MICROOH 麦可网

Android-从程序员到架构师之路

出品人：Sundy
讲师：高焕堂（台湾）

http://www.microoh.com

E04_b

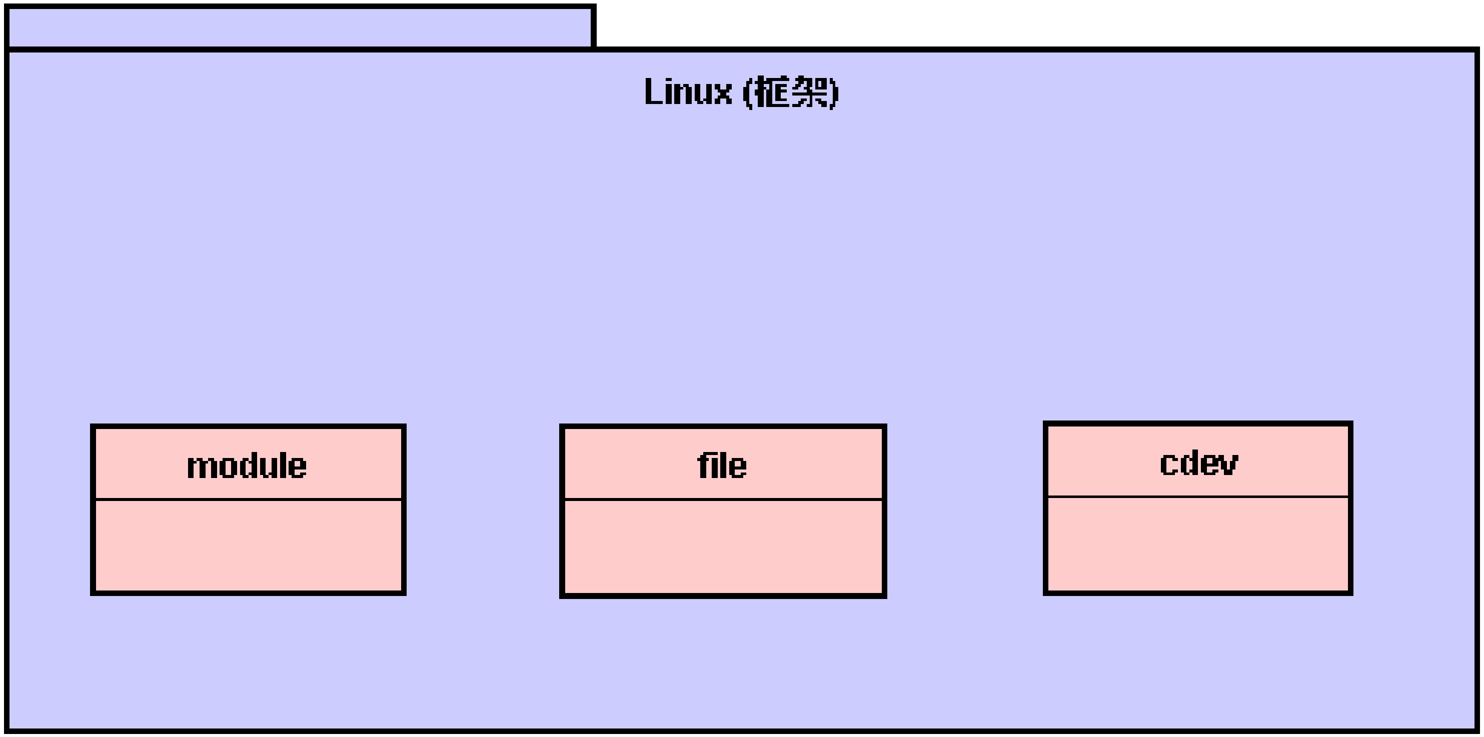
从框架看HAL和Linux驱动开发(b)

By 高焕堂

2、Linux驱动框架的函数表

- 在Linux中，所有的设备(device)都被视为一个特殊档案(file)， 称为装置文件(device file)，每个档案都有自己特殊的编号和型态，定义于Linux的/dev目录区里。
- 所以，在Linux里，共有三个最主要的概念(concept)：

