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

E02_c

HAL框架与Stub开发 (c)

By 高煥堂

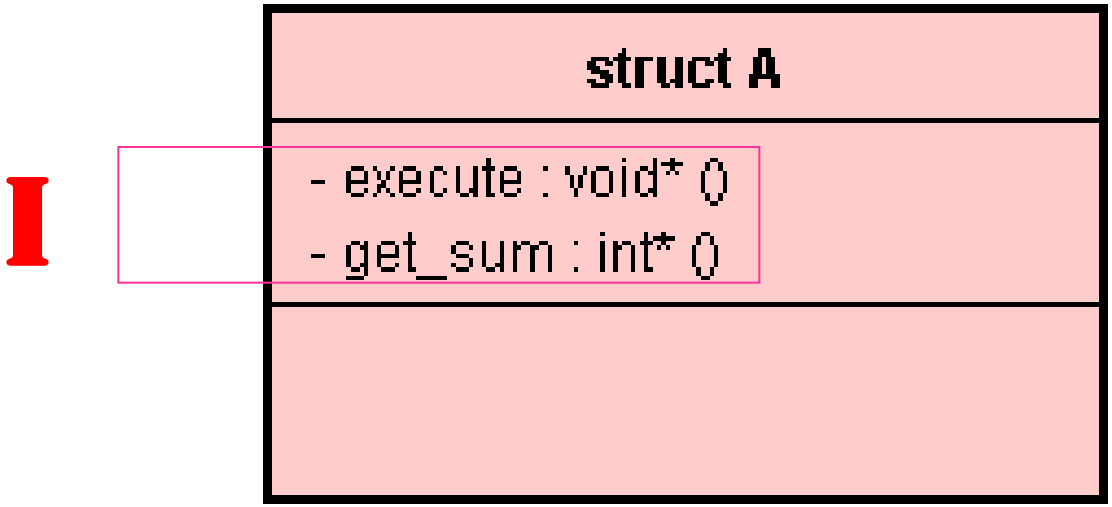
2、认识HAL的架构

前言

- 一个C的struct定义

```
struct A {
    void (*execute)();
     int (*get_sum)();
};
```

- 函数指针(function pointer)指有函数定义，而没有代码；其相当于抽象函数(abstract function)。



- 基于上述的<E&I>定义，我们如何写<T>呢?

创建对象&
设定函数指针

撰写<I>的函数
的实现代码

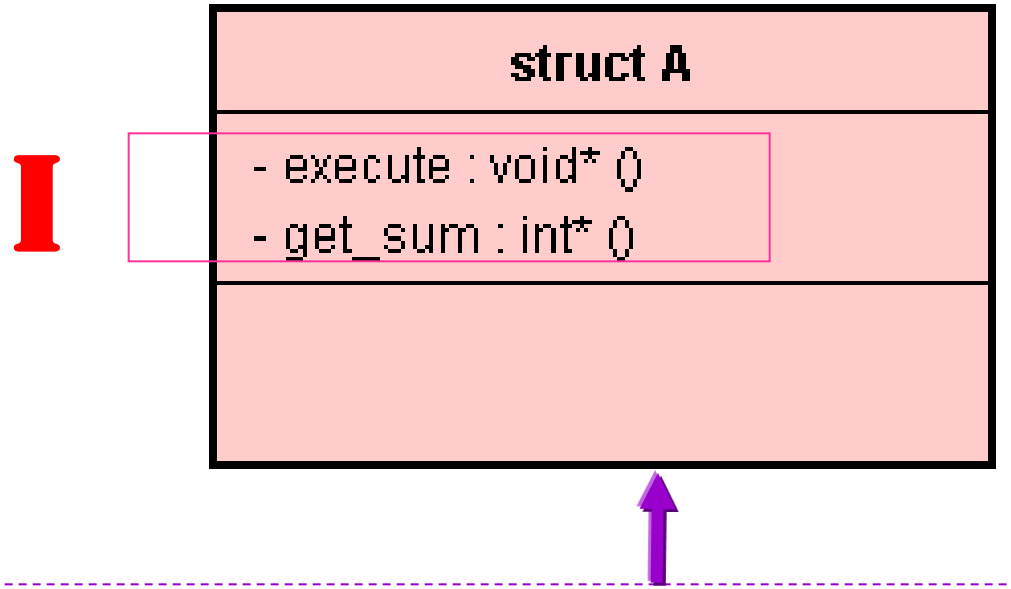
- 基于上述的<E&I>定义，我们如何写<T>呢?

```
void* init() {
    struct A *t =  (struct A*)
            malloc(sizeof(struct A));
    t->execute = exec;
    t->get_sum = get_value;
    return (void*)t;
}
```

(创建对象&设定函数指针)

```
static void exec(){
        // .....
 }
static int get_value() {
        // ......
 }
```

(<I>的函数的实现代码)



I

T

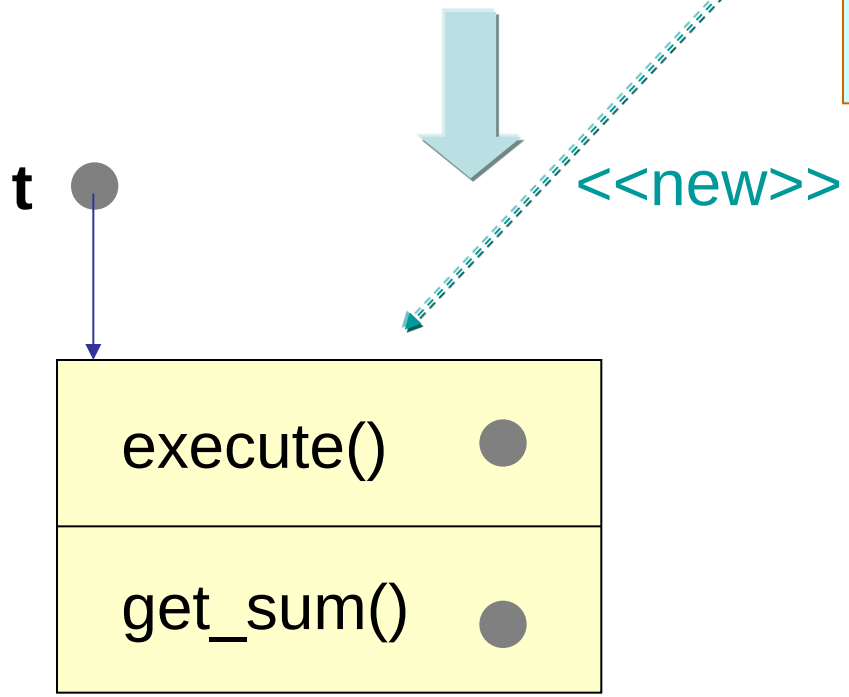
```
void* init() {
    struct A *t =  (struct A*)
            malloc(sizeof(struct A));
    t->execute = exec;
    t->get_sum = get_value;
    return (void*)t;
}
```

