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_c

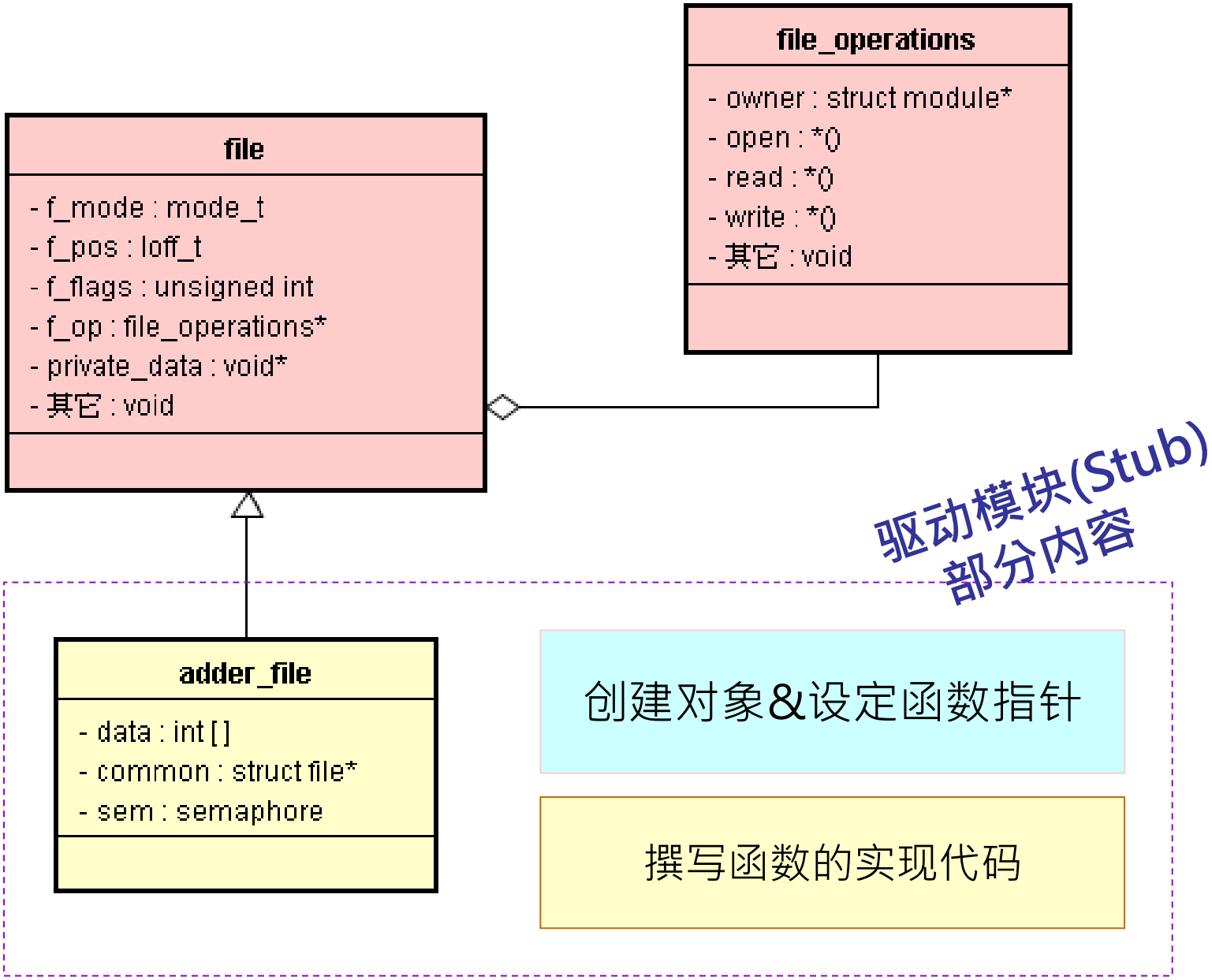
从框架看HAL和Linux驱动开发(c)