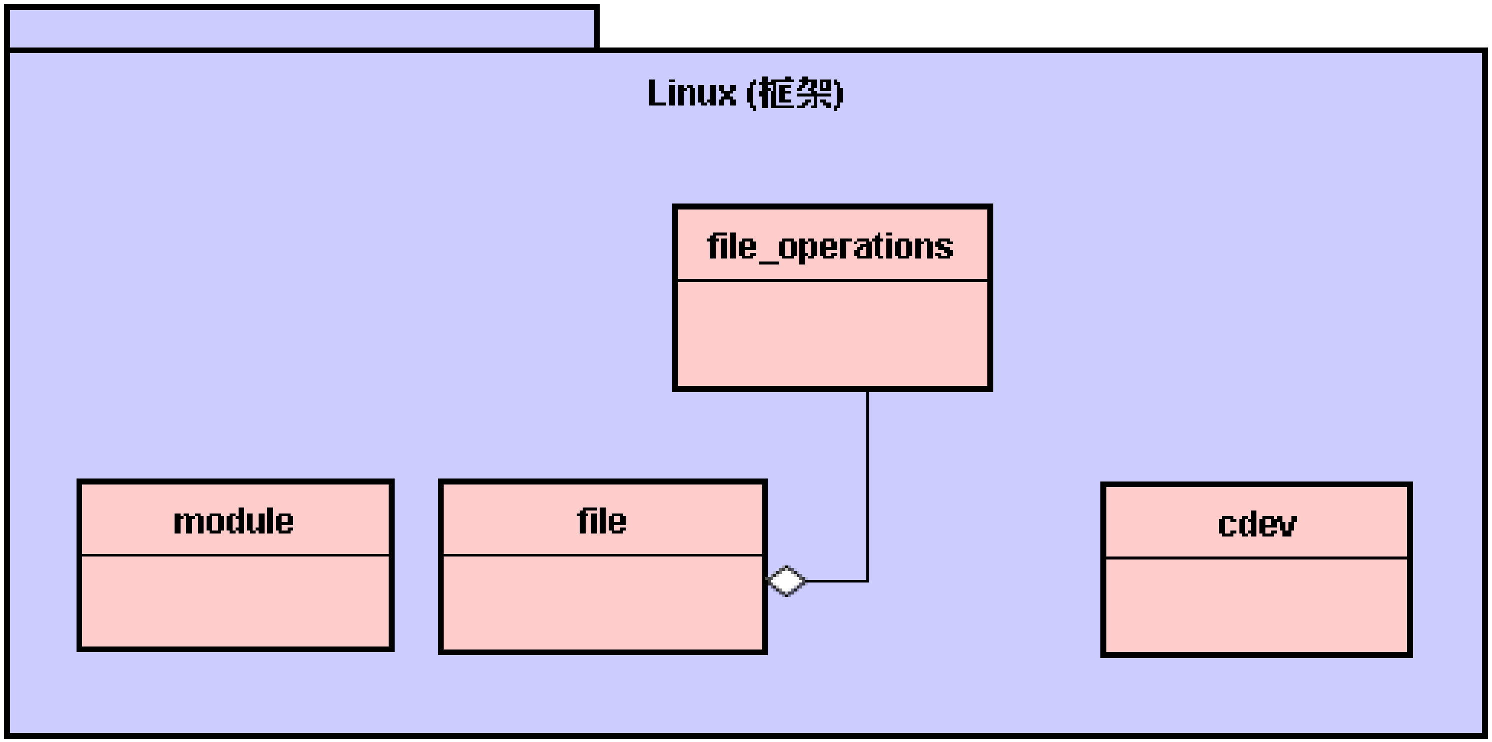
- 设备，表现于struct cdev结构，相当于是cdev类。
- 档案，表现于struct file结构，相当于是file类。
- 模块，表现于struct module结构，相当于是module类。



