第 5 章

新建 Pin 类模块

前面已经验证了 MicroPython 开发文档中描述的新增模块(Module)的操作过程,这种方法多用于创建静态模块。具体来说,就是简单地将 C 语言实现的函数套一个 Python 的“马甲”,以便从 Python 内核调用 C 语言实现的底层函数。

作者在早期研究 MicroPython 时,也曾经使用这种直接套接的方式对 C 语言实现的固件库进行封装,快速实现 C 代码向 MicroPython 的集成。例如,曾经写过如下的 MicroPython 应用代码,见代码 5-1。

代码 5-1　一种使用 Pin 类模块的方法

```
import Pin

Pin.init(0, 2, 1)   # (0, 2) = PTA2, 1 = GPIO_PinMode_Out_PushPull
Pin.write(0, 2, 1)  # (0, 2) = PTA2, 1 = LOGIC_1
Pin.write(0, 2, 0)  # (0, 2) = PTA2, 0 = LOGIC_0
```

在代码 5-1 中,虽然已经实现了 Python 对底层的调用,但这种用法实际上还是 C 语言的使用方式。

Python 语言的特点在于:一切皆对象,并且可以用有意义的字符串,甚至是别的对象作为传参,创建新的对象,用对象实例调用对象的属性和方法。例如,期望有下面的用法,见代码 5-2。

代码 5-2　Python 风格的使用 Pin 类模块的方法

```
import Pin
p0 = Pin('PA2', mode = Pin.OUT_PUSHPULL)      # 实例化一个 Pin 的对象,用名称指定引脚
p0.high()                                     # 通过 Pin 的实例化对象调用属性方法
p0(0)                                         # call()方法操作对象
led = Pin(p0)                                 # 用 Pin 对象实例化 Pin 对象
led.on()                                      # 通过 Pin 的实例化对象调用属性方法
led.off()                                     # 通过 Pin 的实例化对象调用属性方法
```

本章将借鉴 MicroPython 在其他微控制器平台的移植项目中创建新扩展类的做法,面向实际应用场景,以实现一个 Pin 类为例,说明在 MicroPython 项目中添加一个“原汁原味”的 Python 扩展类的操作方法。在描述过程中,同时会展示基于 MM32F3 微控制器平台的一些 MicroPython 编码规范。

5.1 新建硬件外设类模块框架

MicroPython 官方的开发文档中没有详细介绍如何具体实现 Pin 类，但通过阅读 MicroPython 中集成的微控制器平台上的实现代码，可以发现，各微控制器平台上的代码实现大体遵循一些设计惯例，实现的方法大同小异。这为在 MM32F3 微控制器平台上自行实现 Pin 类提供了非常有价值的参考，从而才有后续实现的各种外设模块的移植。这里描述的一些设计要点，很大一部分是从现存源代码中提取出的设计思路，而非原创。借此机会，将这些设计经验进行归纳总结，呈现给读者。而本书额外提出的一点点奇思妙想，更多的是体现在向 MM32F3 微控制器平台的具体移植过程中。

参考 MicroPython 官方开发文档的说明，所有硬件外设相关的类模块统一归属在 machine 类之下，作为其中的一个子类。例如，在 Python 代码中将会使用如下方式引用新创建的 Pin 类，见代码 5-3。

代码 5-3 从 machine 类中导入 Pin 子类

```
from machine import Pin
```

为了实现 Pin 类模块，按照惯例，在 ports/mm32f3 目录下创建 machine_pin.c 和 machine_pin.h 文件。另外，在 ports/mm32f3/boards/plus-f3270 目录下新建了一个 machine_pin_board_pins.c 文件，专用于存放于板子相关的引脚映射表。若是以后需要支持同样使用 MM32F3270 微控制器的其他板子，也会在其对应板子的目录下面有一个特定的 machine_pin_board_pins.c 文件，其中定义了绑定到这块板子具体的电路的引脚映射。