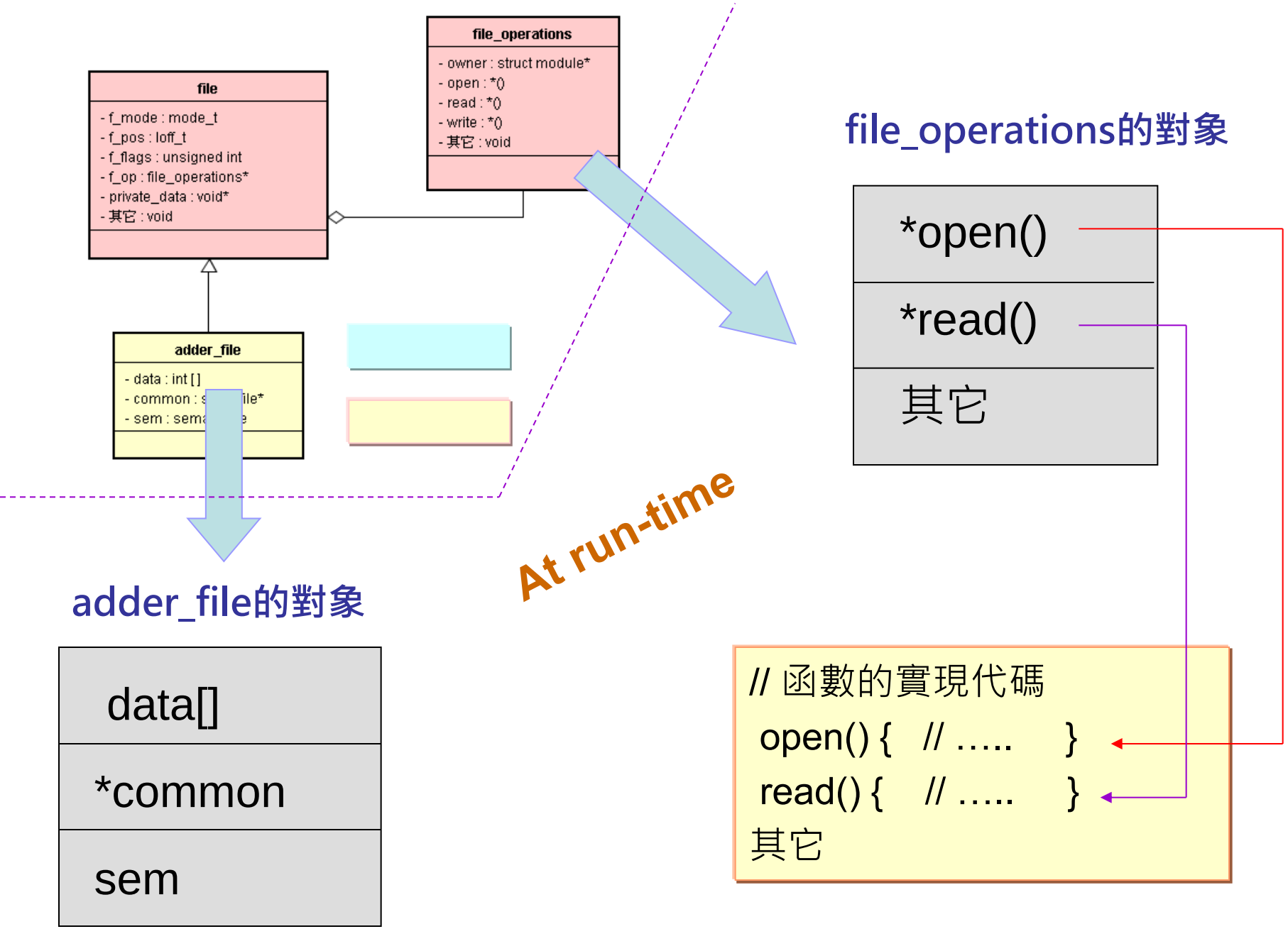
By 高煥堂

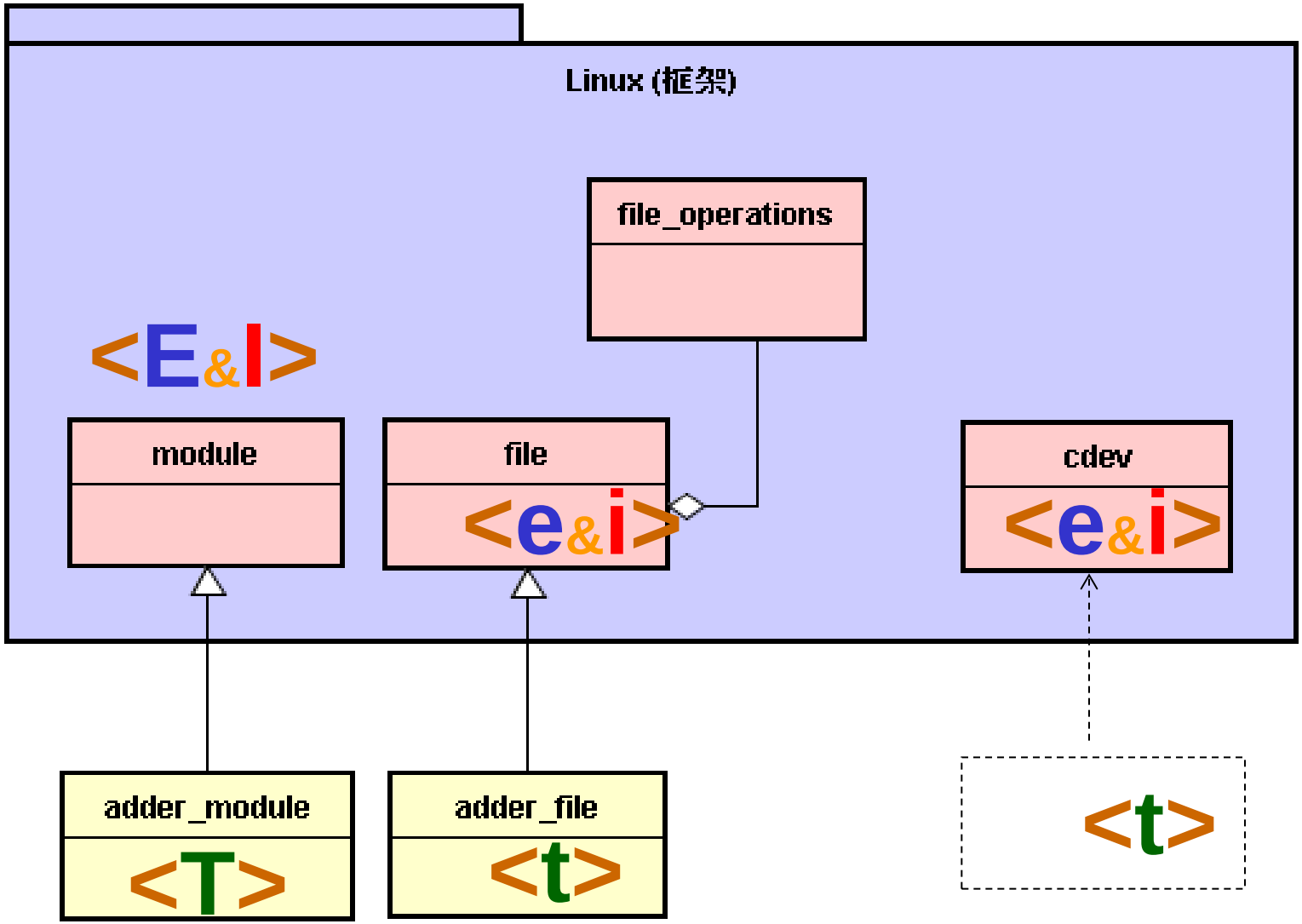
3、活用工厂EIT造形

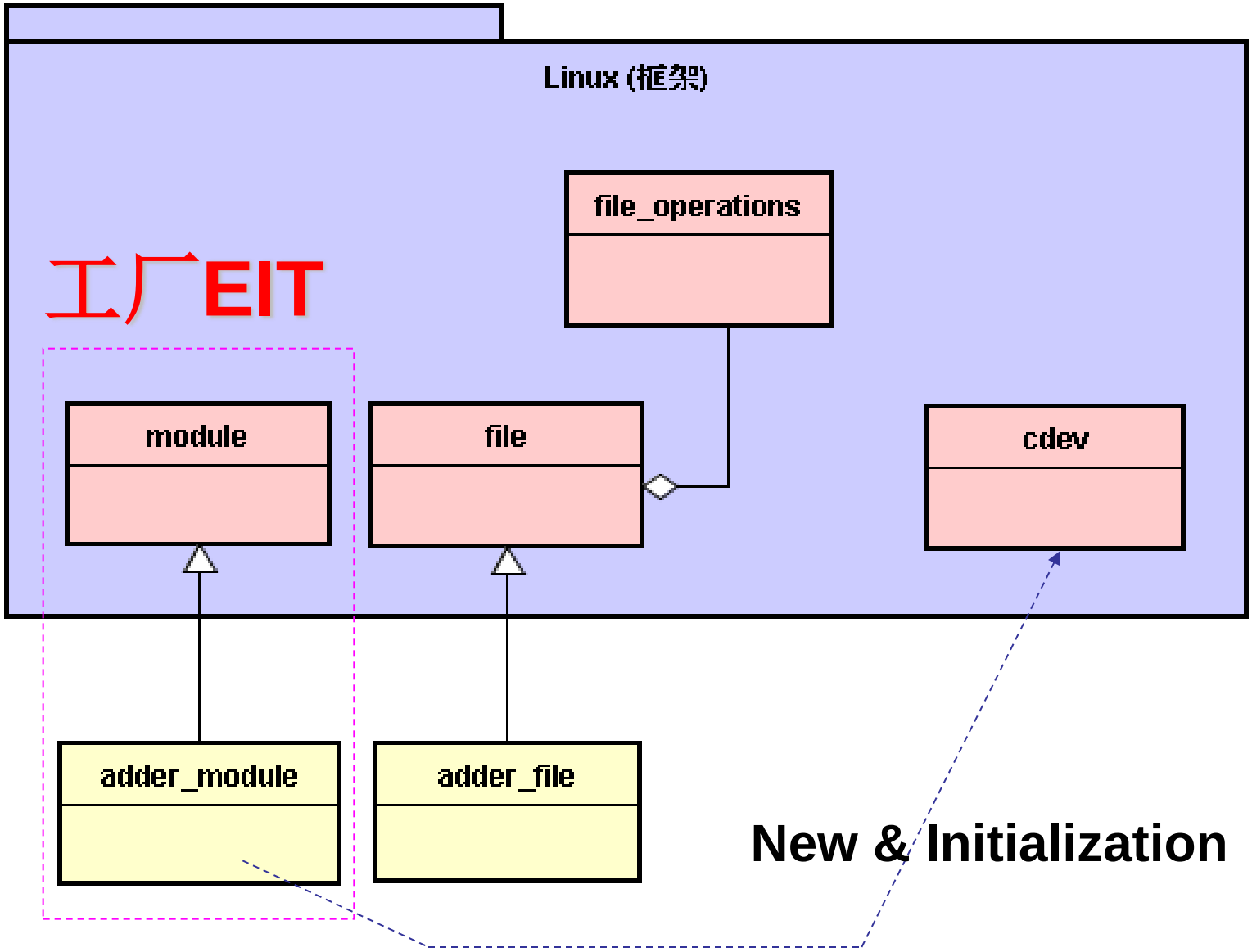
撰写adder_module代码

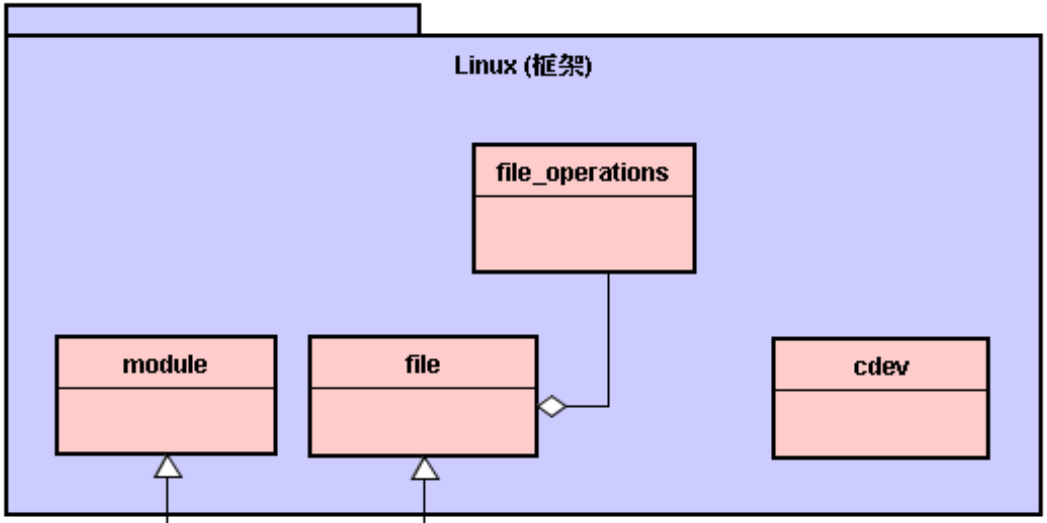
前言


file_operations
- owner : struct module*
- open : *()
- read : *()
- write : *()
- 其它 : void
file
- f_mode : mode_t
- f_pos : loff_t
- f_flags : unsigned int
- f_op : file_operations*
- private_data : void*
- 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


