以 OOPC 封裝與 HAL 的 API

PS. OOPC 是由高煥堂所創的，並提供爲 GPL 協議下的開源軟件，歡迎使用。

內容：

1. 複習面向對象 OOPC 語言

認識 LW_OOPC

LW_OOPC 是一種輕便又快速的面向對象 C 語言。在嵌入式程序師還是蠻青睞 C 語言的，只是 C 語言沒有對象、類等概念，程序很容易變成義大利面型的結構，維護上比較費力。在 1986 年 C++上市時，希望大家改用 C++，但是 C++的效率不如 C，並不受嵌入式程序師的喜愛。於是，由高煥堂領導的 MISOO 團隊設計一個輕便又高效率的 OOPC 語言。輕便的意思是：它只用了約 20 個 C 宏而已，簡單易學。其宏如下：

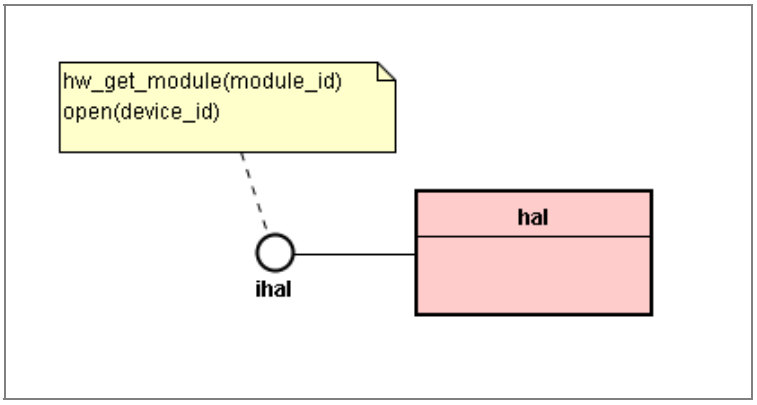
```
/*    lw_oopc.h */    /*  這就MISOO團隊所設計的C宏  */
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;    };
```

```
#define FUNCTION_SETTING(f1, f2)    t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) struct type
#endif
/*          end          */
```

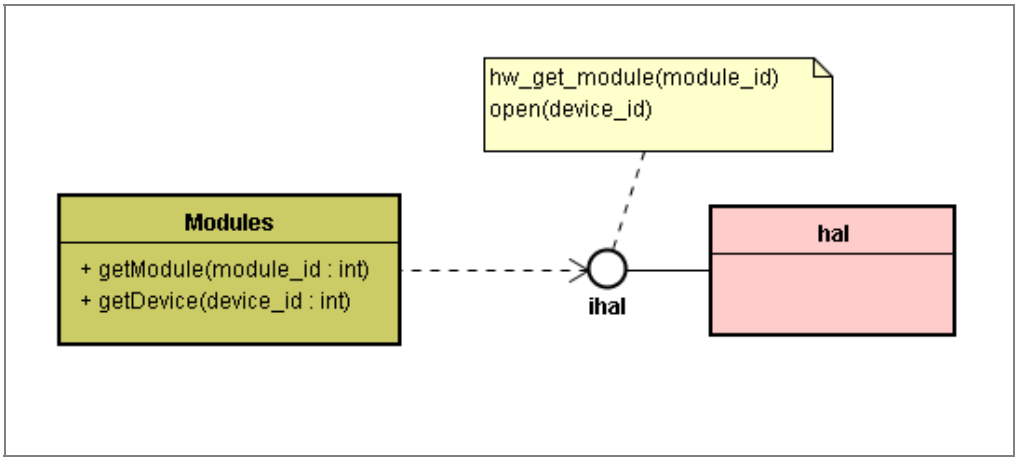
其高效率的意思是，它沒提供類繼承，內部沒有虛擬函數表(Virtual Function Table)，所以仍保持原來 C 語言的高效率。除了沒有繼承機制之外，它提供有類、對象、資訊傳遞、接口和接口多型等常用的機制。目前受到不少 C 程序員的喜愛。