在 machine_pin.c 文件中，基于第 4 章介绍的基本的添加类模块的方法，添加了新增类模块的框架代码，定义了 machine_pin_type 类型。然后，向其中填充合适的代码，调用硬件相关驱动程序的 API 操作硬件等，从而实现 Pin 类访问硬件 GPIO 的功能。关于完整的 machine_pin.h/.c 文件的源代码，可见本书配套资源中的相关源文件，本章将会抽取其中关键部分的设计思路专门讲解。

在 machine_pin.c 文件中基本编写好 Pin 类模块的实现代码后，将会定义一个表示 Pin 类模块的类型对象 machine_pin_type，见代码 5-4。

代码 5-4 定义 Pin 类模块的类型对象 machine_pin_type

```
const mp_obj_type_t machine_pin_type =
{
    { &mp_type_type },
    .name        = MP_QSTR_Pin,
    .print       = machine_pin_obj_print,    /* __repr__(), which would be called by print
                                                (<ClassName>). */
    .call        = machine_pin_obj_call,     /* __call__(), which can be called as
                                                <ClassName>(). */
    .make_new    = machine_pin_obj_make_new, /* create new class instance. */
    .protocol    = &pin_pin_p,               /* to support virpin. */
    .locals_dict = (mp_obj_dict_t *)&machine_pin_locals_dict,
};
```

然后，需要在 ports/mm32f3/modmachine.c 文件中添加对 machine_pin_type 类型的引用，并且将 Pin 类模块作为子类加入到 machine 类的属性列表中，见代码 5-5。

代码 5-5　向 machine 类中添加 Pin 类模块

```
extern const mp_obj_type_t machine_pin_type;
…
STATIC const mp_rom_map_elem_t machine_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR__name__),              MP_ROM_QSTR(MP_QSTR_umachine) },
    …
    { MP_ROM_QSTR(MP_QSTR_Pin),                  MP_ROM_PTR(&machine_pin_type) },
    …
}
…
```

在 Makefile 中添加此处新增的 machine_pin.c 和 machine_pin_board_pins.c 文件，并且确保 GPIO 外设模块的驱动源程序 hal_gpio.c 文件已经被包含在 Makefile 文件中，见代码 5-6。

代码 5-6　更新 Makefile 添加 Pin 类模块的源文件

```
SRC_HAL_MM32_C += \
    $(MCU_DIR)/devices/$(CMSIS_MCU)/system_$(CMSIS_MCU).c \
    $(MCU_DIR)/drivers/hal_rcc.c \
    $(MCU_DIR)/drivers/hal_gpio.c \
…

SRC_BRD_MM32_C += \
    $(BOARD_DIR)/clock_init.c \
    $(BOARD_DIR)/pin_init.c \
    $(BOARD_DIR)/board_init.c \
    $(BOARD_DIR)/machine_pin_board_pins.c \
…

SRC_C += \
    main.c \
    modmachine.c \
    machine_pin.c \
    …
    $(SRC_HAL_MM32_C) \
    $(SRC_BRD_MM32_C) \
    $(SRC_MOD) \
    $(DRIVERS_SRC_C) \
…

# list of sources for qstr extraction
SRC_QSTR += modmachine.c \
            machine_pin.c \
            $(BOARD_DIR)/machine_pin_board_pins.c \
…
```

至此，Pin 类模块作为 machine 类的子模块，已经被添加到 MicroPython 的工程中。当编写完成 machine_pin.c 的实现代码之后，编译工程，创建可执行文件并下载到电路板，就可以在 MicroPython 中使用 Pin 类模块了。

5.2 定义 machine_pin_obj_t 结构

在 machine_pin.h 文件中，定义了表示 Pin 对象的结构体类型 machine_pin_obj_t，用于存放 Pin 对象的实例所需要保存的与本实例相关的所有信息，见代码 5-7。