Linux (框架)
module
file
cdev

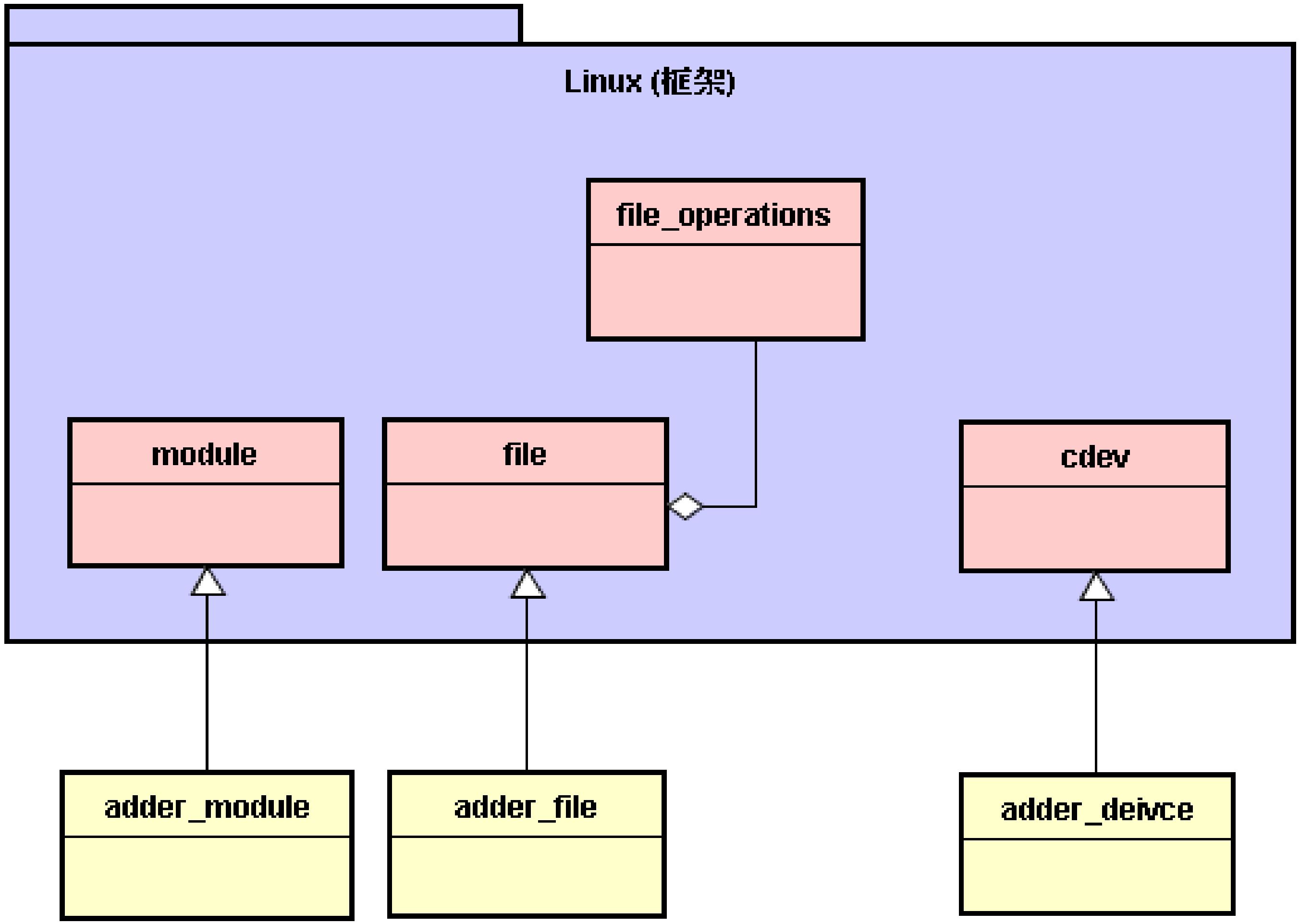
函数表(接口)

- 将struct file里的一部分函数定义独立出来，成为函数表(function table)，也就是接口(interface)了。
- 例如，从struct file独立出来，成为struct file_operations：