2. 初步封裝 HAL 的 API

Android 的原來 HAL 架構接口如下：



可用 OOPC 定義一個 Modules 類如下：



這把 ihal 接口包裝起來，而呈現新的接口。此 Modules 類的定義如下：

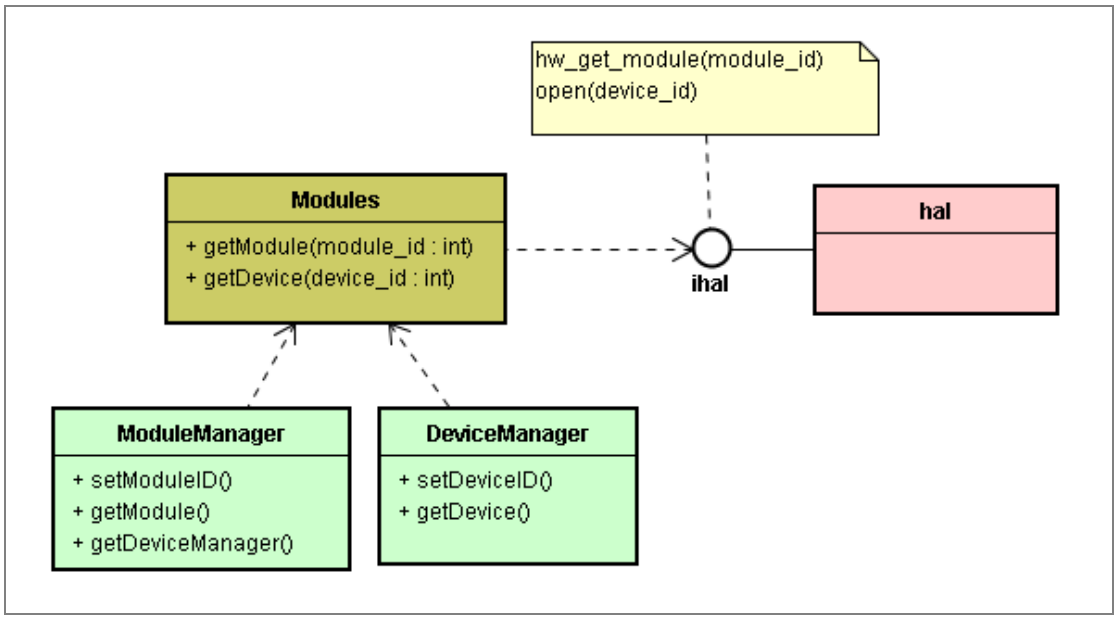
```
/* Modules.h    */
#ifndef MODULES_H
#define MODULES_H

#include <misoo/lw_oopc.h>
```

```
#include <libled/IModule.h>
#include <libled/IDevice.h>
#include <stdio.h>

CLASS(Modules)
{
	struct hw_module_t* phw_mod;
	struct IModule* p_mod;

	IModule* (*getModule)(Modules* thiz, const char* mod_id);
	IDevice* (*getDevice)(Modules* thiz, const char* dev_id);
};
Modules* getModulesInstance();
#endif
```

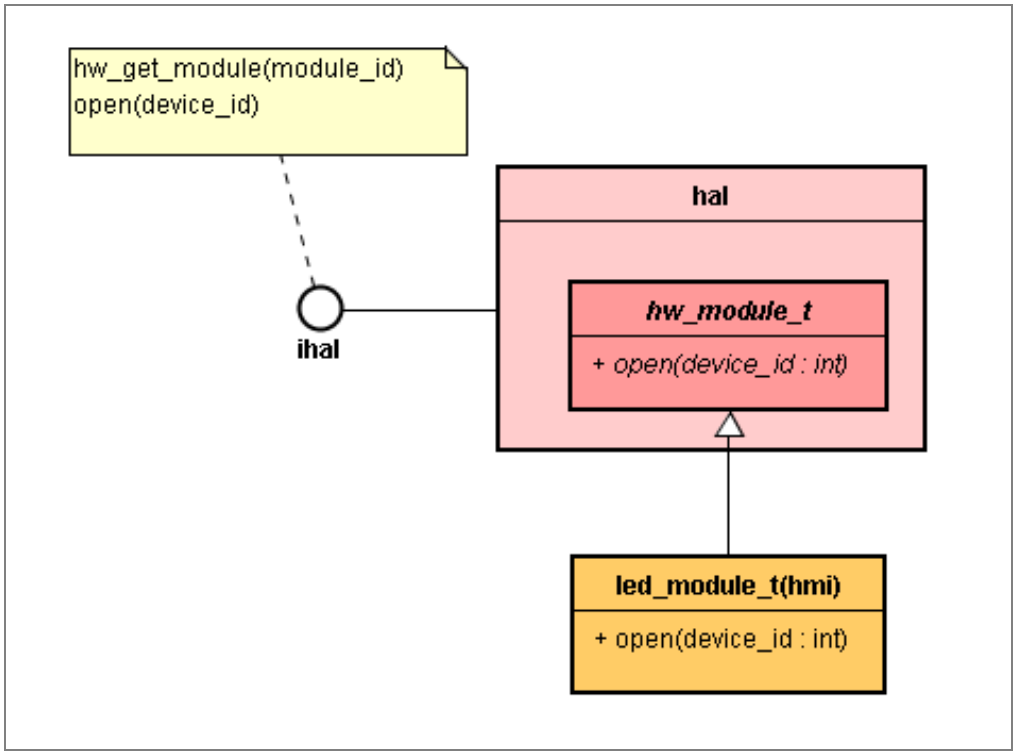
```
/*	Modules.c	*/
#include "Modules.h"
#include <hardware/hardware.h>
#include <stdio.h>

extern IModule* getModuleInstance();

static Modules* gModules = 0;

Modules* getModulesInstance()
{
	if (gModules == 0)
	{
		printf("getModulesInstance\n");
		gModules = Modules_new();
	}
	return gModules;
}

//private :
static check(Modules* thiz)
{
	if (thiz->p_mod == 0)
	{
		printf("Error : p_mod = 0\n");
		return 0;
	}

	return 1;
}

// public :
static IDevice* getDevice(Modules* thiz, const char* dev_id)
{
	if (! check(thiz) )
		return 0;

	IDevice* dev = 0;
	thiz->phw_mod->methods->open((hw_module_t*)(thiz->p_mod), dev_id,
```

```
(hw_device_t**)(&dev));
        return dev;
}

static IModule* getModule(Modules* thiz, const char* mod_id)
{
        int t = hw_get_module(mod_id, (const struct hw_module_t**)&(thiz->phw_mod));
        if (t != 0 )
        {
                printf("Error : hw_get_module = -1\n");
                return 0;
        }

        thiz->p_mod = getModuleInstance();
        if (! check(thiz) )
                return 0;

        thiz->phw_mod->methods->open = thiz->p_mod->openDev;

        return thiz->p_mod;
}

CTOR(Modules)
        FUNCTION_SETTING(getModule, getModule);
        FUNCTION_SETTING(getDevice, getDevice);
END_CTOR

DTOR(Modules)
END_DTOR
```