代码 5-7 定义 Pin 对象实例结构体

```
/* Pin class instance configuration structure. */
typedef struct
{
    mp_obj_base_t base;       /* object base class. */
    qstr          name;       /* pad name. */
    GPIO_Type    * gpio_port; /* gpio instance for pin. */
    uint32_t      gpio_pin;  /* pin number. */
} machine_pin_obj_t;
```

通常情况下，在定义类模块的属性方法函数时，第一个传入参数就是表示当前类模块的实例化对象，为了方便 MicroPython 内核在上层以统一定义的接口调用这些属性方法函数，这个参数都以通用对象的类型进行声明，但实际上，在实现函数内部，会将这个对象的访问指针转换成各自类模块的“句柄”(Handler)，从而能够访问到该类的实例对象的内部信息。例如，此处的 machine_pin_obj_t 就定义了 Pin 对象实例的“句柄”。这里以 Pin 类中的一个属性方法实现的函数 machine_pin_high()为例，说明 machine_pin_obj_t 的作用，见代码 5-8。

代码 5-8 定义 Pin 类模块的属性方法函数 machine_pin_high()

```
/* pin.high() */
STATIC mp_obj_t machine_pin_high(mp_obj_t self_in)
{
    /* self_in is machine_pin_obj_t. */
    machine_pin_obj_t * pin = (machine_pin_obj_t *)self_in;
    GPIO_WriteBit(pin->gpio_port, 1u << pin->gpio_pin, 1u);

    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_1(machine_pin_high_obj, machine_pin_high);
```

当创建一个 Pin 类的实例化对象 pin 后，在 Python 脚本中使用 pin.high()语句时，Python 内核会调用底层的 machine_pin_high()函数。此时，就像执行回调函数一样，Python 内核会将表示当前 Pin 类对象实例 pin 的句柄作为参数传入 machine_pin_high()函数。machine_pin_high()函数内部将控制当前 pin 对应引脚输出高电平，这当然要调用硬件 GPIO 外设外设驱动程序中的 GPIO_WriteBit()函数，但仍需要拿到当前 pin 对应的 GPIO

端口号和引脚号并传入 GPIO_WriteBit() 才能正常工作。了解到这个传参过程，就可以知道，在 machine_pin_obj_t 结构体类型的定义中，必须要包含本 Pin 类对象实例的 GPIO 端口号和引脚号。同理，当在其他类模块所属的句柄类型中定义的属性字段，也多是基于在实现本类属性方法时的输入传参需求。

machine_pin_obj_t 结构体类型的第一个字段是 mp_obj_base_t base，它将会存放指向 machine_pin_type 类型对象的指针。这个字段可被用于在 Python 内核实现类型匹配的功能，前面已经描述过 MicroPython 如何用一个巧妙的方法实现对象实例的类型匹配。同时，Pin 类对象的实例通过 base 找到 Pin 类的类属性方法，然后把自己专属的实例属性信息作为传参，代入到公用的类属性方法中，最终让公用的类属性方法为这个具体的类实例服务。类似地，当后续创建其他的外设类模块时，在所定义的 machine_xxx_obj_t 结构体类型中，也会在首先包含 mp_obj_base_t base 字段，并将本类的类型对象存入其中。

定义 QSTR 字符串类型的 name 字段，将用于使用字符串对 Pin 对象进行索引。实际上，一个真实的 machine_pin_obj_t 结构体实例将会是如下内容，见代码 5-9。

代码 5-9　定义一个 Pin 类对象的实例

```
const machine_pin_obj_t pin_PE2 =
{
    .base = { &machine_pin_type },
    .name = MP_QSTR_PE2,
    .gpio_port = GPIOE,
    .gpio_pin = 2
};
```