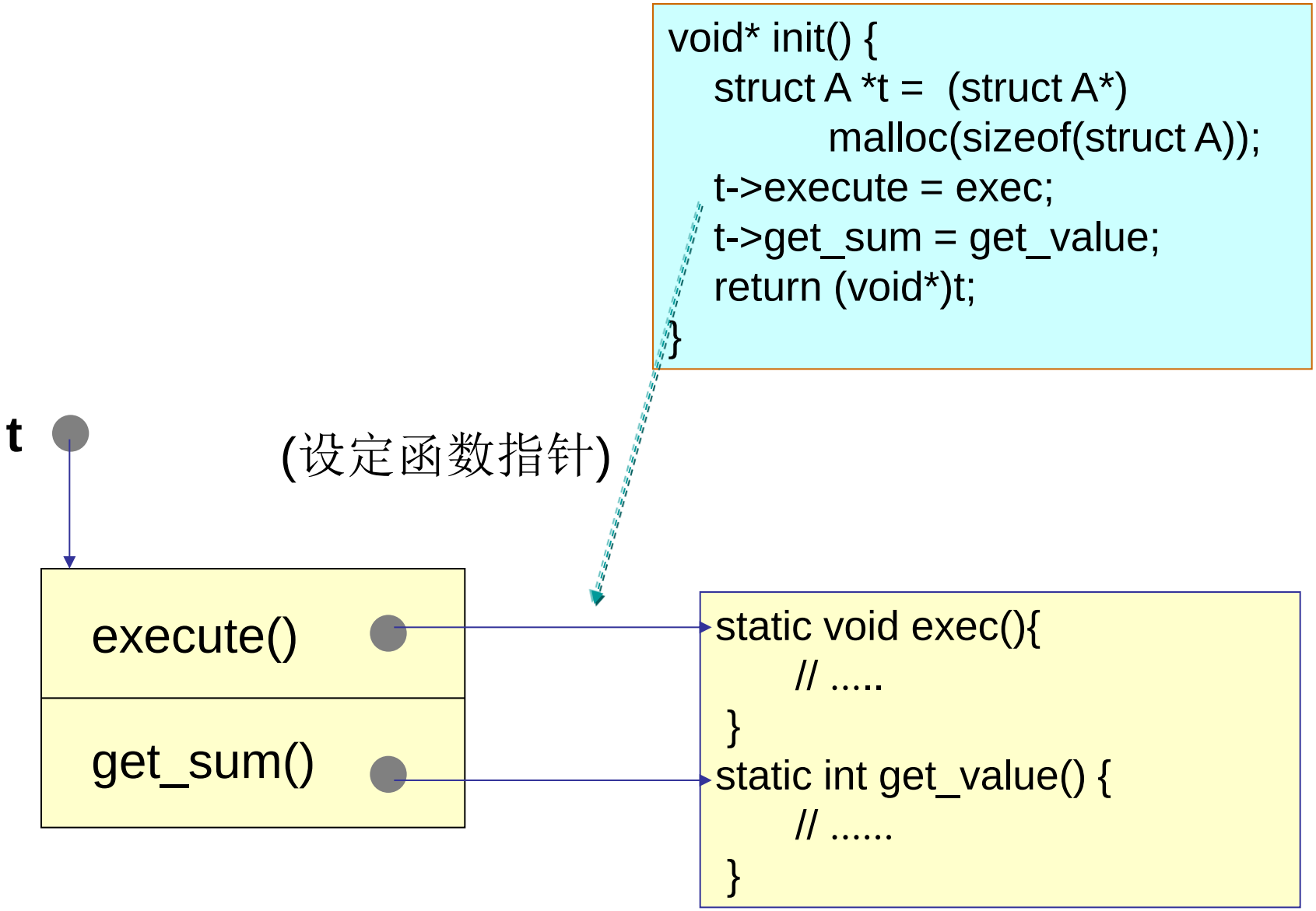
```
static void exec(){
        // .....
 }
static int get_value() {
        // ......
 }
```

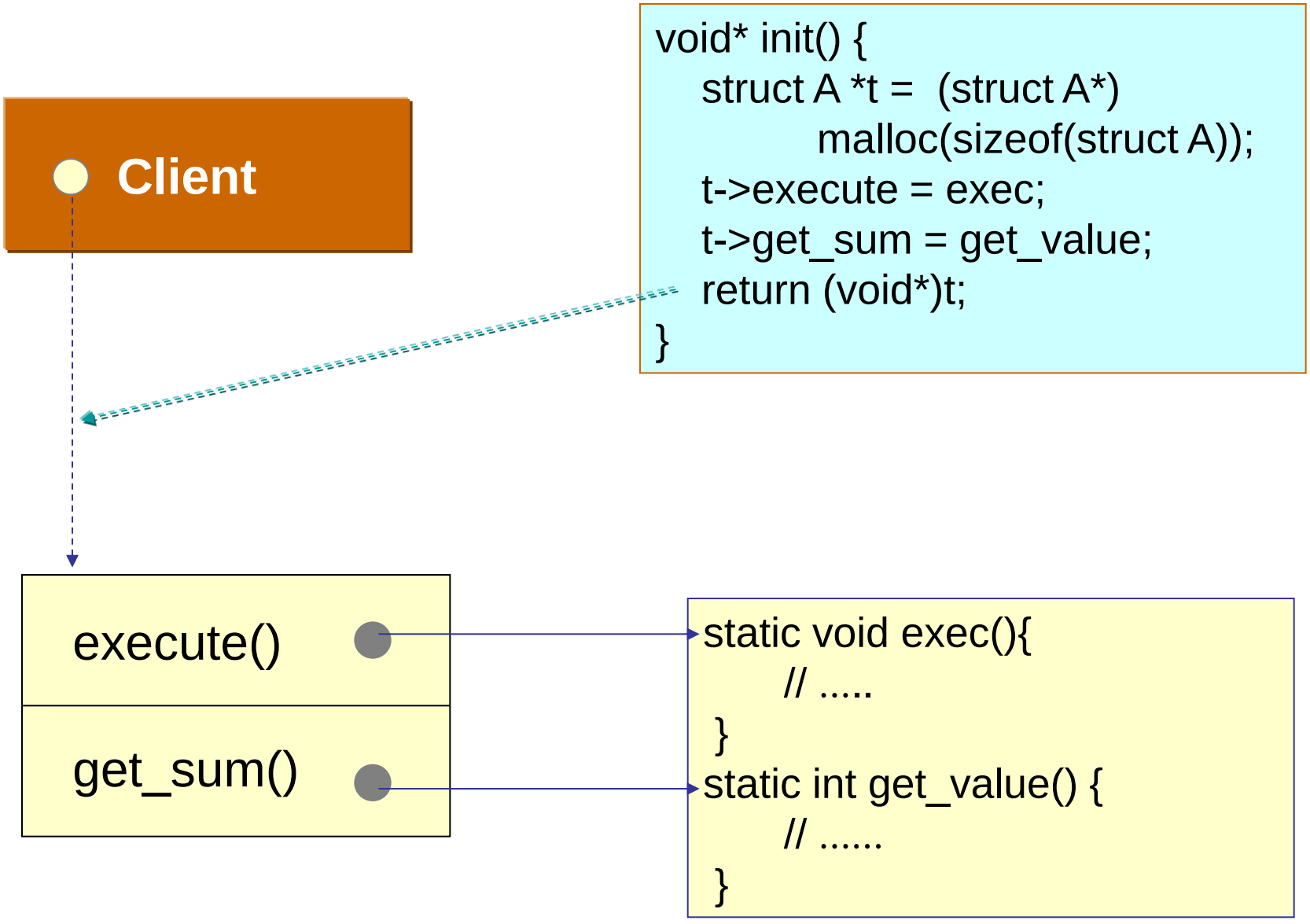
At run-time

```
struct A {
    void (*execute)();
    int (*get_sum)();
};
```