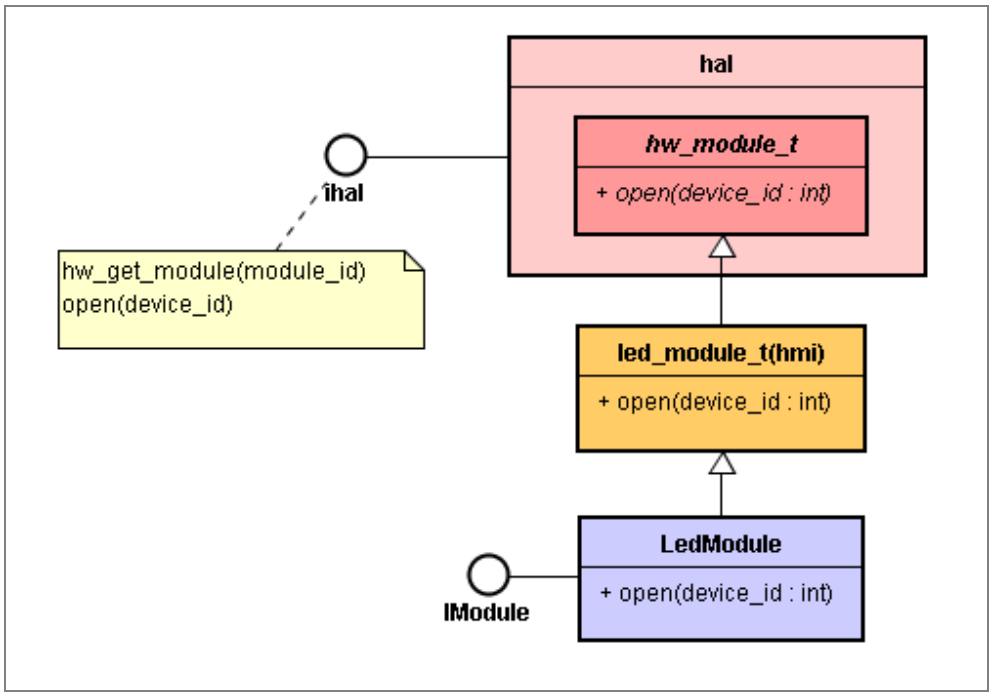
依樣畫葫蘆，可以替Midules定義兩個擔任Helper角色的類：ModuleManager和DeviceManager，如下圖：