5.3　在构造函数中实现返回实例化对象

当在 Python 脚本中调用“pin=Pin(...)”这样的语句时，会创建 Pin 类的一个实例 pin，之后就可以用这个实例调用本身归属类的类属性方法了。在很长一段时间里，作者都很好奇这个创建实例的过程是如何实现的。事实上，当通过这种语句创建类实例时，Python 内核会在内部调用注册在对应类型对象中 make_new 字段的函数并返回一个类实例的属性句柄，而此处在 Pin 模块的类对象 machine_pin_type 中注册的，正是 machine_pin_obj_make_new()函数，它返回一个 machine_pin_obj_t 类型的结构体实例，见代码 5-10。

代码 5-10　Pin 类对象的实例化方法函数 machine_pin_obj_make_new()

```
/* return an instance of machine_pin_obj_t when calling pin = Pin(...). */
mp_obj_t machine_pin_obj_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw,
const mp_obj_t *args)
{
    mp_arg_check_num(n_args, n_kw, 1, MP_OBJ_FUN_ARGS_MAX, true);

    const machine_pin_obj_t *pin = pin_find(args[0]);

    if ( (n_args > 1) || (n_kw > 0) )
```

```
        {
            mp_map_t kw_args;
            mp_map_init_fixed_table(&kw_args, n_kw, args + n_args);
                        /* 将关键字参数从总的参数列表中提取出来,单独封装成 kw_args. */
            machine_pin_obj_init_helper(pin, n_args - 1, args + 1, &kw_args);
        }

        return (mp_obj_t)pin;
    }
```

make_new()函数还是一个回调函数,它的传参是 MicroPython 内核预先定义好的固定模式:

- type 是本对象的类型,可用于在函数内部为新实例分配内存后填充其 base 字段。但在当前的实现中完全没有用到,在应用时,使用静态内存存放预先分配好的 Pin 实例列表,并在其中填充了 type。
- n_args、n_kw 和 args,都是用于描述一个不定长的参数列表。其中 n_args 表示参数列表的总数量,n_kw 表示关键字参数的数量,args 就是参数列表中的各参数内容了。

make_new()函数传入的不定长参数数组分为两个部分:固定位置参数和关键字参数。固定位置参数将会被按照顺序存放在参数数组的特定位置上,且必须由上层调用者提供;而关键字参数是可选提供的,如果上层调用者没有传入,那么在下层的程序中会使用一个预设的默认值。具体解析关键字参数的过程位于 machine_pin_obj_init_helper()函数中。此处的 mp_arg_check_num()函数专门用于验证传入参数数组的数量是否符合预设的要求。py/runtime.h 文件中有 mp_arg_check_num()函数的实现,见代码 5-11。

代码 5-11　runtime.h 文件中的 mp_arg_check_num()函数

```
static inline void mp_arg_check_num(size_t n_args, size_t n_kw, size_t n_args_min, size_t
n_args_max, bool takes_kw) {
    mp_arg_check_num_sig(n_args, n_kw, MP_OBJ_FUN_MAKE_SIG(n_args_min, n_args_max, takes_kw));
}
```

可以看出,machine_pin_obj_make_new()函数内部对参数验证的要求是:

- 总参数数量最少为 1(n_args_min=1);
- 总参数数量最大为 MP_OBJ_FUN_ARGS_MAX(n_args_max=MP_OBJ_FUN_ARGS_MAX);
- 在参数数组中允许关键字参数存在(takes_kw=true)。

至于 mp_arg_check_num()函数内部调用的 mp_arg_check_num_sig()函数,其定义位于 argcheck.c 文件中,这个函数内部将传入的参数数组及其限制条件进行比对,当出现不匹配的情况时,会通过 mp_raise_TypeError() 函数报错,这导致 REPL 可能会输出如下的报错信息之一:

- mp_raise_TypeError(MP_ERROR_TEXT("function doesn't take keyword arguments"));