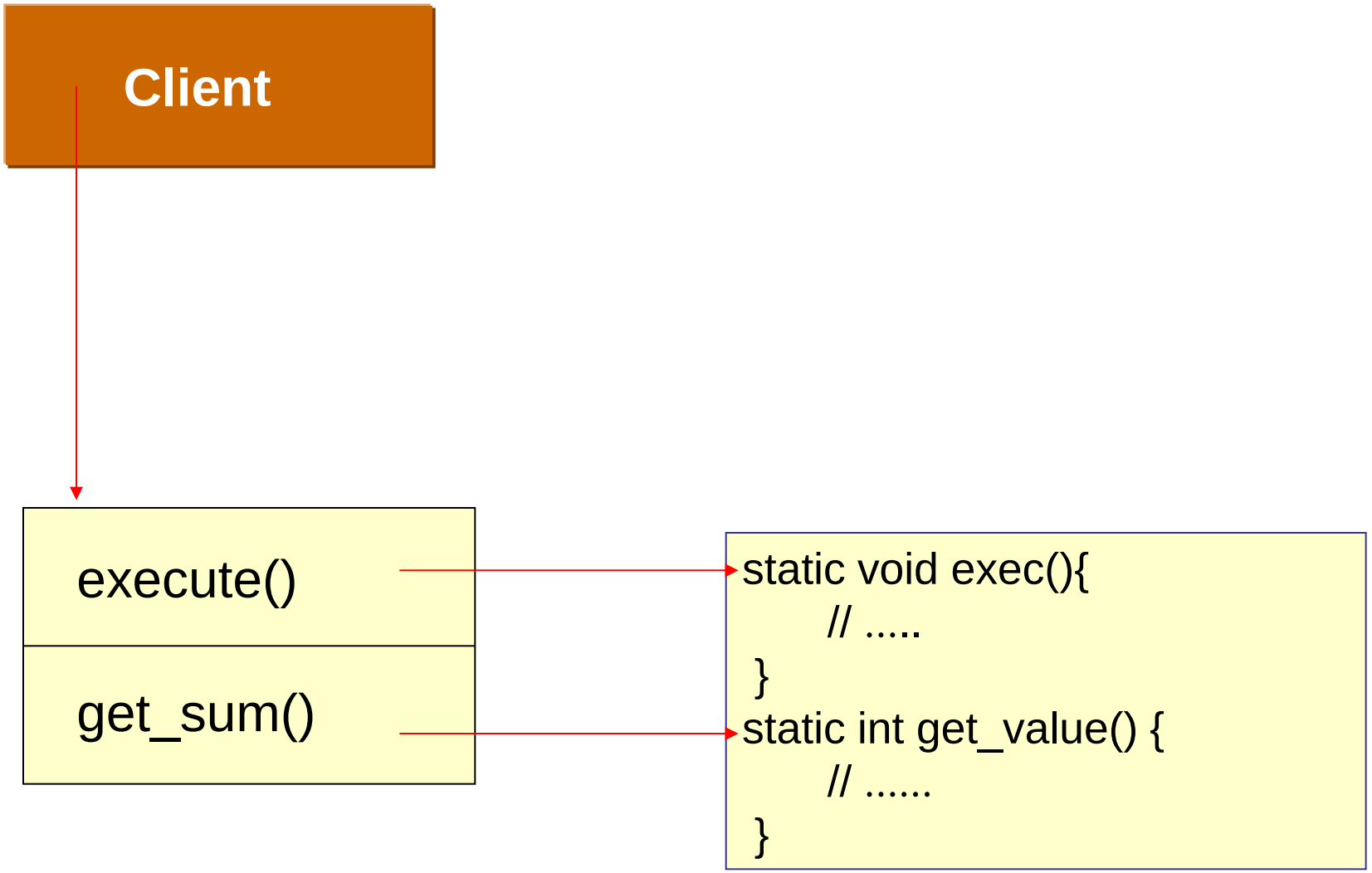
```
void* init() {
    struct A *t =  (struct A*)
            malloc(sizeof(struct A));
    t->execute = exec;
    t->get_sum = get_value;
    return (void*)t;
}
```

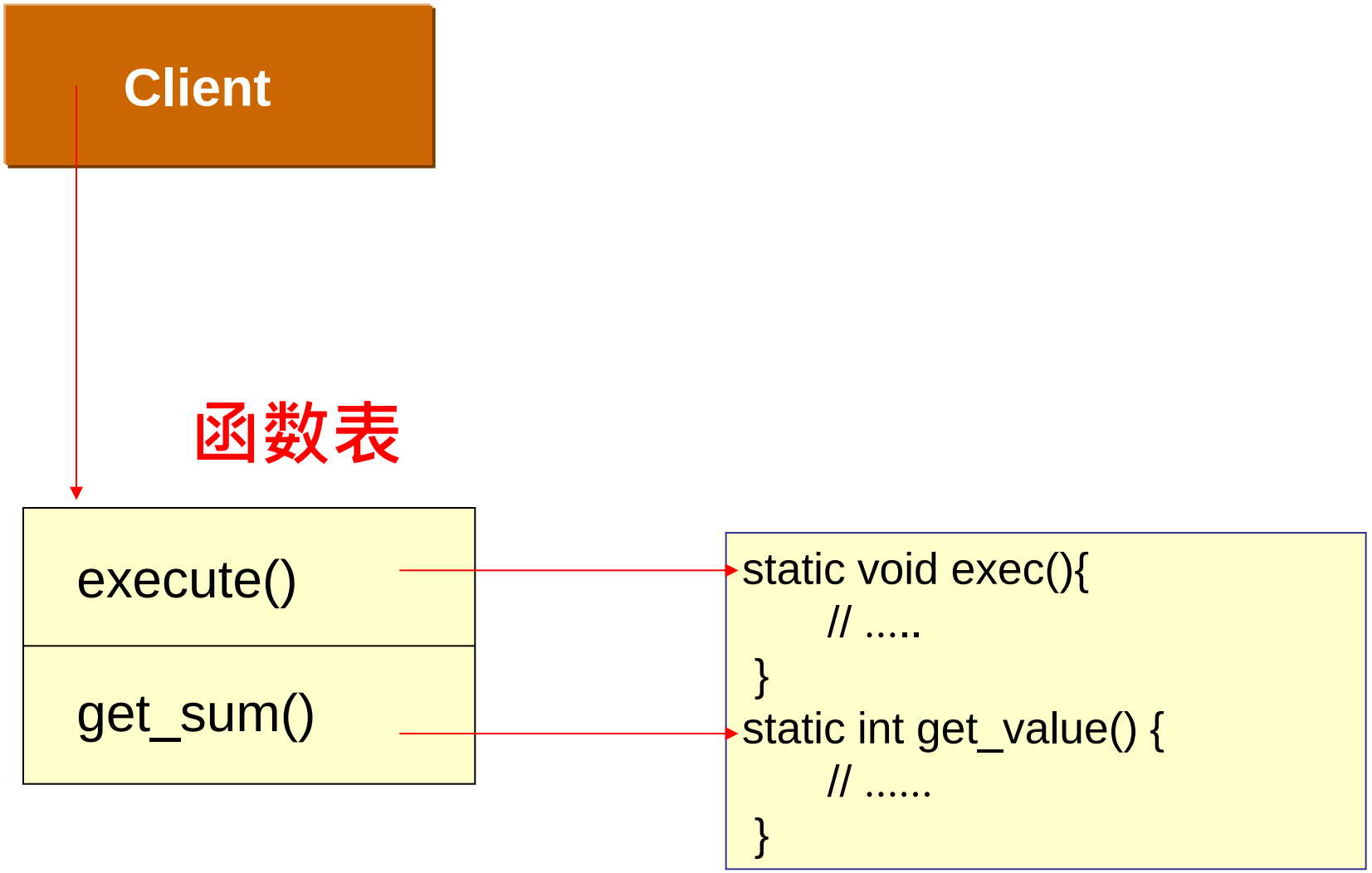



void* init() {
struct A *t = (struct A*)
malloc(sizeof(struct A));
t->execute = exec;
t->get_sum = get_value;
return (void*)t;
}
t
(设定函数指针)
execute()
get_sum()
static void exec(){
// .....
}
static int get_value() {
// .....
}


Client
void* init() {
struct A *t = (struct A*)
malloc(sizeof(struct A));
t->execute = exec;
t->get_sum = get_value;
return (void*)t;
}
execute()
get_sum()
static void exec(){
// .....
}
static int get_value() {
// .....
}

Client调用函数

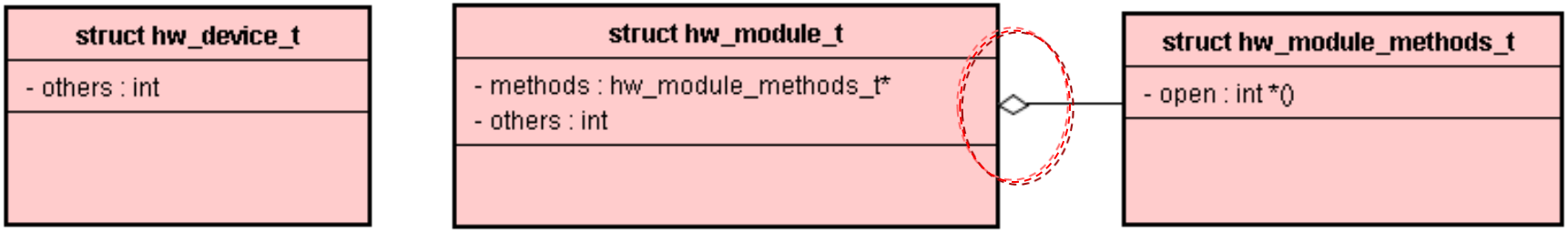

Client
execute()
get_sum()
static void exec(){
// .....
}
static int get_value() {
// ......
}