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
adder_file
- data : int [ ]
- common : s ile*
- sem : sema e
file_operations的對象
*open()
*read()
其它
At run-time
adder_file的對象
data[]
*common
sem
// 函數的實現代碼
open() { // ..... }
read() { // ..... }
其它


Linux (框架)
file_operations
<E&I>
module
file
<e&i>
cdev
<e&i>
adder_module
<T>
adder_file
<t>
<t>


Linux (框架)
file_operations
工厂EIT
module
file
cdev
adder_module
adder_file
New & Initialization



驱动模块(*.ko)

adder_module

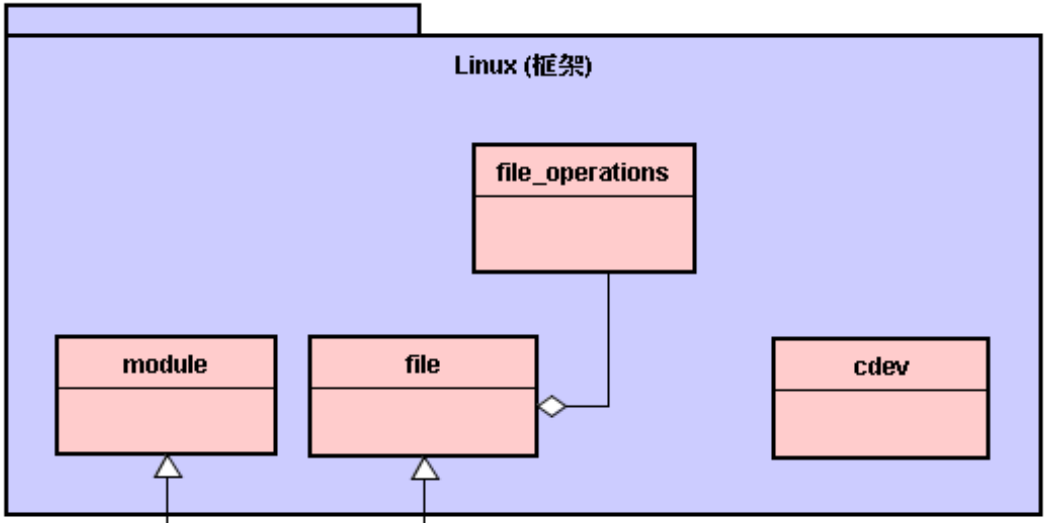
adder_file

创建对象&设定函数指针

创建对象

撰写函数的实现代码

函数代码(起始設定)



驱动模块(*.ko)

adder_module

adder_file

创建对象&设定函数指针

创建对象

撰写函数的实现代码

函数代码(起始設定)

- add_module子类必须实作module_init()和module_exit()函数。

/ adder_module */*

```
#define ADD_MAJOR    48
#define ADD_MINOR    0
struct cdev add_device;
int add_mod_init(void) {
        int result;
        dev_t devno = MKDEV(ADD_MAJOR, ADD_MINOR);
        result = register_chrdev_region(devno, 1, "androidin");
        if(result < 0)  return result;
        init_MUTEX(&adder.sem);
        cdev_init(&add_device, &fop);
        my_cdev.owner = THIS_MODULE;
```

```
        result = cdev_add(&add_device, devno, 1);
        if(result)          goto fail;
        return 0;
fail:
        add_cleanup();
        return result;
}
void add_mod_cleanup(void) {
        dev_t devno = MKDEV(ADD_MAJOR, ADD_MINOR);
        cdev_del(&add_device);
        unregister_chrdev_region(devno, 1);
}
module_init(add_mod_init);
module_exit(add_mod_cleanup);
MODULE_LICENSE("GPL");
```

- module_init()函数负责初期化(initialization)动作。首先呼叫cdev类别的cdev_init()函数来进行初始化动作，如下指令：

```
cdev_init(add_device, &fop);
```