- mp_raise_msg_varg(&mp_type_TypeError, MP_ERROR_TEXT("function takes %d positional arguments but %d were given"), n_args_min, n_args);
- mp_raise_msg_varg(&mp_type_TypeError, MP_ERROR_TEXT("function missing %d required positional arguments"), n_args_min - n_args);
- mp_raise_msg_varg(&mp_type_TypeError, MP_ERROR_TEXT("function expected at most %d arguments, got %d"), n_args_max, n_args)。

make_new()函数原本是要创建一个 Pin 类对象的实例，涉及从对堆存储空间中分配出一块存储区，向其中填充必要的属性信息(包括本类的通用类属性信息等)，还有一些根据参数数组传入的信息，最终完成对这个对象的初始化工作。但在此处设计实现同底层硬件相关的 Pin 类模块，结合微控制器系统的特性，在最终实现的程序设计中进行了一些改变：

- 考虑到嵌入式系统的内存资源有限和系统运行时避免出现内存溢出的情况，使用静态内存预分配的方式取代了动态分配内存分配的过程。实际上本章为微控制器芯片上的每一个引脚都对应预先定义了各自的 Pin 实例，并且用 const 关键字将它们对应的存储空间映射到 Flash 中而非 SRAM 中，微控制器芯片内部集成的 Flash 存储空间远大于 SRAM，存放在 Flash 中的常量在编译时就会参与计算代码大小，可以预判内存是否够用。因此，消除了因为在运行时动态分配内存可能造成内存溢出的风险。
- make_new()函数内部实现的初始化实例对象的操作，不仅要初始化内存(实际上在预分配 Pin 对象结构体中已经填好信息了)，还要完成对硬件外设的初始化配置，毕竟这是一个与硬件外设相关的类模块。但此处，未将同硬件相关的操作直接在 make_new()函数中展开，而是将对硬件初始化的操作单独封装成 machine_pin_obj_init_helper()函数，是因为 Pin 类还实现了一个 init()方法，这个 init()也是实现初始化硬件的操作，其内部的实现也将复用 machine_pin_obj_init_helper()对硬件进行初始化。

machine_pin_obj_init_helper()函数不仅实现了对底层外设的初始化，还承担了大部分的解析参数列表的工作。在 make_new()函数中，使用 mp_arg_check_num()函数对参数数组的有效性进行查验，通过 mp_map_init_fixed_table()提取参数数组的关键字参数部分，最后在 machine_pin_obj_init_helper()函数中完成对参数的解析，并对硬件进行初始化。这里具体看一下 helper()函数是如何解析关键字参数的，见代码 5-12。

代码 5-12　解析实例化参数的 machine_pin_obj_init_helper()函数

```
typedef enum
{
    PIN_INIT_ARG_MODE = 0,
    PIN_INIT_ARG_VALUE,
    PIN_INIT_ARG_AF,
} machine_pin_init_arg_t;

STATIC mp_obj_t machine_pin_obj_init_helper(
    const machine_pin_obj_t *self,  /* machine_pin_obj_t 类型的变量,包含硬件信息 */
    size_t n_args,                  /* 位置参数数量 */
```

```
    const mp_obj_t * pos_args,          /* 位置参数清单 */
    mp_map_t * kw_args )                /* 关键字参数清单结构体 */
{
    static const mp_arg_t allowed_args[] =
    {
        [PIN_INIT_ARG_MODE] { MP_QSTR_mode , MP_ARG_REQUIRED | MP_ARG_INT, {.u_int =
PIN_MODE_IN_PULLUP} },
        [PIN_INIT_ARG_VALUE]{ MP_QSTR_value, MP_ARG_KW_ONLY  | MP_ARG_OBJ, {.u_obj =
MP_OBJ_NULL} },
        [PIN_INIT_ARG_AF]   { MP_QSTR_af   , MP_ARG_KW_ONLY  | MP_ARG_INT, {.u_int = 0}},
    };

    /* 解析参数 */
    mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
    mp_arg_parse_all(n_args, pos_args, kw_args, MP_ARRAY_SIZE(allowed_args), allowed_args,
args);

    /* 配置硬件 */
    GPIO_Init_Type gpio_init;
    gpio_init.Speed = GPIO_Speed_50MHz;
    gpio_init.Pins = (1u << self->gpio_pin);
    gpio_init.PinMode = machine_pin_modes[args[PIN_INIT_ARG_MODE].u_int];
    GPIO_Init(self->gpio_port, &gpio_init);

    if (args[PIN_INIT_ARG_MODE].u_int < PIN_MODE_AF_OPENDRAIN)
    {
        if (args[PIN_INIT_ARG_VALUE].u_obj != MP_OBJ_NULL)
        {
            if ( mp_obj_is_true(args[PIN_INIT_ARG_VALUE].u_obj) )
            {
                GPIO_WriteBit(self->gpio_port, 1u << self->gpio_pin, 1u);
            }
            else
            {
                GPIO_WriteBit(self->gpio_port, 1u << self->gpio_pin, 0u);
            }
        }
    }
    else
    {
        GPIO_PinAFConf(self->gpio_port, 1u << self->gpio_pin, (uint8_t)(args[PIN_INIT_
ARG_AF].u_int));
    }
    return mp_const_none;
}
```