Client
函数表
execute()
get_sum()
static void exec(){
// .....
}
static int get_value() {
// ......
}

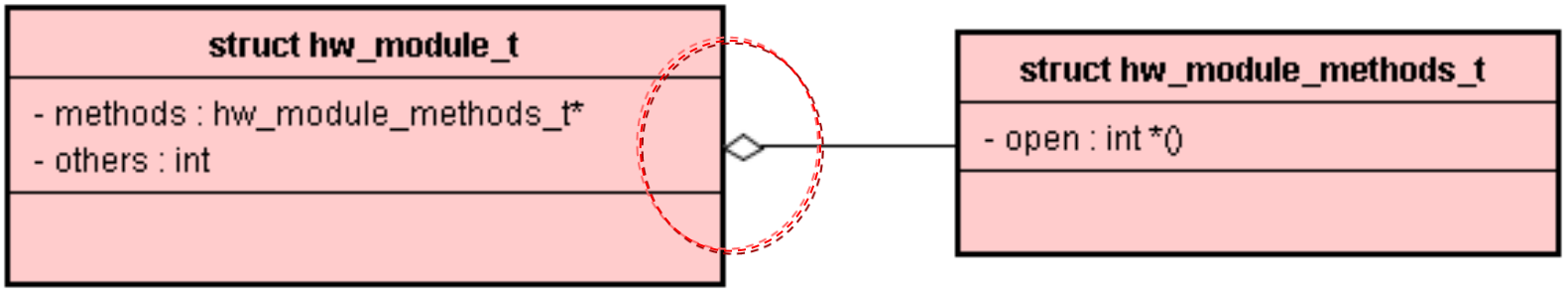
HAL的架构

- HAL框架里只有3个主要的struct结构。
- 其中的hw_module_methods_t是从hw_module_t独立出来的<函数表定义>。

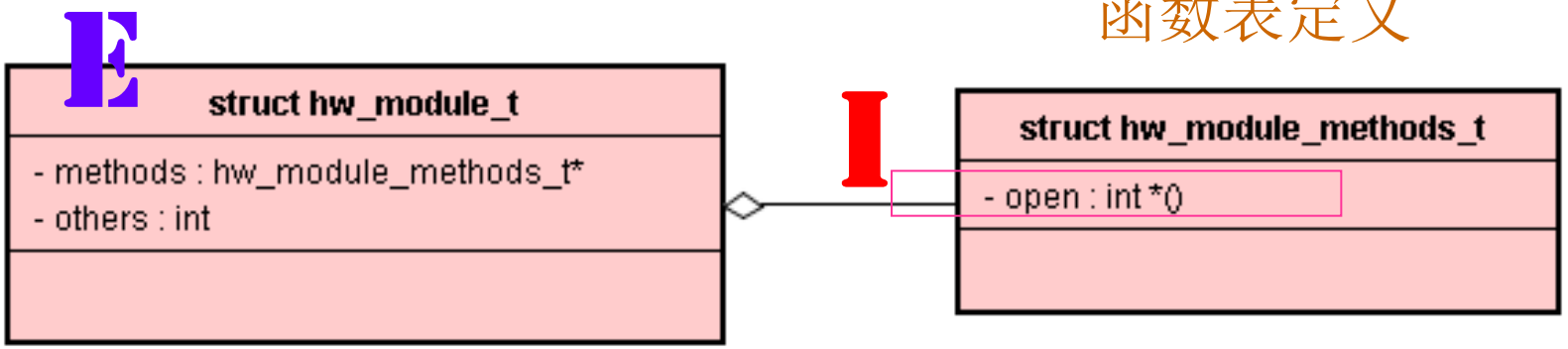
函数表定义


struct hw_device_t
- others : int
struct hw_module_t
- methods : hw_module_methods_t*
- others : int
struct hw_module_methods_t
- open : int *()

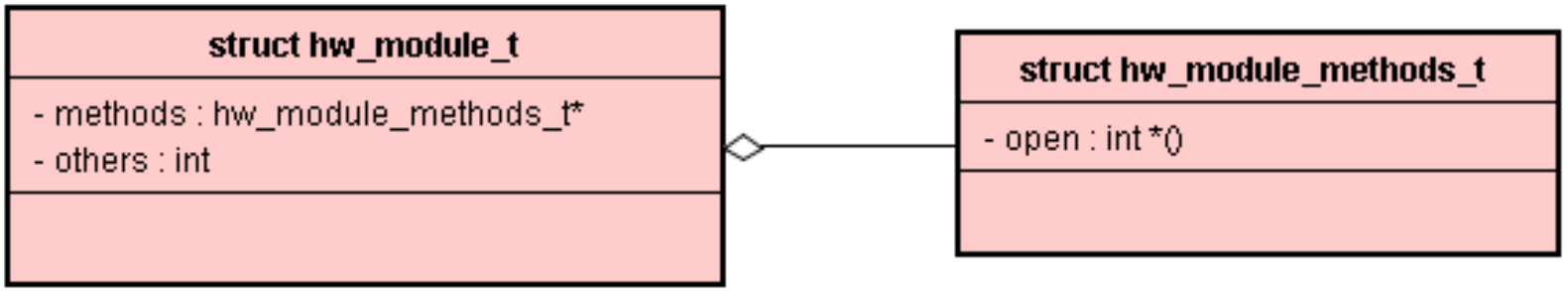
函数表定义


struct hw_module_t
- methods : hw_module_methods_t*
- others : int
struct hw_module_methods_t
- open : int *()

