模块初始化框架

赵庶林

April 17, 2014

Contents

1 准备工作

1.1 需求分析

设计一个模块的初始化框架。

1.2 测试方法

如何测试?

使用printk打印调试信息。

2 模块init和exit函数

今天的任务就是写出init和exit这两个最基本的函数。

2.1 最简单的hello.c测试文件

为了上手，先测试最简单的hello.c类型的文件。zsl_cache.c文件内容如下：

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Zhao Shulin");
MODULE_DESCRIPTION("First step for my cache");

static int __init zsl_init(void)
{
    printk(KERN_ALERT "hello, zsl.\n");
    return 0;
}
static void __exit zsl_exit(void)
{
    printk(KERN_ALERT "goodbye, zsl.\n");
}

module_init(zsl_init);
module_exit(zsl_exit);
```

Makefile文件内容如下：

```
obj-m := zsl_cache.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
    rm -rf Modules.markers module.order module.symvers
```

然后，编译命令如下：

(1) 执行sudo -i命令：进入root权限；

(2) 执行make命令：生成zsl_cache.ko文件；

(3) 执行insmod ./zsl_cache.ko命令：加载模块；

(4) 执行rmmod zsl_cache 命令：卸载模块。

注意：执行insmod和rmmod命令之后，无法在终端输出打印信息，试图通过命令echo 8 > /proc/sys/kernel/printk 来降低控制台的loglevel，通过命令cat /proc/sys/kernel/printk来查看是否完成。但是依然无法输出到控制台。Google之后，试图通过vi /etc/rsyslog.d/50-default.conf 命令来编辑日志配置文件。通过命令who来查看当前终端的pts号（假设是1）。然后在/etc/rsyslog.d/50-default.conf 文件中讲kern.* -/var/log/kern.log一行修改为kern.* -/dev/pts/1，然后执行命令sudo service rsyslog restart，即可在终端打印printk信息了。如果还不行，可以通过命令tail -n 10 /var/log/kern.log来查看输出信息。

ps：Ubuntu Host与Ubuntu Guest共享文件夹：安装Virtual Box的增强功能之后，选择自动挂载，即可在/media下看到共享文件夹（以sf_开头）。

2.2 实现基本的init

2.2.1 首先分配一块cache出来

kmem_cache_create函数的Kernel API介绍如下：`http://oss.org.cn/ossdocs/gnu_linux/kernel-api/r3758.html`

简要总结如下：

- 函数作用：创建slab缓存。
- 头文件：linux/slab.h
- 入参：
  - name：缓存名称，用于/proc/slabinfo 文件中唯一确认此缓存。如："my_cache"
  - size：使用此cache的对象的大小。如：sizeof(struct my_object)
  - align：对象对齐偏移量。如：__ alignof__struct my_object)
  - flags：SLAB标志。一般为：0

  - ctor：对象析构函数，将在cache建立新页面时被调用，一般为：NULL

- 返回值：
  - 成功时：返回指向此cache的指针，一般的返回指针类型为struct kmem_cache *
  - 失败时：返回NULL

这样，可以很简单的使用此函数了。此时的zsl_cache.c文件内容如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>

static struct kmem_cache *my_cache;
struct my_object{
    int should_add_later;
};

static int __init  zsl_init(void)
{
    printk(KERN_ALERT "hello, zsl.\n");
    my_cache = kmem_cache_create("my_cache",sizeof(struct
        my_object), __alignof__(struct my_object), 0, NULL );

    if(!my_cache){
        printk(KERN_EMERG "error: kmem_cache_create failed!\n");
        return -1;
    }
    printk(KERN_ALERT "kmem_cache_create succeed!\n");
    return 0;

}

static void __exit zsl_exit(void)
{
    printk(KERN_ALERT "goodbye, zsl.\n");
}

module_init(zsl_init);
module_exit(zsl_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Zhao Shulin");
```

```
34 MODULE_DESCRIPTION("Stay Focus!");
```

验证通过！

2.2.2 然后分配一个内存池

上一小节分配了一块cache，接着这个活，我们为了确保内存的成功分配，所以准备接着创建一个内存池。mempool_create 函数的Kernel API介绍如下：`https://www.kernel.org/doc/htmldocs/kernel-api/API-mempool-create.html`