在解析实例化参数的过程中，定义了一个关键字参数的匹配数组 allowed_args 参数列表，其中定义了关键字参数的名字（用到了 QSTR 字符串）、类型以及默认值。这里专门看一下 mp_arg_t 类型的定义，位于 runtime.h 文件中，见代码 5-13。

代码 5-13　runtime.h 文件中定义的 mp_arg_t 类型

```
typedef enum {
    MP_ARG_BOOL       = 0x001,
    MP_ARG_INT        = 0x002,
    MP_ARG_OBJ        = 0x003,
    MP_ARG_KIND_MASK  = 0x0ff,
    MP_ARG_REQUIRED   = 0x100,
    MP_ARG_KW_ONLY    = 0x200,
} mp_arg_flag_t;

typedef union _mp_arg_val_t {
    bool u_bool;
    mp_int_t u_int;
    mp_obj_t u_obj;
    mp_rom_obj_t u_rom_obj;
} mp_arg_val_t;

typedef struct _mp_arg_t {
    uint16_t qst;
    uint16_t flags;
    mp_arg_val_t defval;
} mp_arg_t;
```

对于 mp_arg_t 结构体中的 flags 字段，可使用的类型定义在枚举类型 mp_arg_flag_t 中，可以使用 MP_ARG_BOOL、MP_ARG_INT 或 MP_ARG_OBJ 指定关键字参数的基本数据类型，同时还可以使用 MP_ARG_REQUIRED 或 MP_ARG_KW_ONLY 指定这个参数是否为必须提供或者是否仅作为关键字参数。在实际使用的时候，可以由多个 flags 选项相或，产生叠加作用。

mp_arg_t 结构体中的 defval 字段可用于指定关键字参数的默认值，其使用了 mp_arg_val_t 类型，用 union 定义，可以使用多种类型中的一种。

之后，通过 mp_arg_parse_all()函数将传入参数数组中的关键字参数的值解析出来，见代码 5-14。

代码 5-14　调用 mp_arg_parse_all()函数解析实例化参数

```
/* 解析参数 */
mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
mp_arg_parse_all(n_args, pos_args, kw_args, MP_ARRAY_SIZE(allowed_args), allowed_args, args);
```

解析出的值按照 allowed_args 参数列表中定义的顺序存放于 args 数组中。例如，若用户在实例化 Pin 类对象时传入了“af=3”作为参数列表的一部分，那么在 helper() 函数中，由 MP_QSTR_af 指定的参数位于 allowed_args 数组的第二个(从零开始数)位置，那么在保存解析出的参数的数组 args 中，args[2]的值将会是 2。如果未在参数列表中指定这个 af 的值，那么在后续的解析过程结束后引用的 args[2]的值将会是在定义 allowed_args 数组时指定的默认值 0。