定義往下層的接口。原來的接口：



可定義新接口：



此 LedModule 類的定義：

```
/* LedModule.h */
#ifndef LED_MODULE_H
```

```
#define LED_MODULE_H
#include <misoo/lw_oopc.h>
#include "LedControlDevice.h"
#include "LedDataDevice.h"
#include "IModule.h"

CLASS(LedModule)
{
	EXTENDS(IModule);

	LedControlDevice* (*onLedControlDevice)(LedModule* thiz);
	LedDataDevice* (*onLedDataDevice)(LedModule* thiz);
	int (*getDevIdNum)(LedModule* thiz);
	char* (*getDevID)(LedModule* thiz, const int i);

};
#endif
```

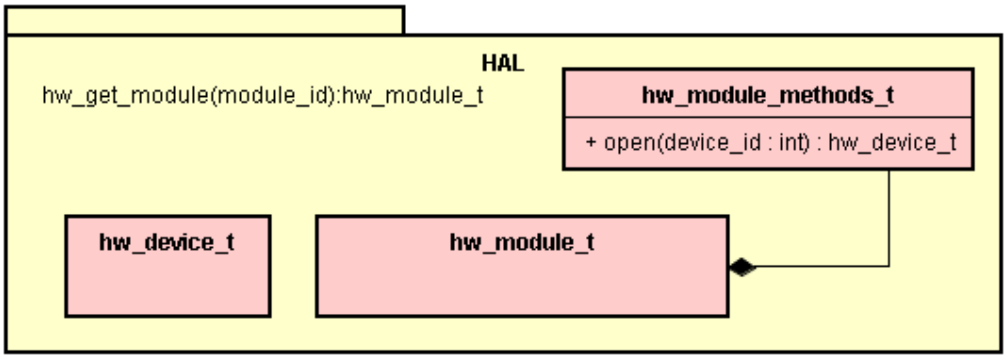
/* LedModule.c */

```
#include "LedModule.h"
#include "LedType.h"
#include <stdio.h>
#include "IDevice.h"

static LedModule* gLedModule = 0;

IModule* getModuleInstance()
{
	if ( gLedModule == 0 )
	{
		printf("getModuleInstance\n");
		gLedModule = LedModule_new();
	}
	return (IModule*)gLedModule;
}


static IDevice* open(IModule* sthiz, const char* dev_id)
{
	printf("LedModule open ...\n");

	LedModule* thiz = (LedModule*)sthiz;

	if (!strcmp(dev_id, LED_CONTROL_DEVICE ))
		return (IDevice*)thiz->onLedControlDevice(thiz);
	else if (!strcmp(dev_id, LED_DATA_DEVICE ))
		return (IDevice*)thiz->onLedDataDevice(thiz);
	else
		return 0;
}

static int getDevIdNum(LedModule* thiz)
```

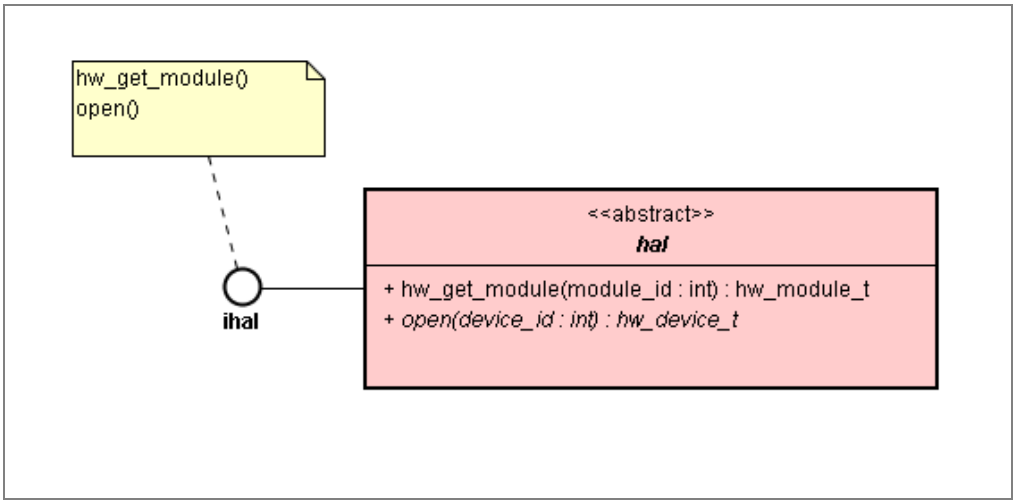
```
{
	printf("LedModule_getDevIdNum ...\n");
	return 2;
}

static char* getDevID(LedModule* thiz, const int i)
{
	printf("LedModule_getDevID ...\n");
	if    ( i == 0)
		return (char*)LED_CONTROL_DEVICE;
	else if ( i == 1 )
		return (char*)LED_DATA_DEVICE;
	else
		return 0;
}

static LedControlDevice* onLedControlDevice(LedModule* thiz)
{
	printf("myLedModule onLedControlDevice ...\n");
	return (LedControlDevice*)LedControlDevice_new();
}

static LedDataDevice* onLedDataDevice(LedModule* thiz)
{
	printf("myLedModule onLedDataDevice ...\n");
	return (LedDataDevice*)LedDataDevice_new();
}

CTOR(LedModule)
	IModule_inheritCtor((IModule*)(&(thiz->IModule)));
	FUNCTION_SETTING(IModule.open, open);
	FUNCTION_SETTING(getDevIdNum, getDevIdNum);
	FUNCTION_SETTING(getDevID, getDevID);
	FUNCTION_SETTING(onLedControlDevice, onLedControlDevice);
	FUNCTION_SETTING(onLedDataDevice, onLedDataDevice);
END_CTOR

DTOR(LedModule)
END_DTOR
```