简要总结如下：

- 函数作用：创建一个内存池。
- 头文件：linux/mempool.h
- 入参：
  - min_nr：为内存池分配的最小内存成员数量。如：1024
  - alloc_fn：内存分配函数。如：mempool.h文件中定义的mempool_alloc_slab函数
  - free_fn：内存释放函数。如：mempool.h 文件中定义的mempool_free_slab函数
  - pool_data：要管理的cache指针，本例中即为上一小节中的：my_cache
- 返回值：
  - 成功时：返回指向此pool的指针，一般的返回指针类型为mempool.h文件中定义的struct mempool_s *
  - 失败时：返回NULL

这样，可以很简单的使用此函数了。此时的zsl_cache.c文件内容如下：

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/slab.h>
4 #include <linux/mempool.h>
5
6 static struct kmem_cache *my_cache;
7 struct my_object{
```

```
8       int should_add_later;
9   };
10  static mempool_t *my_pool;
11
12  static int __init  zsl_init(void)
13  {
14      printk(KERN_ALERT "hello, zsl.\n");
15      my_cache = kmem_cache_create("my_cache",sizeof(struct
            my_object), __alignof__(struct my_object), 0, NULL );
16
17      if(!my_cache){
18          printk(KERN_EMERG "error: kmem_cache_create failed!\n");
19          return -1;
20      }
21      printk(KERN_ALERT "kmem_cache_create succeed!\n");
22
23      my_pool = mempool_create(1024, mempool_alloc_slab,
            mempool_free_slab, my_cache);
24      if(!my_pool){
25          printk(KERN_EMERG "error: mempool_create failed!\n");
26          kmem_cache_destroy(my_cache);
27          return -2;
28      }
29
30      printk(KERN_ALERT "mempool_create succeed!\n");
31
32      return 0;
33
34  }
35
36  static void __exit zsl_exit(void)
37  {
38      printk(KERN_ALERT "goodbye, zsl.\n");
39  }
40
41  module_init(zsl_init);
42  module_exit(zsl_exit);
43
44  MODULE_LICENSE("GPL");
45  MODULE_AUTHOR("Zhao Shulin");
46  MODULE_DESCRIPTION("Stay Focus!");
```

验证通过！

2.2.3 创建一个工作队列

因为工作队列的优势在于可以允许重新调度、睡眠，所以我们要初始化一个工作队列。工作队列（workqueue）可以将工作推后执行，即所谓的“下半部分”。create_singlethread_workqueue 函数的Kernel API介绍如下：http://lwn.net/Articles/81119/

简要总结如下：

- 函数作用：创建一个工作队列。
- 头文件：linux/workqueue.h
- 入参：name：描述此工作队列的字符串，比如：”my_workqueue”
- 返回值：
  - 成功时：返回指向此workqueue 的指针，一般的返回指针类型为workqueue.h 文件中定义的struct workqueue_struct *
  - 失败时：返回NULL

这样，可以很简单的使用此函数了。此时的zsl_cache.c文件内容如下：

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/slab.h>
4  #include <linux/mempool.h>
5  #include <linux/workqueue.h>
6
7  static struct workqueue_struct *my_workqueue;
8  static struct kmem_cache *my_cache;
9  struct my_object{
10     int should_add_later;
11 };
12 static mempool_t *my_pool;
13
14 static int __init  zsl_init(void)
15 {
16     printk(KERN_ALERT "hello, zsl.\n");
17     my_cache = kmem_cache_create("my_cache",sizeof(struct
           my_object), __alignof__(struct my_object), 0, NULL );
18
19     if(!my_cache){
20         printk(KERN_EMERG "error: kmem_cache_create failed!\n");
21         return -1;
```

```
22      }
23      printk(KERN_ALERT "kmem_cache_create succeed!\n");
24
25      my_pool = mempool_create(1024, mempool_alloc_slab,
            mempool_free_slab, my_cache);
26      if(!my_pool){
27          printk(KERN_EMERG "error: mempool_create failed!\n");
28          kmem_cache_destroy(my_cache);
29          return -2;
30      }
31          printk(KERN_ALERT "mempool_create succeed!\n");
32
33      my_workqueue = create_singlethread_workqueue("zsl_kcache");
34      if(!my_workqueue){
35          printk(KERN_EMERG "error: create_singlethread_workqueue
                failed!\n");
36          return -3;
37      }
38
39      printk(KERN_ALERT "create_singlethread_workqueue succeed!\n"
            );
40
41      return 0;
42
43  }
44
45  static void __exit zsl_exit(void)
46  {
47      printk(KERN_ALERT "goodbye, zsl.\n");
48  }
49
50  module_init(zsl_init);
51  module_exit(zsl_exit);
52
53  MODULE_LICENSE("GPL");
54  MODULE_AUTHOR("Zhao Shulin");
55  MODULE_DESCRIPTION("Stay Focus!");
```