函数表定义

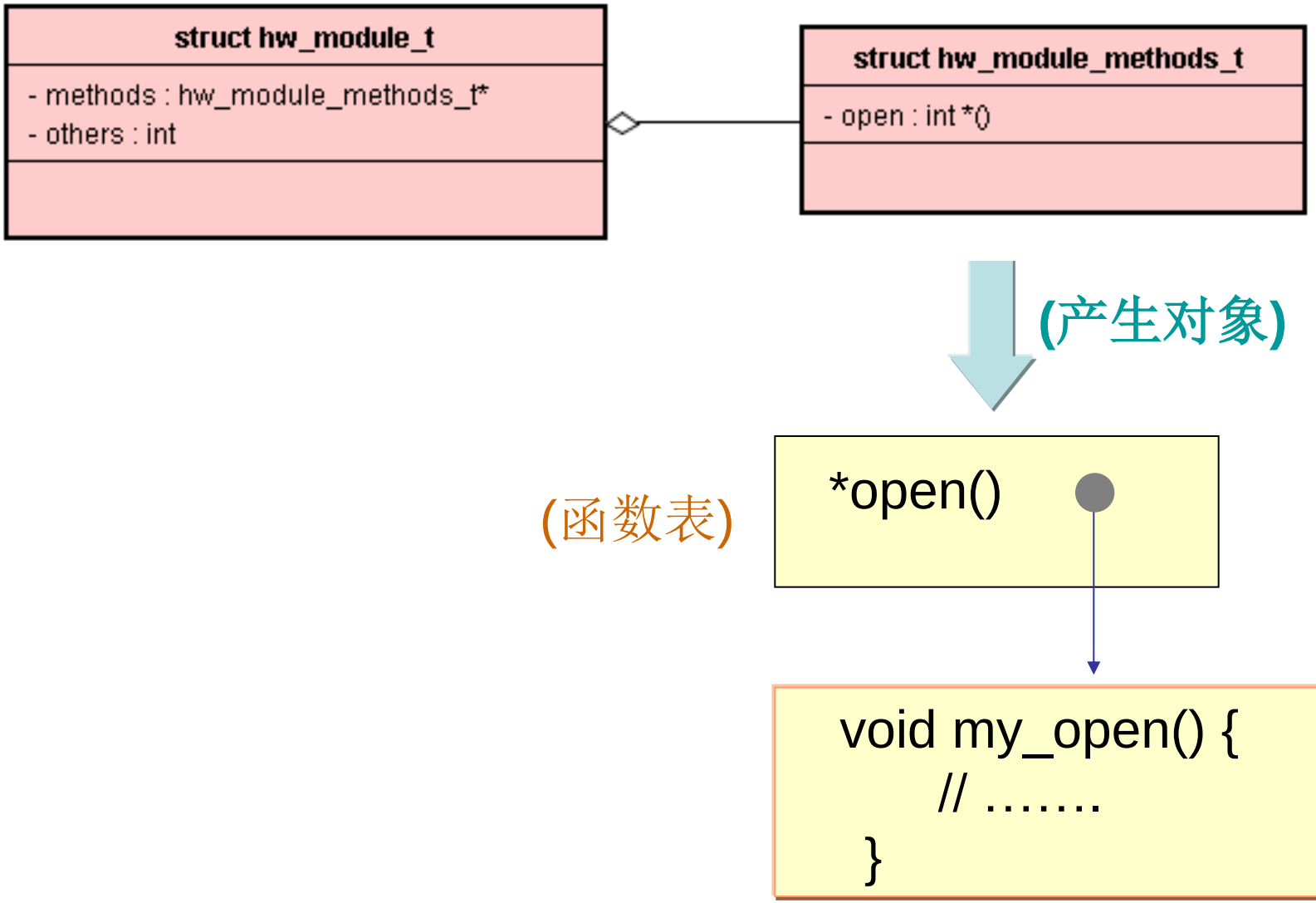

E
struct hw_module_t
- methods : hw_module_methods_t*
- others : int
I
struct hw_module_methods_t
- open : int *()

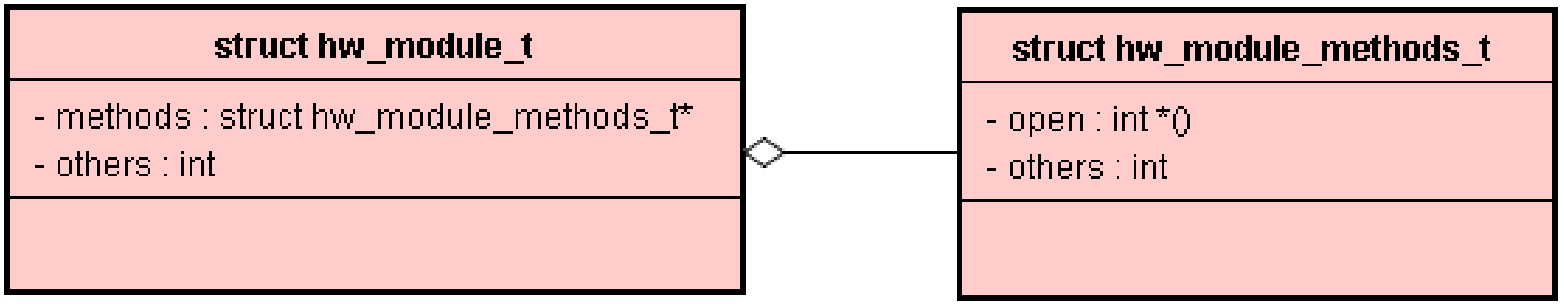
函数表定义




```
void my_open() {
    // .....
}
```

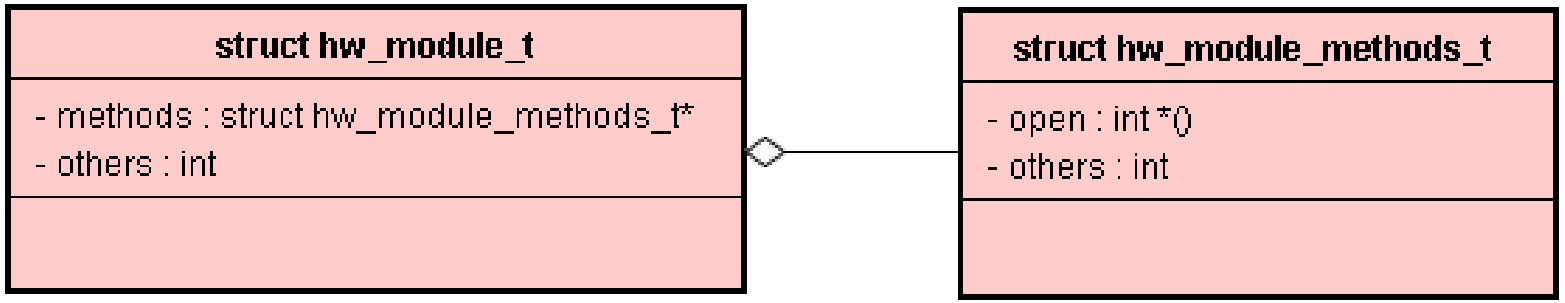
函数表定义


struct hw_module_t
- methods : hw_module_methods_t*
- others : int
struct hw_module_methods_t
- open : int *()
(产生对象)
(函数表)
*open()
void my_open() {
// .....
}



写代码来创建对象

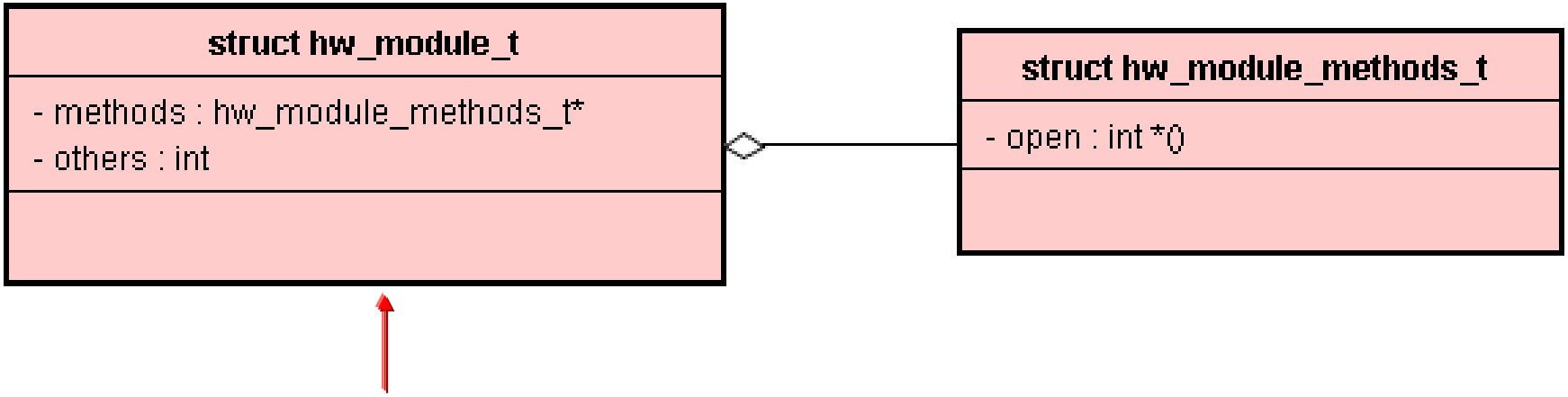
```
void my_open() {
    // .....
}
```




```
static struct hw_module_methods_t  my_methods = {
    open: my_open
};
```

```
const struct hw_module_t
   HAL_MODULE_INFO_SYM = {
        // ……..
        methods: &my_methods,
    };
```

```
void my_open() {
        // …….
    }
```