- 这让add_device对象指向fop对象。此外，也让Linux所诞生的struct file对象(内含f_op指针)指向fop对象。

- 接着，执行如下指令：

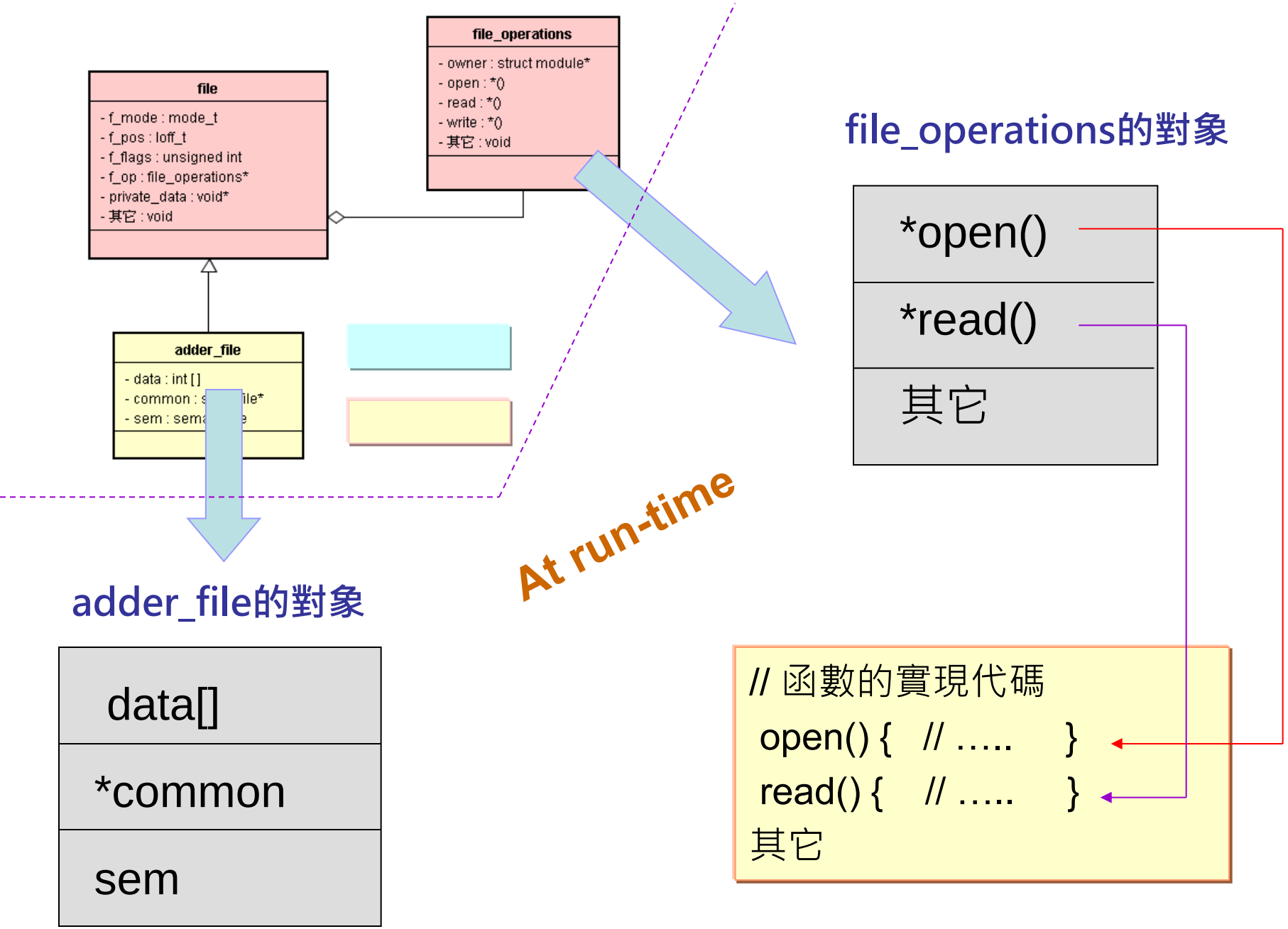
```
cdev_add(&add_device, devno, 1);
```

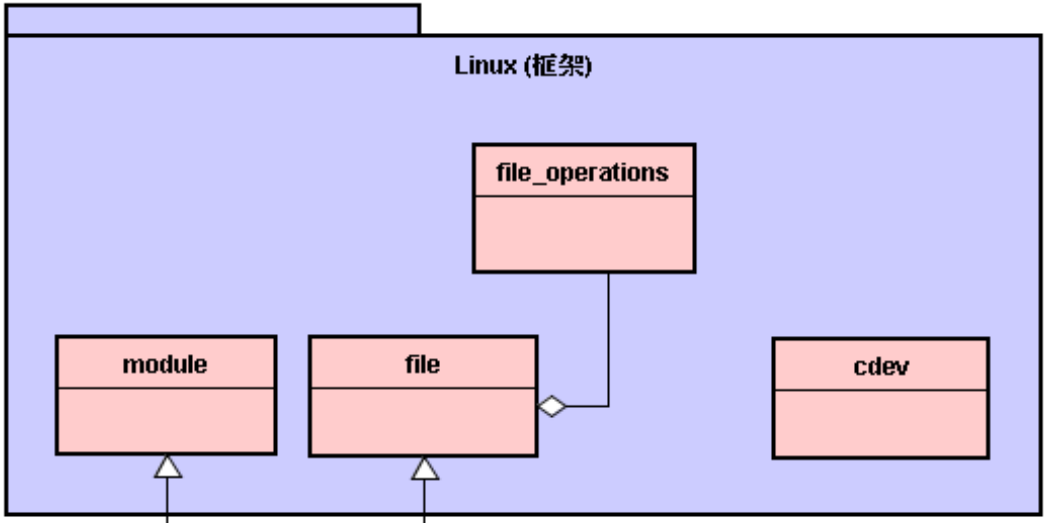
- 这呼叫cdev_add()函数把cdev 对象指针存入到Linux内核的cdev_map里。让Linux内核可以透过file_operations接口而呼叫到cdev类里的函数。

- 对系统而言，当设备驱动成功调用了cdev_add()之后，就意味着一个设备对象已经加入到了系统，让系统就可以找到它。对用户态的应用程序而言，调用cdev_add()之后，就已经可以通过System Call呼叫驱动程序了。