3. 進一步封裝 HAL 的 API

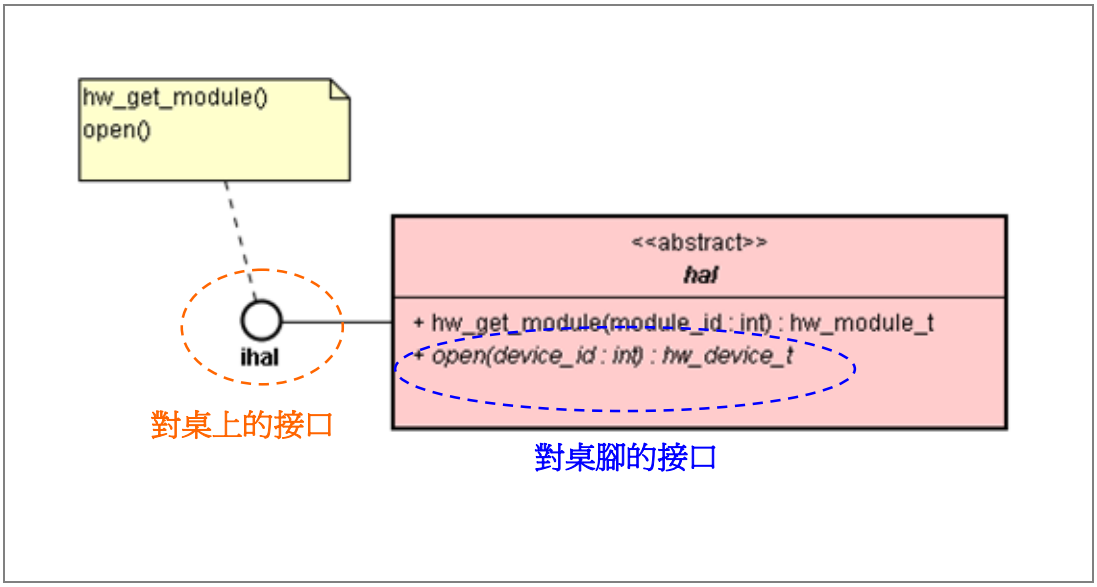
Android 提供的 HAL 內涵：



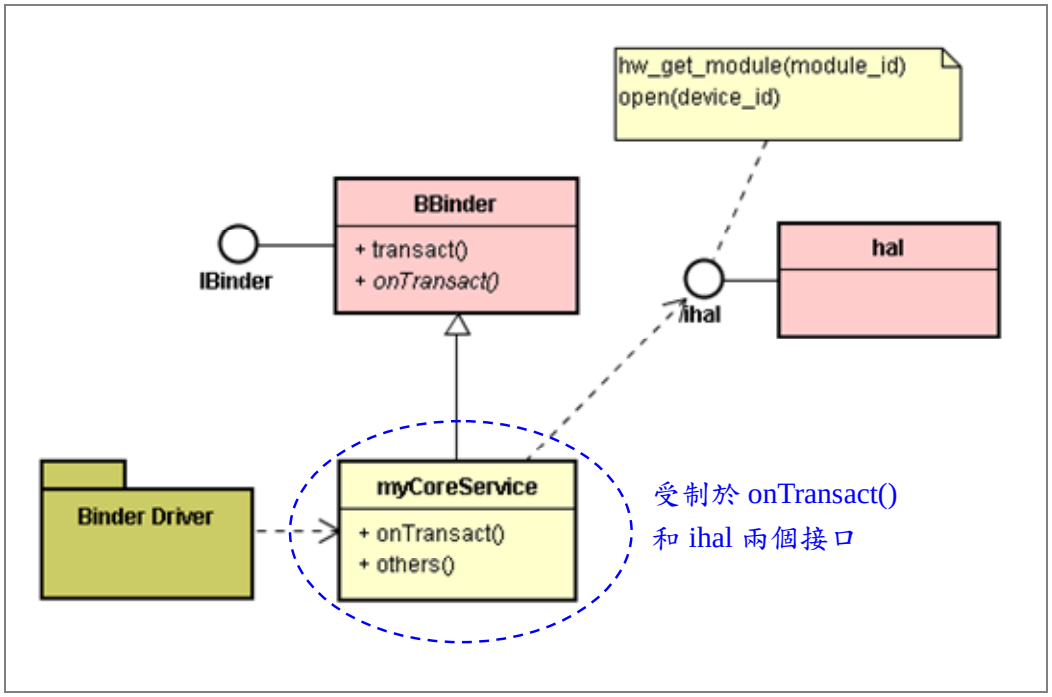
它是一個框架，其接口：