验证通过！

2.2.4 新建一个工作work

INIT_WORK(_work, _func)可以新建一个work，该work在创建好之后，一般可以通过queue_work(workqueue, &work)函数来把该work放到上一小节中创建的workqueue中工作（即运行自定义的_func函数）。

INIT_WORK(_work, _func)简要总结如下：

- 函数作用：创建一个工作。
- 头文件：linux/workqueue.h
- 入参：
  - _work：该工作的名称，一般为workqueue.h中定义的struct work_struct *
  - _func：用户自定义的函数
- 返回值：无

此时的zsl_cache.c文件内容如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/mempool.h>
#include <linux/workqueue.h>

static struct work_struct my_work;
static struct workqueue_struct *my_workqueue;
static struct kmem_cache *my_cache;
struct my_object{
    int should_add_later;
};
static mempool_t *my_pool;

static void do_work()
{
    printk(KERN_ALERT "This is do_work!\n");
}

static int __init  zsl_init(void)
{
    printk(KERN_ALERT "hello, zsl.\n");
    my_cache = kmem_cache_create("my_cache",sizeof(struct
        my_object), __alignof__(struct my_object), 0, NULL );
```

```
24
25      if(!my_cache){
26          printk(KERN_EMERG "error: kmem_cache_create failed!\n");
27          return -1;
28      }
29      printk(KERN_ALERT "kmem_cache_create succeed!\n");
30
31      my_pool = mempool_create(1024, mempool_alloc_slab,
            mempool_free_slab, my_cache);
32      if(!my_pool){
33          printk(KERN_EMERG "error: mempool_create failed!\n");
34          kmem_cache_destroy(my_cache);
35          return -2;
36      }
37          printk(KERN_ALERT "mempool_create succeed!\n");
38
39      my_workqueue = create_singlethread_workqueue("zsl_kcache");
40      if(!my_workqueue){
41          printk(KERN_EMERG "error: create_singlethread_workqueue
                failed!\n");
42          return -3;
43      }
44
45      printk(KERN_ALERT "create_singlethread_workqueue succeed!\n"
            );
46
47      INIT_WORK(&my_work, do_work);
48      return 0;
49
50  }
51
52  static void __exit zsl_exit(void)
53  {
54      printk(KERN_ALERT "goodbye, zsl.\n");
55  }
56
57  module_init(zsl_init);
58  module_exit(zsl_exit);
59
60  MODULE_LICENSE("GPL");
61  MODULE_AUTHOR("Zhao Shulin");
62  MODULE_DESCRIPTION("Stay Focus!");
```

验证成功。

至此，最基本的init已经完成。下一节会实现device mapper。

2.3 添加device mapper

2.3.1 Device Mapper 的基础知识

假如一台主机插入了多块硬盘，单块硬盘的容量和性能都是有限的，如果能将多块硬盘组合一个逻辑的整体，对于这台主机来讲，就实现了最简单意义上的“云存储”。所以，Linux内核中出现了Device Mapper。简单来讲，Device Mapper是一种组合多个块设备变成一个逻辑块设备的机制。

Device Mapper的设计实现主要分为三层，如图1所示：

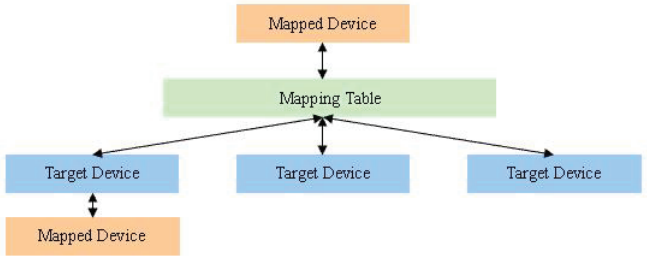

Figure 1: Device Mapper的设计实现。

- Mapped Device：映射之后的逻辑设备，对应于内核中的struct mapped_device
- Mapping Table：映射规则表，对应于内核中的struct dm_table
- Target Device：底层的实际设备（注：既可以是物理设备，也可以是低一层的Device Mapper映射出的逻辑设备。），对应于内核中的struct dm_target

再看两个内核里面的两个重要内容：

- struct target_type：自定义一种target type的类型

- dm_register_target：此函数使用上面自定义的结构体注册一个新的target type

注意：上述内容需要头文件linux/device-mapper.h。