- 现在已经将adder_file和adder_module撰写完毕了。经过编译&连结成为*.ko模块之后，就能将此驱动模块挂载到Linux内核里了。

At run-time


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
adder_file
- data : int [ ]
- common : s ile*
- sem : sema e
file_operations的對象
*open()
*read()
其它
At run-time
adder_file的對象
data[]
*common
sem
// 函數的實現代碼
open() { // ..... }
read() { // ..... }
其它



驱动模块(*.ko)

adder_module

adder_file

创建对象&设定函数指针

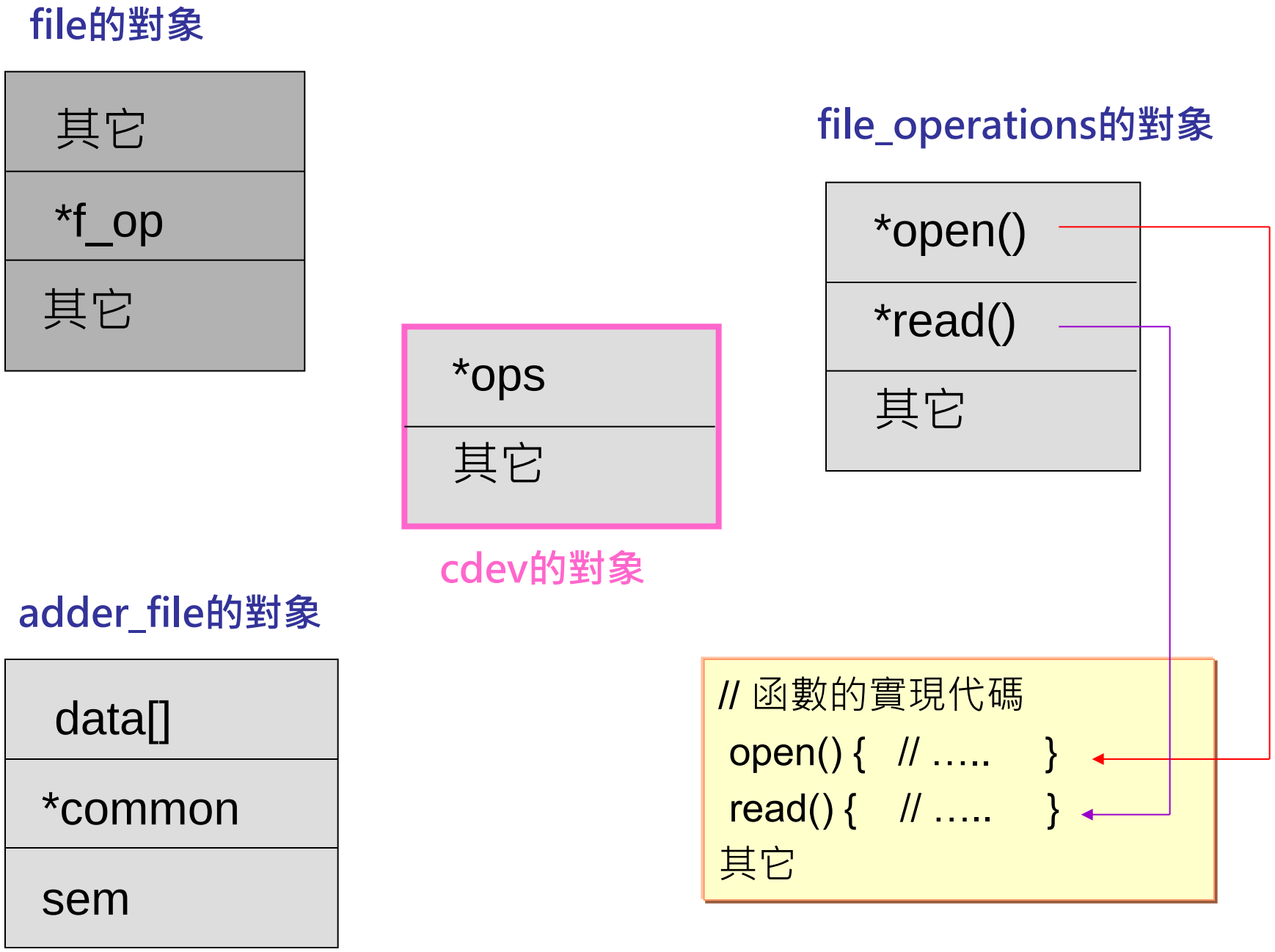
创建对象

撰写函数的实现代码

函数代码(起始設定)

