
[28]	.shstrtab	STRTAB	0000000000000000	<b>Roddnüker</b>
	0000000000000110	0000000000000000	Θ	0 1

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:

```
[GMPY@13:54 tmp]$ll
总用量 32K
2383946 -rw-rw-r-- 1 gmpy gmpy 8.4K 12月 24 11:14 ldscript.lds
2383526 -rwxrwxr-x 1 gmpy gmpy 8.9K 12月 24 11:24 section*
2382459 -rw-rw-r-- 1 gmpy gmpy 1.1K 12月 24 10:45 section.c
2365096 -rw-rw-r-- 1 gmpy gmpy 2.7K 1月 7 13:45 section.o
[GMPY@13:55 tmp]$./section
init address: 0x601050
A init
init address: 0x601058
B init
init address: 0x601060
                                                     h Linuxer
C_init
```

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。

### 内存管理直播

Linux的任督二脉之内存管理线上微信群直播报名(2018.1.29-2.2)

喜欢此内容的人还喜欢

### BPF内核实现详解

Linux阅码场