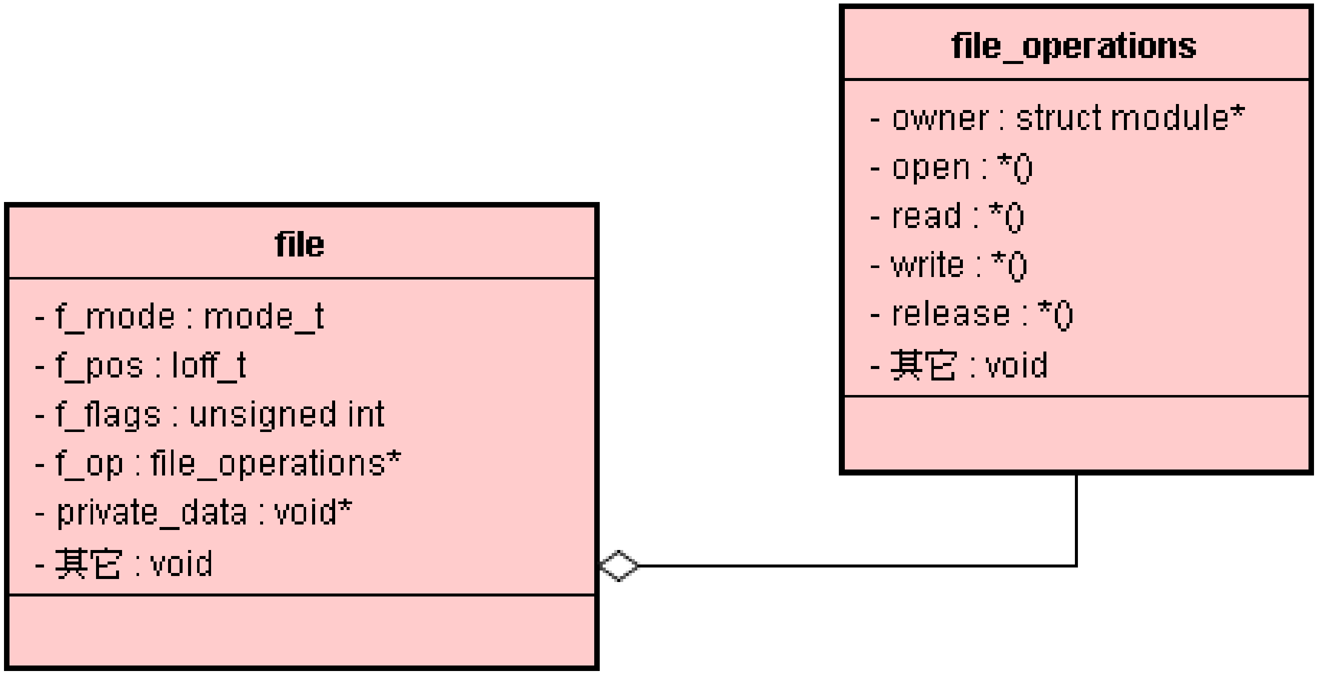

Linux (框架)
file_operations
module
file
cdev

- 这struct module、struct file和struct cdev就成为Linux衔接驱动模块(Stub)的主角了；也就相当于基类的角色。

- 基于这些基础的struct结构(即基类)，我们就能加以扩充出子结构(即子类)了。


Linux (框架)
file_operations
module
file
cdev
adder_module
adder_file
adder_deivce