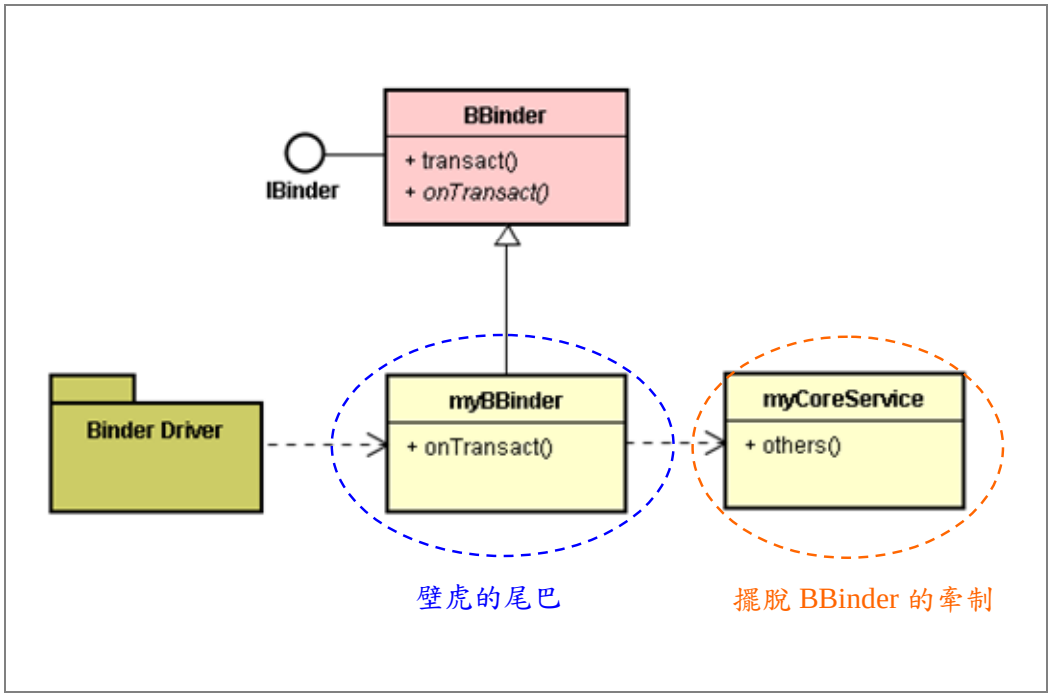
其實，它還有另一個接口：



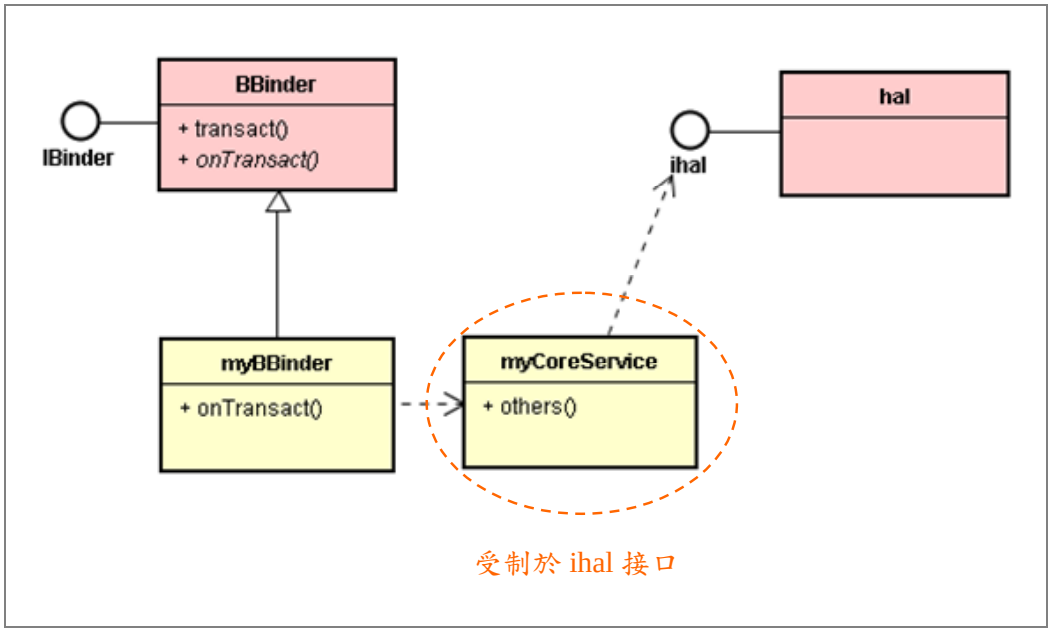
Android 的核心服務(Core Service)或 JNI 模塊，常常會使用 HAL 的接口，如下：