再之后的程序，就是根据传入的参数，通过 SDK 的 API 配置硬件实现对应功能。

5.4 在构造函数中实现多种传参方式指定实例化对象

在 C 语言的程序中，要求传入函数的参数必须为某一个确定的类型，但在 MicroPython 中，允许使用多种不同的类型作为类实例化函数的参数，例如，可以通过引脚名、数字编号，甚至一个已有的实例化对象，都可以实例化一个新的 Pin 类实例。如下代码显示了不同的实例化 Pin 类对象的方法，见代码 5-15。

代码 5-15 实例化 Pin 类对象的多种方法

```
pin1 = Pin('PA1')
pin2 = Pin(7)
pin3 = Pin(pin1)
```

这个“多类型传参”的机制曾让作者百思不得其解，但在研读代码之后，便豁然开朗。实现这个机制的关键在于 pin_find()函数，是对前文所述的 MicroPython 的类型匹配机制的一种具体应用，见代码 5-16。

代码 5-16 在 pin_find()函数中匹配多种类型的输入参数

```
/* 格式化 pin 对象，传入参数无论是已经初始化好的 pin 对象，还是一个表示 pin 清单中的索引编号，通过本函数都返回一个期望的 pin 对象 */
const machine_pin_obj_t *pin_find(mp_obj_t user_obj)
{
    /* 如果传入参数本身就是一个 Pin 的实例，则直接送出这个 pin */
    if ( mp_obj_is_type(user_obj, &machine_pin_type) )
    {
        return user_obj;
    }

    /* 如果传入参数是一个代表 Pin 清单的索引，则通过索引在 Pin 清单中找到并送出这个 pin */
    if ( mp_obj_is_small_int(user_obj) )
    {
        uint8_t pin_idx = MP_OBJ_SMALL_INT_VALUE(user_obj);
        if ( pin_idx < machine_pin_board_pins_num)
        {
            return machine_pin_board_pins[pin_idx];
        }
    }

    /* 如果传入参数是一个字符串，则通过这个字符串在 Pin 清单中匹配引脚名字，然后送出找到的 pin */
    const machine_pin_obj_t *named_pin_obj = pin_find_by_name(&machine_pin_board_pins_locals_dict, user_obj);
    if ( named_pin_obj )
    {
```

```
            return named_pin_obj;
        }

        mp_raise_ValueError(MP_ERROR_TEXT("Pin doesn't exist"));
    }

    /* 通过字符串在引脚清单中匹配引脚 */
    const machine_pin_obj_t *pin_find_by_name(const mp_obj_dict_t *name_dict, mp_obj_t name)
    {
        mp_map_t *name_map = mp_obj_dict_get_map((mp_obj_t)name_dict);
        mp_map_elem_t *name_elem = mp_map_lookup(name_map, name, MP_MAP_LOOKUP);

        if ( (name_elem != NULL) && (name_elem->value != NULL) )
        {
            return name_elem->value;
        }
        return NULL;
    }
```

make_new()函数会传入用户在实例化类对象时指定的标识引脚的参数，其位于参数列表 args(包含固定位置参数和关键字参数的整个列表)中的第一个位置。这个 args[0]被传入 pin_find()函数后，返回一个与标识相关的引脚对象。在 pin_find()函数内部，首先把 args[0]作为一个 MicroPython 内部的对象实体，判定其对象类型。在 obj.h 文件中有关于如何判定这些类型的定义，见代码 5-17。

代码 5-17　在 obj.h 文件中定义判定整数类型的函数

```
#define mp_obj_is_type(o, t) (mp_obj_is_obj(o) && (((mp_obj_base_t *)MP_OBJ_TO_PTR(o))->
type == (t)))
...
static inline bool mp_obj_is_small_int(mp_const_obj_t o) {
    return (((mp_int_t)(o)) & 1) != 0;
}
#define MP_OBJ_SMALL_INT_VALUE(o) (((mp_int_t)(o)) >> 1)
#define MP_OBJ_NEW_SMALL_INT(small_int) ((mp_obj_t)((((mp_uint_t)(small_int)) << 1) | 1))
...
```