file_operations函数表



- Linux框架定义了struct file：

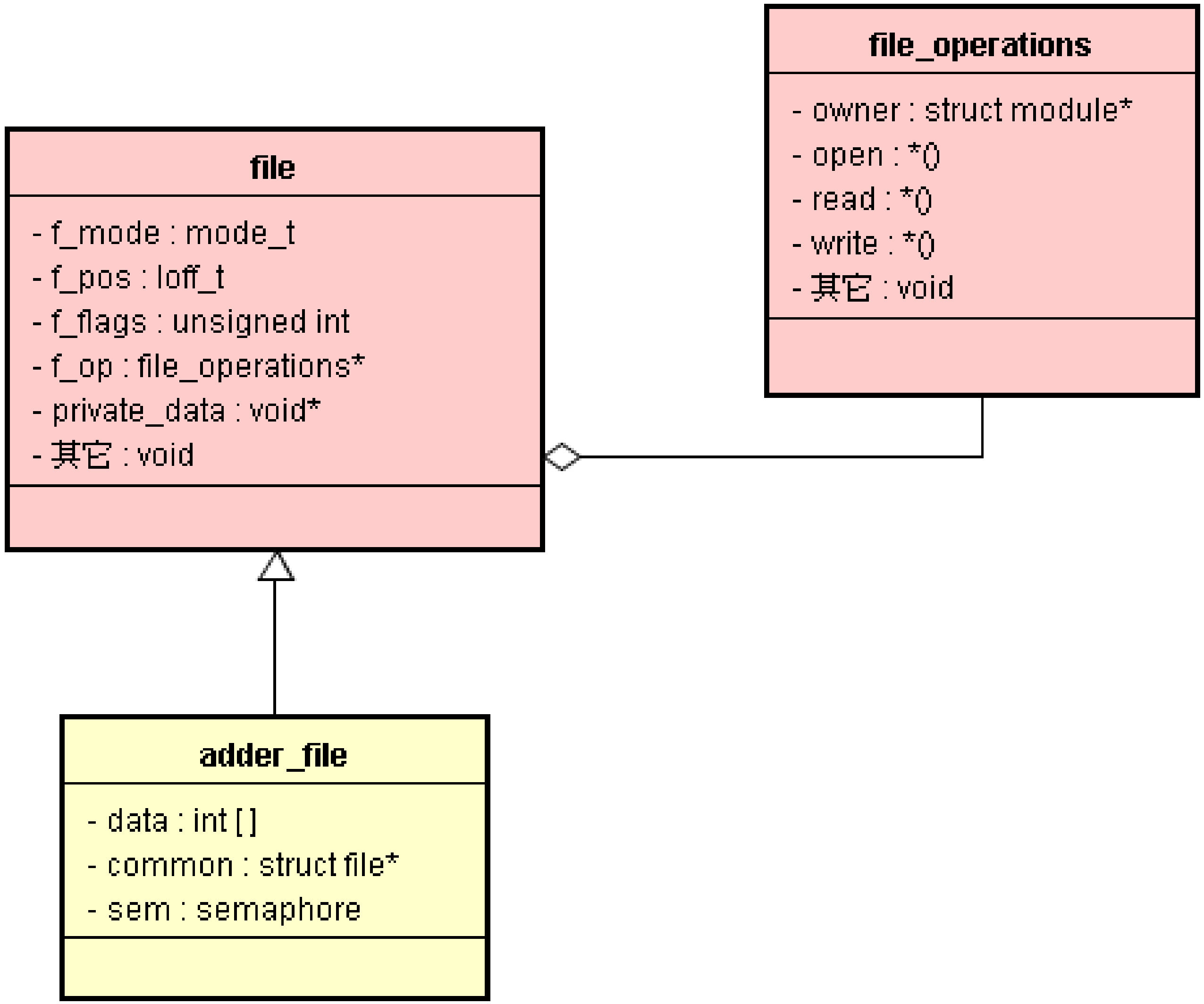
```
struct file {
        mode_t f_mode;
        loff_t f_pos;
        unsigned int f_flags;
        struct file_operations *f_op;
        void *private_data;
        struct dentry *f_dentry;
};
```