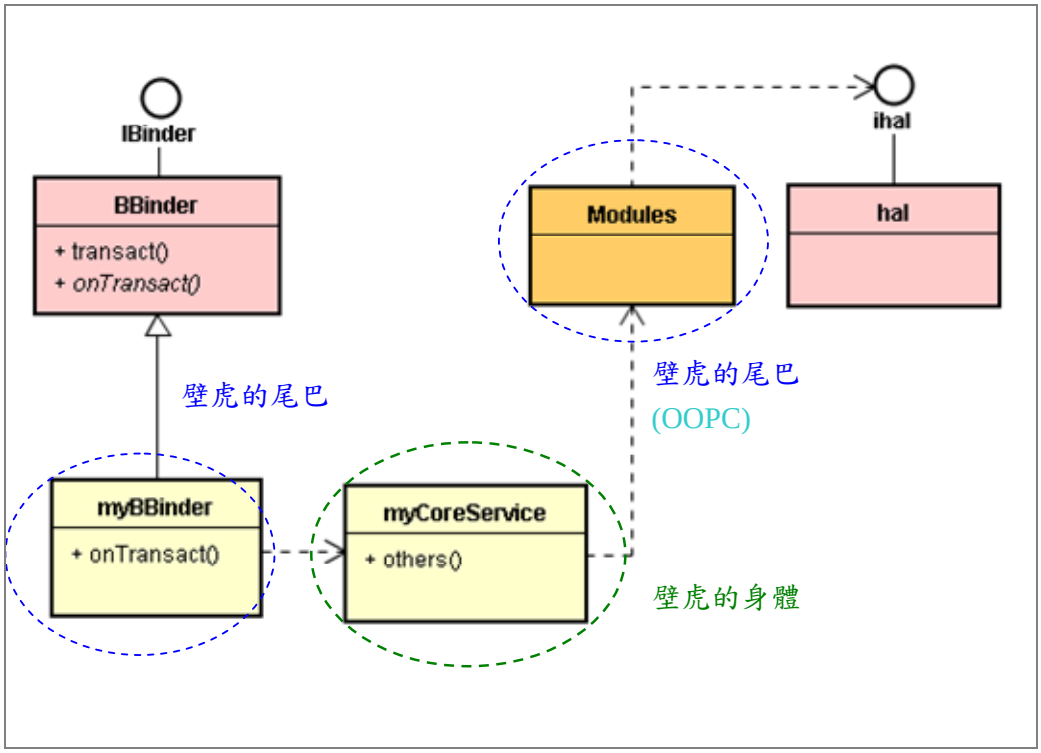
由於沒有進行封裝，Core Service 會受制於 HAL 接口，無法包容 HAL 大框架的變動，不利於 Core Service 的跨平台移植，於是設計壁虎的尾巴：