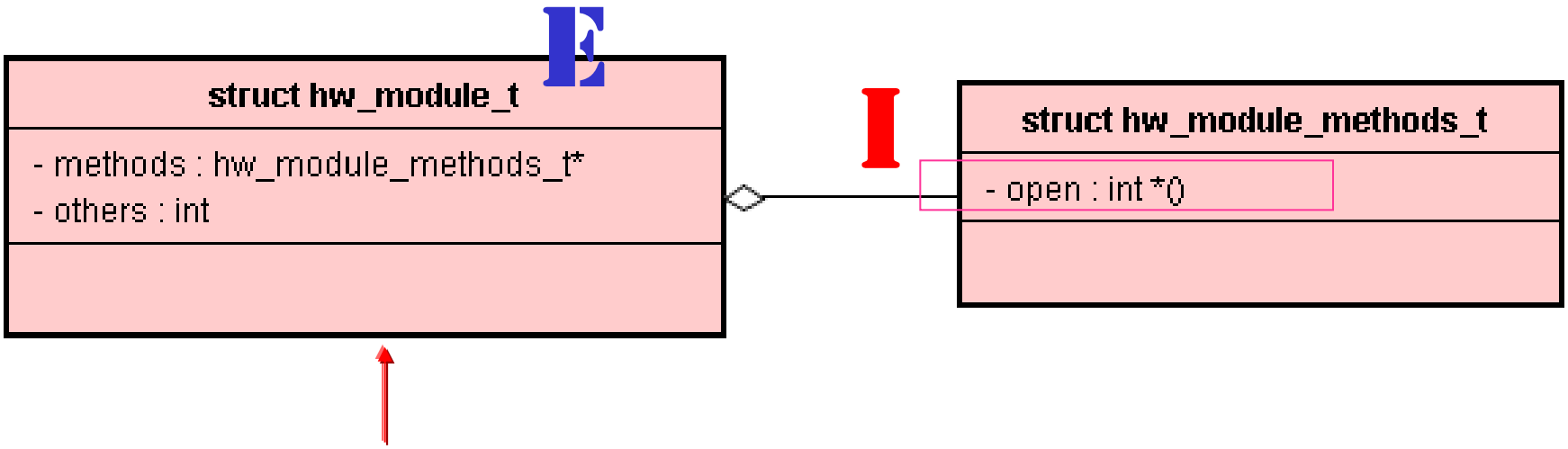



```
static struct hw_module_methods_t  my_methods = {
    open: my_open
};
```

```
const struct hw_module_t
   HAL_MODULE_INFO_SYM = {
        // ……..
        methods: &my_methods,
    };
```

```
void my_open() {
        // …….
    }
```

HAL_Stub




```
static struct hw_module_methods_t  my_methods = {
    open: my_open
};
```

T

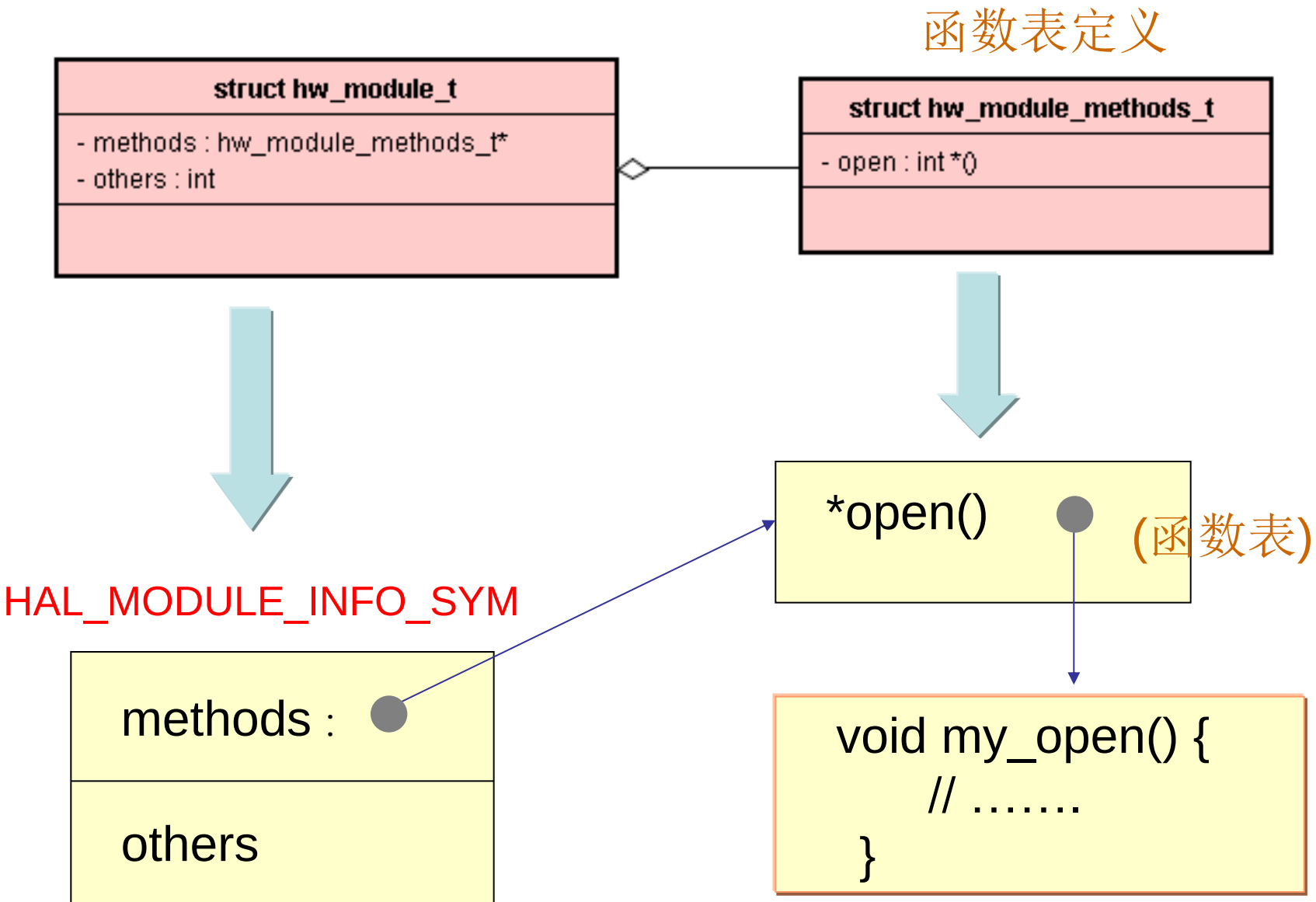
```
const struct hw_module_t
   HAL_MODULE_INFO_SYM = {
        // ……..
        methods: &my_methods,
    };
```

```
void my_open() {
        // …….
 }
```