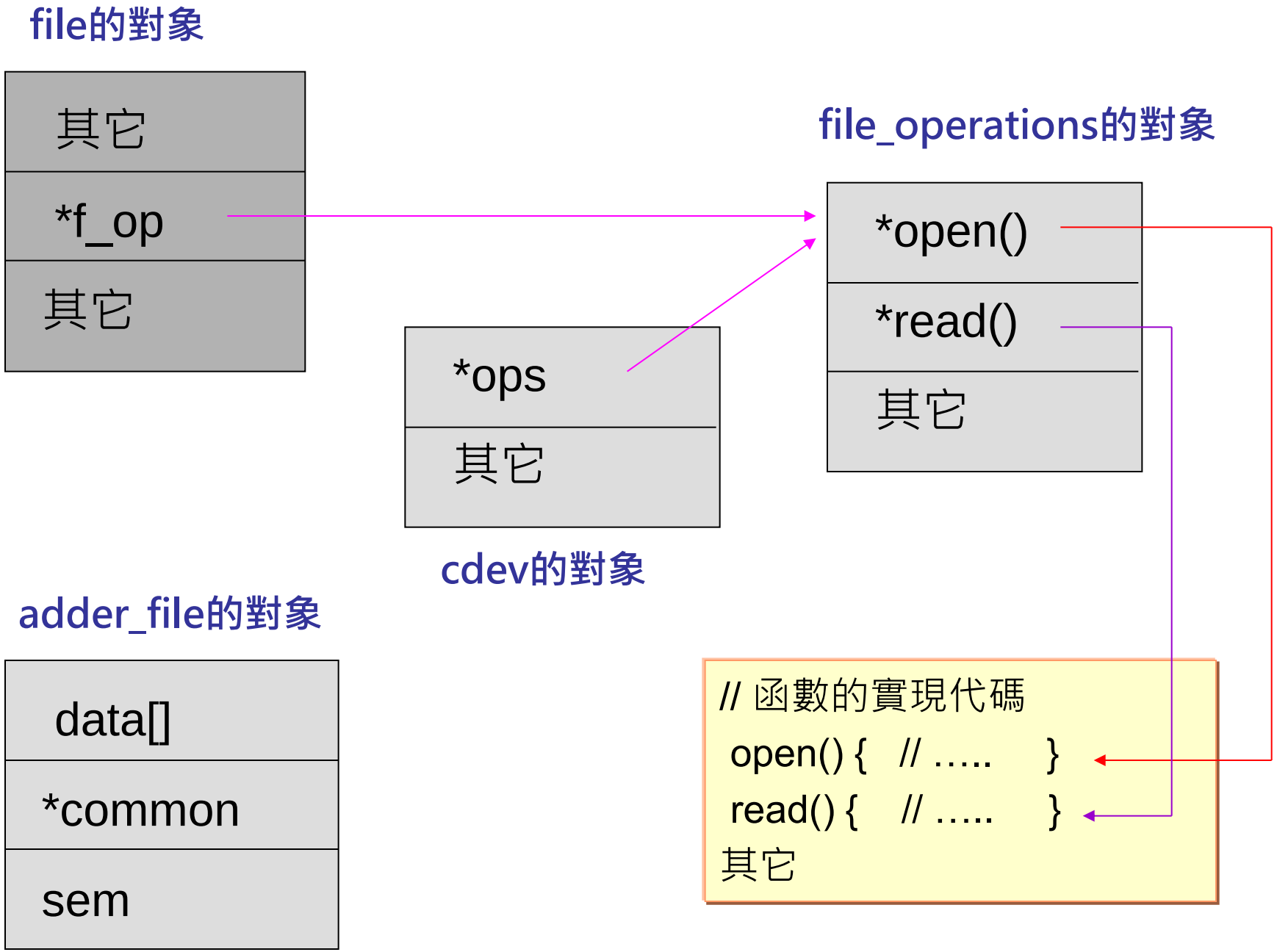
- 这个对象是以静态(static)方式宣告的。

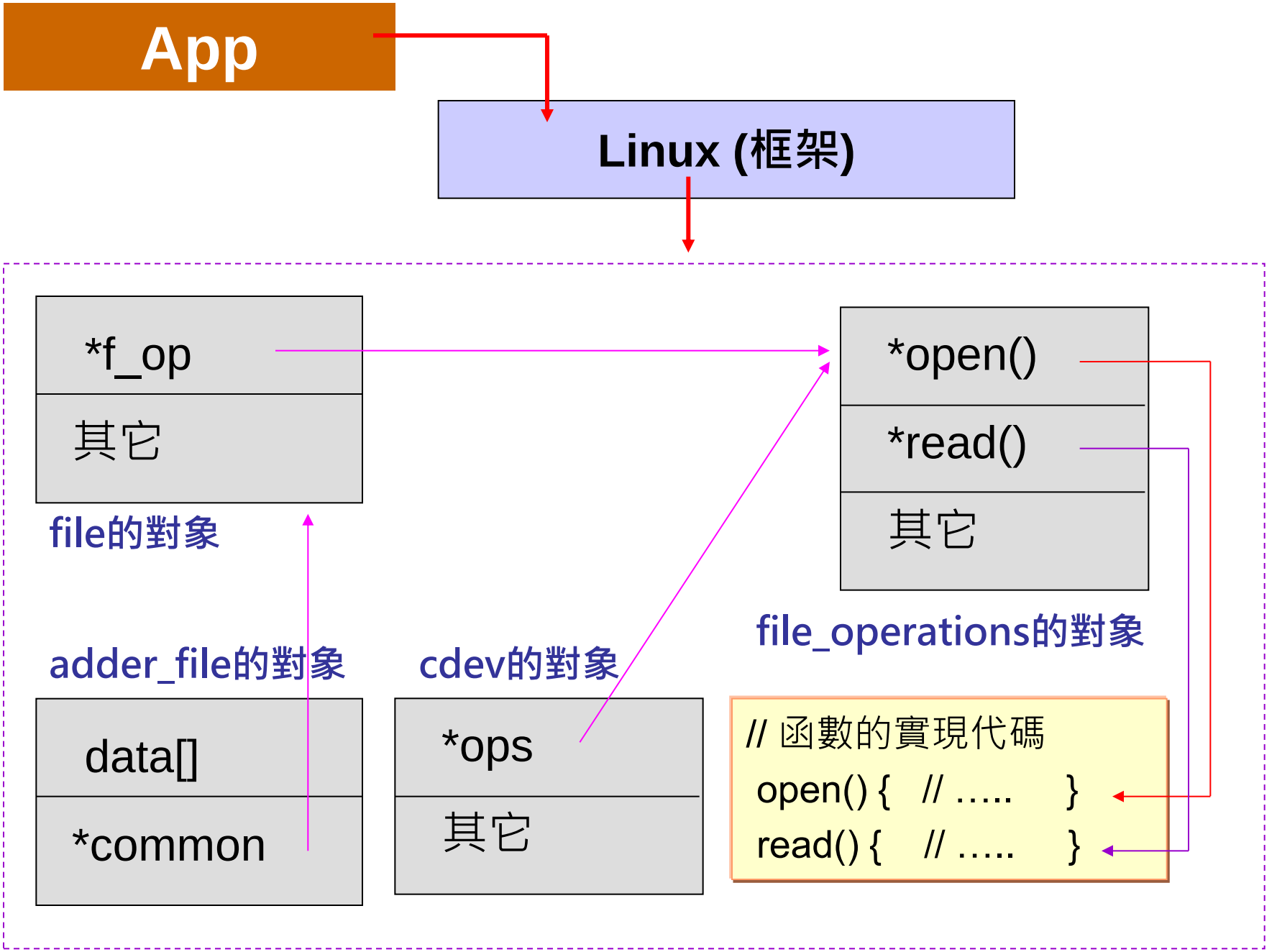
```
struct cdev add_device;
```


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

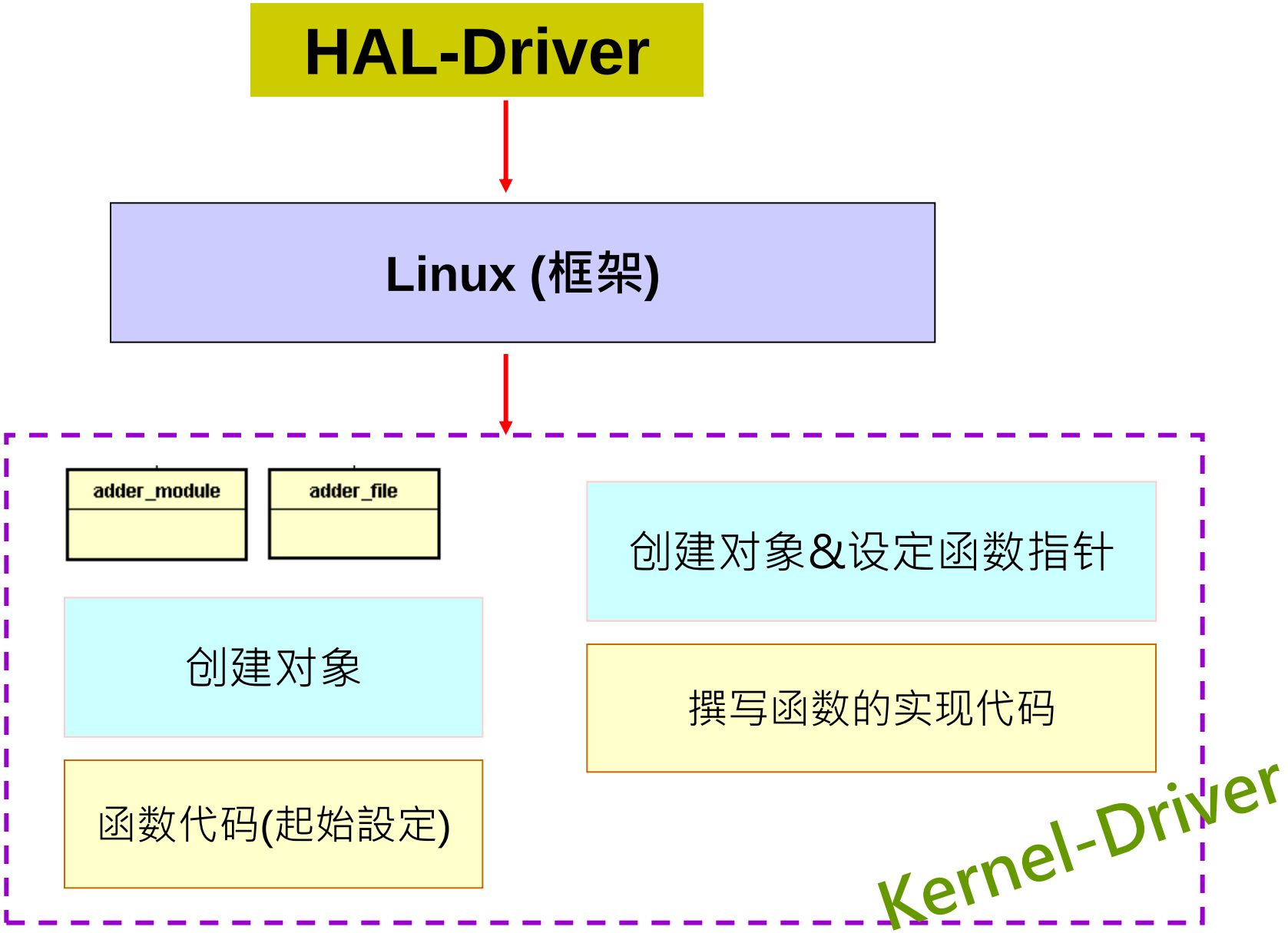
```
cdev_init(&add_device, &fop);
```

- 这让add_device对象指向fop对象。此外，也让Linux所诞生的strut file对象(内含f_op指针)指向fop对象。


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


App
Linux (框架)
*f_op
其它
file的對象
*open()
*read()
其它
file_operations的對象
adder_file的對象
data[]
*common
cdev的對象
*ops
其它
// 函數的實現代碼
open() { // ..... }
read() { // ..... }

Kernel-Driver模块(代码层级)


HAL-Driver
Linux (框架)
adder_module
adder_file
创建对象&设定函数指针
创建对象
撰写函数的实现代码
函数代码(起始設定)
Kernel-Driver

~ Continued ~