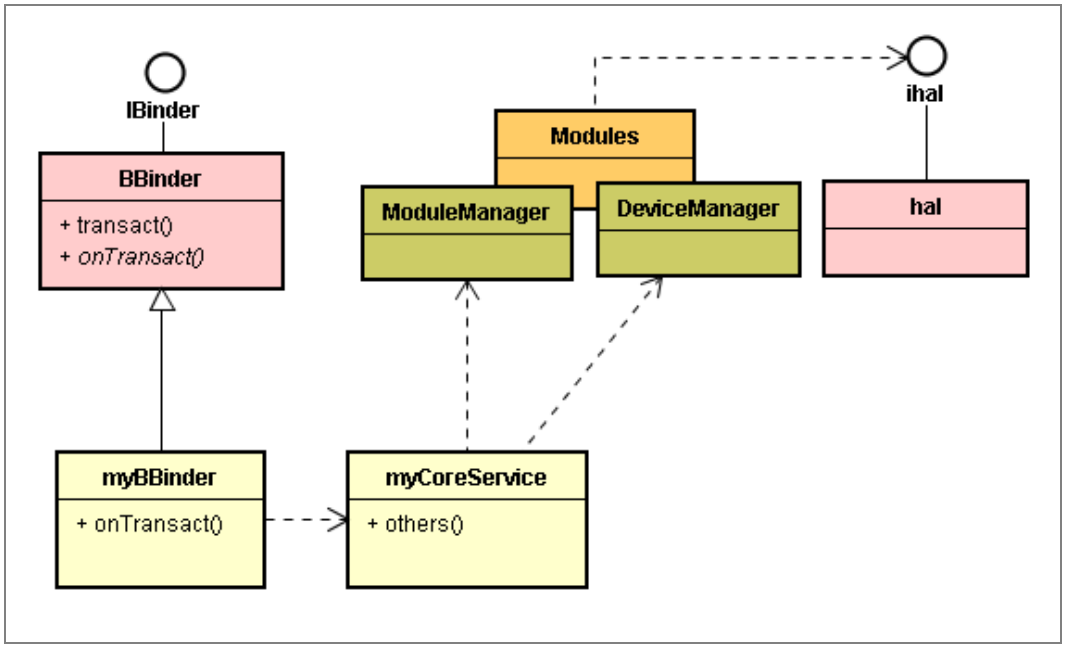
這已經擺脫 BBinder 的牽制，但是請再看看與 ihal 的關係：



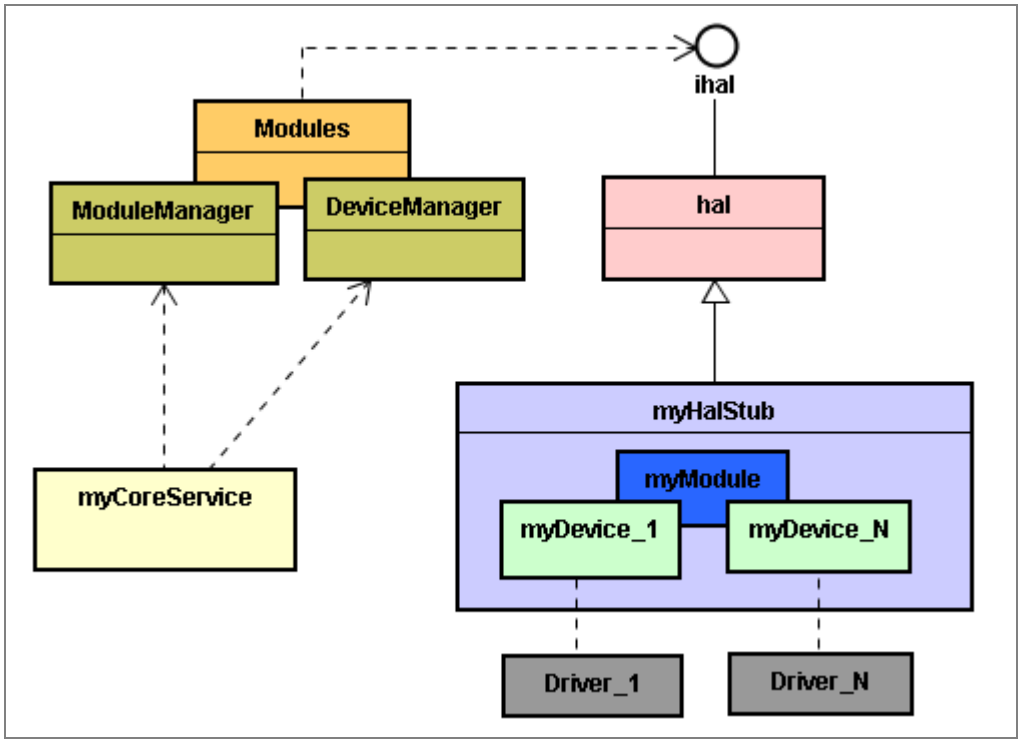
這仍然還受制於 HAL 的接口，於是又設計壁虎的尾巴：



這樣就能包容 HAL 大框架的變動了，也保護了壁虎的身體，增加系統的生命力。此外，還可以繼續精緻化，例如，增加兩個 Helper：



還能加上 HalStub：



這樣就進行較完美的包裝了。

~~ END ~~