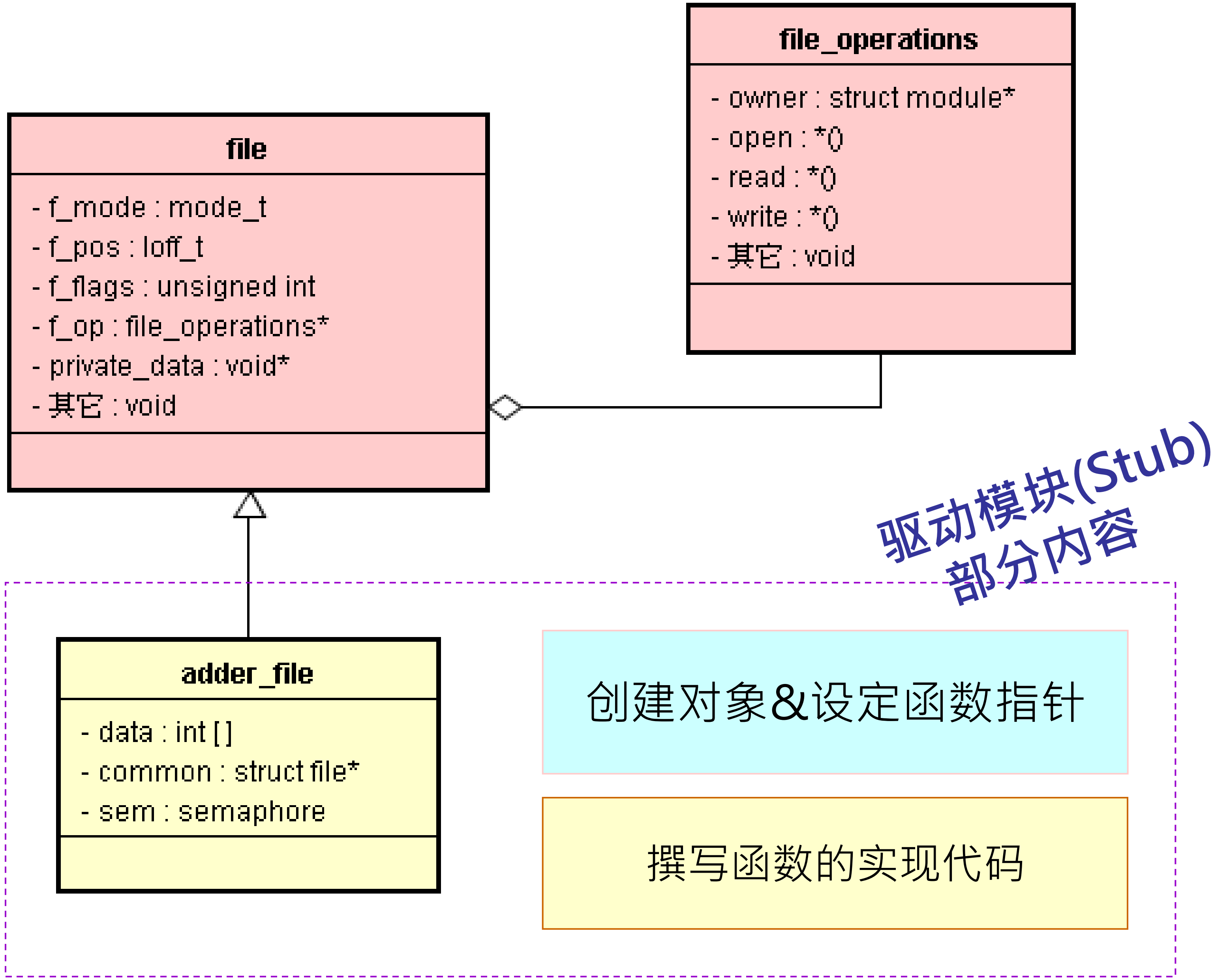
- Linux框架也定义了struct file_operations函数表：

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
```

```
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
                                  unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                                  unsigned long, loff_t *);
    // ……..
};
```

扩充struct file的定义




file
- f_mode : mode_t
- f_pos : loff_t
- f_flags : unsigned int
- f_op : file_operations*
- private_data : void*
- 其它 : void
file_operations
- owner : struct module*
- open : *()
- read : *()
- write : *()
- 其它 : void
驱动模块(Stub)
部分内容
adder_file
- data : int []
- common : struct file*
- sem : semaphore
创建对象&设定函数指针
撰写函数的实现代码

<定义子类：adder_file>

/ adder_file   */*

```
struct adder_file {
Int data[2]
struct file* common;
struct semaphore sem;
};
```

<诞生adder_file对象>

```
struct adder_file add_file;
```

<诞生file_operations对象>

```
struct file_operations fop={
        .owner = THIS_MODULE,
        .open = add_open,
        .read = add_read,
        .write = add_write,
        .release = add_release,
};
```

- 这2个对象是以静态(static)方式宣告的，会在驱动模块加载时刻(loading time)诞生出来。

- 这些函数指针(Function Pointer)是用来指向C函数的实现代码。
- 于是，在Linux驱动模块里，撰写C函数的实现代码，如下：

<撰写函数>

```
int add_open(struct inode *inode, struct file *filp){
        filp->private_data = &add_file;
        add_file.common = filp;
        return 0;
}
ssize_t add_access(int access_dir, struct file *filp, char __user *buf, size_t count){
        int *data = filp->private_data->data;
        ssize_t retval = 0;
        if(down_interruptible(&device->sem))
      return -ERESTARTSYS;
```

```
    if(access_dir == 0) {
        int sum = data[0] + data[1];
        if(count != sizeof(int))  goto out;
        retval = copy_to_user(buf, &sum, sizeof(int));
        }
    else {
        if(count != sizeof(int) * 2)  goto out;
        retval = copy_from_user(data, buf, count);
        }
    if(retval) {
        retval = -EFAULT;
        goto out;
        }
out:
    up(&add_file.sem);
    return retval;
}
```

```
  int add_read(struct file *filp,  char __user *buf, size_t count, loff_t
*f_pos) {
          return add_access(0, filp, buf, count);
  }

  int add_write(struct file *filp, const char __user *buf, size_t count,
loff_t *f_pos){
          return add_access(1, filp, (char __user *)buf, count);
  }