- 写好了上述的HAL-Stub代码，就能编译&连结成为*.so文檔。

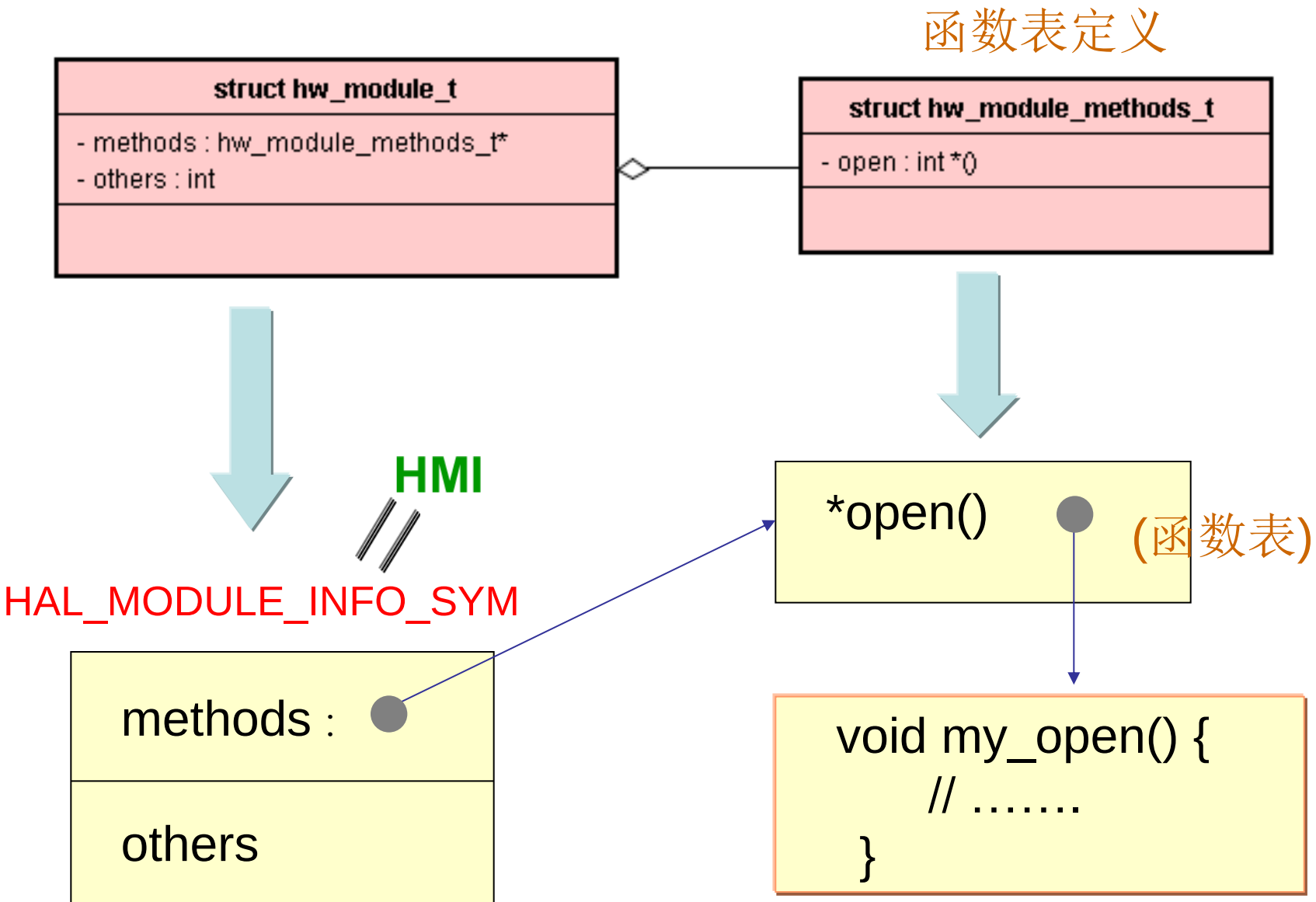
- 载入*.so文檔，执行这些HAL-Stub代码，在run-time就创建对象，并设定函数指针，如下图：

Run-time


函数表定义
struct hw_module_t
- methods : hw_module_methods_t*
- others : int
struct hw_module_methods_t
- open : int *()
*open()
(函数表)
HAL_MODULE_INFO_SYM
methods :
others
void my_open() {
// .....
}

- 在HAL框架里，定义了如下：

```
#define HAL_MODULE_INFO_SYM  HMI
#define HAL_MODULE_INFO_SYM_AS_STR  "HMI"
```

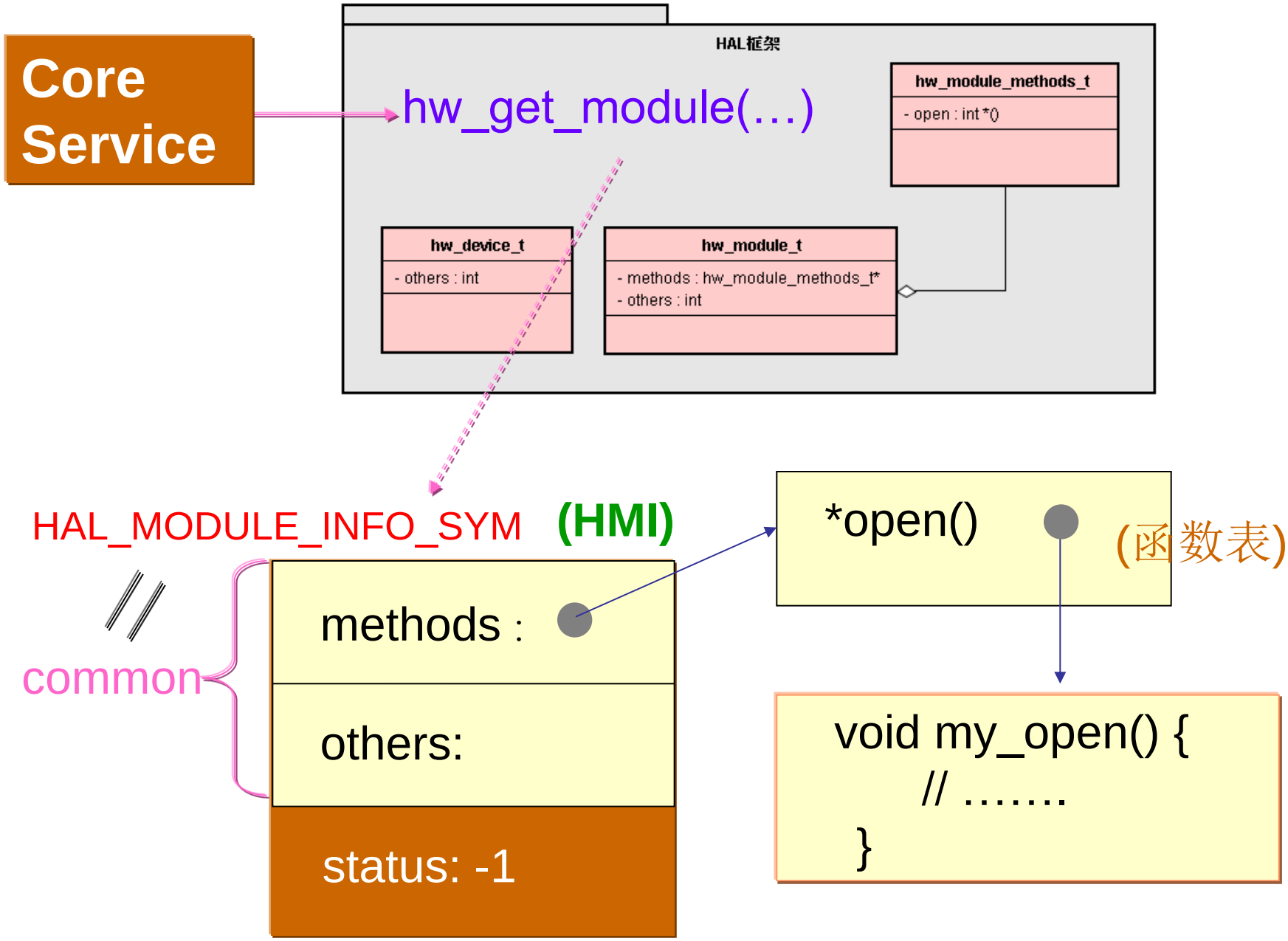

函数表定义
struct hw_module_t
- methods : hw_module_methods_t*
- others : int
struct hw_module_methods_t
- open : int *()
HMI
HAL_MODULE_INFO_SYM
*open()
(函数表)
methods :
others
void my_open() {
// .....
}