此时的zsl_cache.c文件内容如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/mempool.h>
#include <linux/workqueue.h>
#include <linux/device-mapper.h>

static struct work_struct my_work;
static struct workqueue_struct *my_workqueue;
static struct kmem_cache *my_cache;
struct my_object{
    int should_add_later;
};
static mempool_t *my_pool;
static struct target_type my_cache_target = {
    .name = "my_cache",
    .version = {0,0,0},
    .module = THIS_MODULE,
    .ctr = NULL,
    .dtr = NULL,
// should add something later...

};

static void do_work(void)
{
    printk(KERN_ALERT "This is do_work!\n");
}

static int __init zsl_init(void)
{
    printk(KERN_ALERT "hello, zsl.\n");
    my_cache = kmem_cache_create("my_cache",sizeof(struct
        my_object), __alignof__(struct my_object), 0, NULL );

    if(!my_cache){
        printk(KERN_EMERG "error: kmem_cache_create failed!\n");
        return -1;
    }
    printk(KERN_ALERT "kmem_cache_create succeed!\n");
```

```
40
41      my_pool = mempool_create(1024, mempool_alloc_slab,
            mempool_free_slab, my_cache);
42      if(!my_pool){
43          printk(KERN_EMERG "error: mempool_create failed!\n");
44          kmem_cache_destroy(my_cache);
45          return -2;
46      }
47          printk(KERN_ALERT "mempool_create succeed!\n");
48
49      my_workqueue = create_singlethread_workqueue("zsl_kcache");
50      if(!my_workqueue){
51          printk(KERN_EMERG "error: create_singlethread_workqueue
                failed!\n");
52          return -3;
53      }
54
55      printk(KERN_ALERT "create_singlethread_workqueue succeed!\n"
            );
56
57      INIT_WORK(&my_work, do_work);
58
59      if(dm_register_target(&my_cache_target)<0)
60          {
61              printk(KERN_EMERG "error: dm_register_target failed
                    !\n");
62              return -4;
63          }
64      else
65          printk(KERN_ALERT "dm_register_target succeed!\n");
66
67      return 0;
68
69  }
70
71  static void __exit zsl_exit(void)
72  {
73      dm_unregister_target(&my_cache_target);
74      printk(KERN_ALERT "dm_unregister_target succeed!\n");
75
76      mempool_destroy(my_pool);
77      printk(KERN_ALERT "mempool_destroy succeed!\n");
78
79      kmem_cache_destroy(my_cache);
80      printk(KERN_ALERT "kmem_cache_destroy succeed!\n");
```

```
81
82      my_pool = NULL;
83      my_cache = NULL;
84
85      destroy_workqueue(my_workqueue);
86      printk(KERN_ALERT "destroy_workqueue succeed!\n");
87
88
89      printk(KERN_ALERT "goodbye, zsl.\n");
90  }
91
92  module_init(zsl_init);
93  module_exit(zsl_exit);
94
95  MODULE_LICENSE("GPL");
96  MODULE_AUTHOR("Zhao Shulin");
97  MODULE_DESCRIPTION("Stay Focus!");
```

验证成功。

至此，init和exit框架已经搭建完毕了。