  int add_release(struct inode *inode, struct file *filp){
          return 0;
  }
```

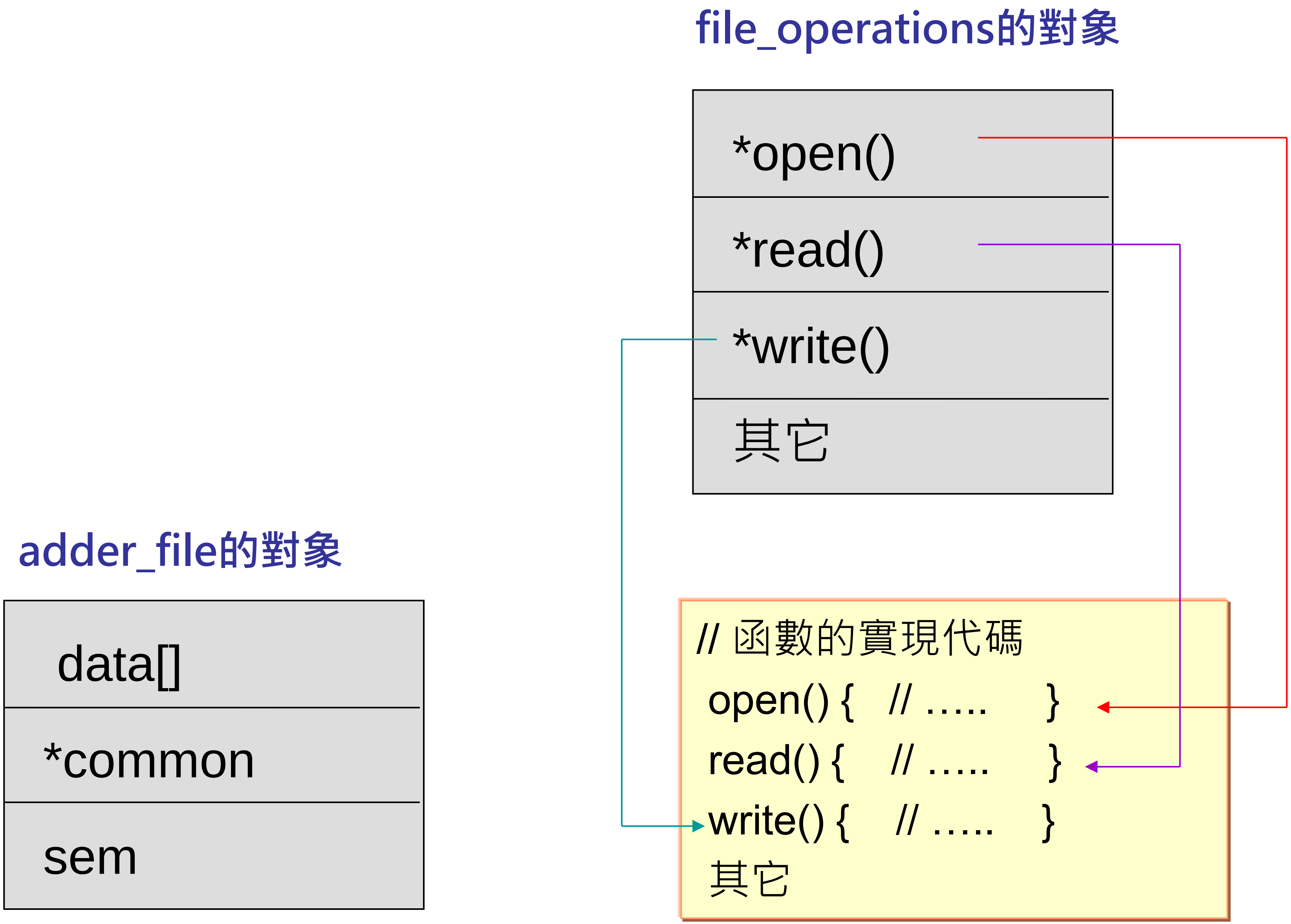
At run-time

- 这2个对象是以静态(static)方式宣告的。

```
struct adder_file add_file;
```

```
struct file_operations fop={
        .owner = THIS_MODULE,
        .open = add_open,
        .read = add_read,
        .write = add_write,
        .release = add_release,
};
```

- 当驱动模块被载入Linux内核时，就诞生这2个对象，并让file_operations的函数指针指向add_open()、add_read()等函数的实现代码。


file_operations的對象
*open()
*read()
*write()
其它
adder_file的對象
data[]
*common
sem
// 函數的實現代碼
open() { // ….. }
read() { // ….. }
write() { // ….. }
其它

- 刚才的对象，是采静态(static)方式来诞生的。在驱动被加载时刻就诞生了。
- 也可以采取动态(dynamic)方式来诞生。亦即，由module_init()诞生它们 。

~ Continued ~