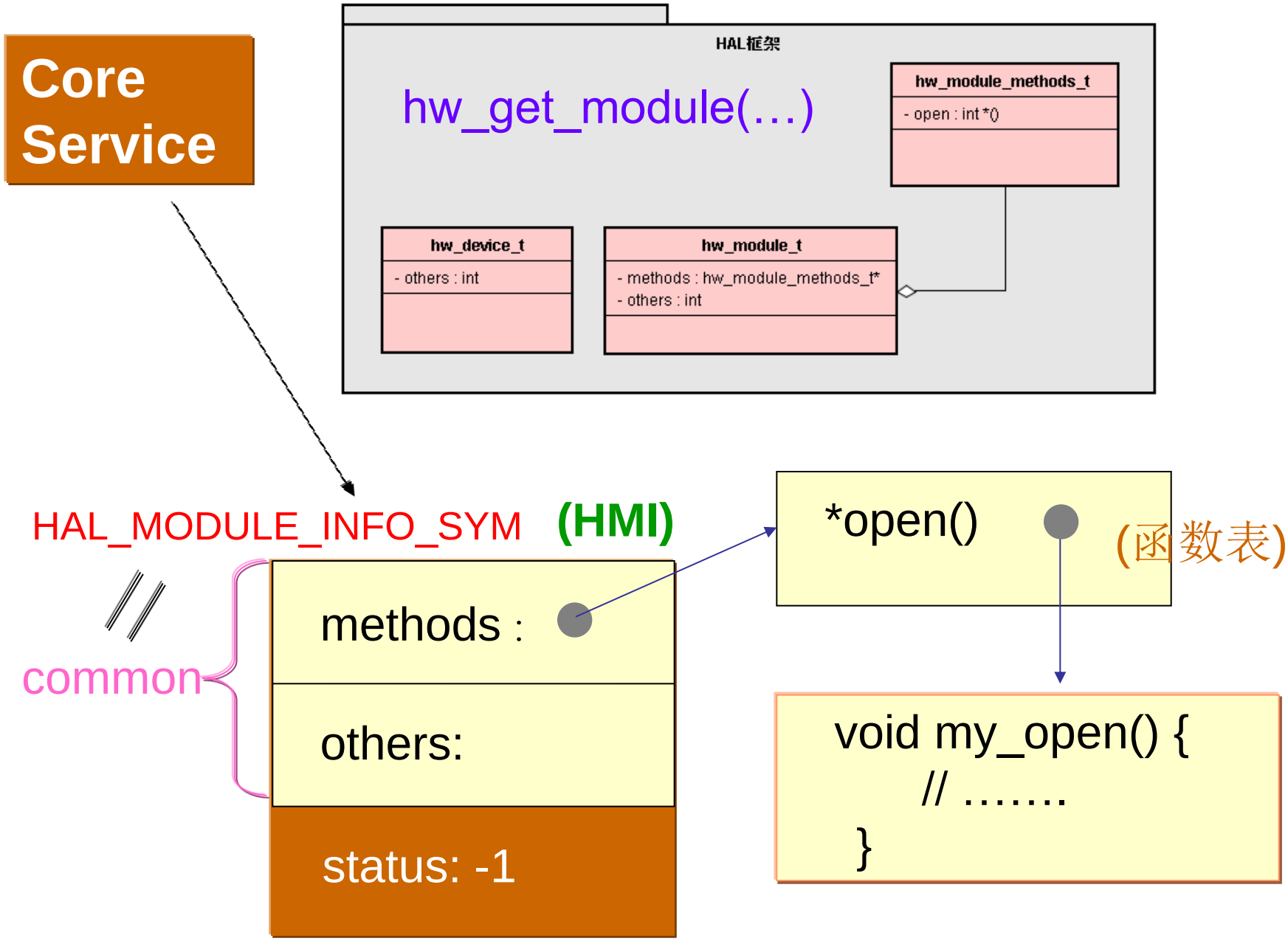
Client使用HAL的第1个步骤

- HAL框架提供了一个公用的函数：
- hw_get_module(const char *id, const struct hw_module_t **module)

- 这个函数的主要功能是根据模块ID(module_id)去查找注册在当前系统中与id对应的硬件对象，然后载入(load)其相应的HAL层驱动模块的*so文件。

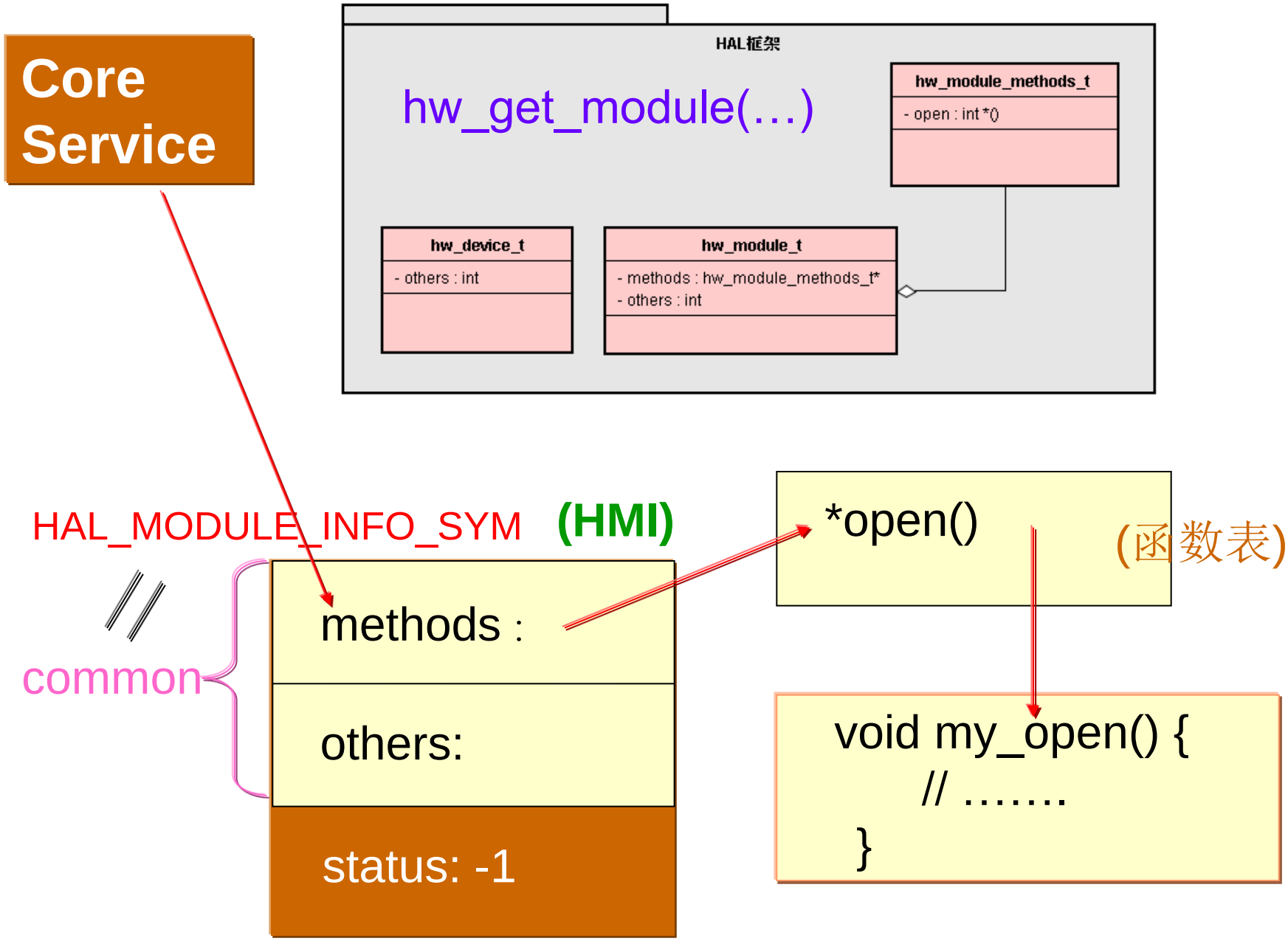
- 從*.so里查找”HMI”这个符号，如果在so代码里有定义的函数名或变量名为HMI，返回其地址。


Core Service
HAL框架
hw_get_module(…)
hw_module_methods_t
- open : int *()
hw_device_t
- others : int
hw_module_t
- methods : hw_module_methods_t*
- others : int
HAL_MODULE_INFO_SYM
(HMI)
common
methods :
others:
status: -1
*open()
(函数表)
void my_open() {
// …….
}


Core Service
HAL框架
hw_get_module(…)
hw_module_methods_t
- open : int *()
hw_device_t
- others : int
hw_module_t
- methods : hw_module_methods_t*
- others : int
HAL_MODULE_INFO_SYM
(HMI)
common
methods :
others:
status: -1
*open()
(函数表)
void my_open() {
// …….
}

- 從*.so里查找”HMI”这个符号，如果在so代码里有定义的函数名或变量名为HMI，返回其地址。

Client使用HAL的第2个步骤


Core Service
HAL框架
hw_get_module(…)
hw_module_methods_t
- open : int *()
hw_device_t
- others : int
hw_module_t
- methods : hw_module_methods_t*
- others : int
HAL_MODULE_INFO_SYM
(HMI)
common
methods :
others:
status: -1
*open()
(函数表)
void my_open() {
// …….
}

~ Continued ~