这里的 mp_obj_is_type()比较容易理解。前文曾提到过，Python 中一切皆对象，每个对象的 type 字段所引用的一个类型对象结构体可用于表示该对象的类型，比较这个 type 指针的值，就可以匹配该对象实例的类型。

small_int 类型的判定稍显复杂，但实际是为了让 MicroPython 以更简单的方式判定 small_int 类型。从代码中可以看出，MicroPython 中表示 small_int 类型的变量将变量值整体左移 1 位后，在末位填 1。这样，在判定 small_int 类型时，只要检查末位是否为 1 即可，但如果要提取其中的值，就需要使用宏函数 MP_OBJ_SMALL_INT_VALUE() 处理一下，实际上，就是把保存的内容再右移 1 位返回。

判定标识参数的类型之后，就分别对应处理了。但一个总体的原则是，处理的结果必须

返回一个 machine_pin_obj_t 类型的对象实例。

- 如果判定当前传入的标识参数本身就是一个 Pin 类实例，则直接返回传入对象。
- 如果判定当前传入的标识参数是一个数字，就是用这个数字作为索引，在预先准备好的 machine_pin_obj_t 对象数组进行索引。这个对象数组中的顺序可由开发者自己定义，方便使用即可。在 Pin 类对象的预分配实例化数组里，是按照芯片封装上的引脚号排序的。也可以根据用户自己设计的电路板引出信号排序，或者按硬件电路的一些规律排序，实际上，在后续其他模块的设计中，就是按照电路板资源进行排序的。例如，在 machine_pin_board_pins.c 文件中定义的 machine_pin_board_pins[] 数组的内容，见代码 5-18。

代码 5-18　在 machine_pin_board_pins.c 文件中预定义 Pin 类实例化对象

```
...
const machine_pin_obj_t pin_PE2 = { .base = { &machine_pin_type }, .name = MP_QSTR_PE2,
.gpio_port = GPIOE, .gpio_pin = 2 };
const machine_pin_obj_t pin_PE3 = { .base = { &machine_pin_type }, .name = MP_QSTR_PE3,
.gpio_port = GPIOE, .gpio_pin = 3 };
const machine_pin_obj_t pin_PE4 = { .base = { &machine_pin_type }, .name = MP_QSTR_PE4,
.gpio_port = GPIOE, .gpio_pin = 4 };
const machine_pin_obj_t pin_PE5 = { .base = { &machine_pin_type }, .name = MP_QSTR_PE5,
.gpio_port = GPIOE, .gpio_pin = 5 };
const machine_pin_obj_t pin_PE6 = { .base = { &machine_pin_type }, .name = MP_QSTR_PE6,
.gpio_port = GPIOE, .gpio_pin = 6 };
const machine_pin_obj_t pin_PC13 = { .base = { &machine_pin_type }, .name = MP_QSTR_PC13,
.gpio_port = GPIOC, .gpio_pin = 13 };

...
/* pin id in the package. */
const machine_pin_obj_t* machine_pin_board_pins[] =
{
    &pin_PE2,
    &pin_PE3,
    &pin_PE4,
    &pin_PE5,
    &pin_PE6,
    NULL, /* VBAT */
    &pin_PC13,
    ...
};

STATIC const mp_rom_map_elem_t machine_pin_board_pins_locals_dict_table[] =
{
    { MP_ROM_QSTR(MP_QSTR_PE2), MP_ROM_PTR(&pin_PE2) },
    { MP_ROM_QSTR(MP_QSTR_PE3), MP_ROM_PTR(&pin_PE3) },
    { MP_ROM_QSTR(MP_QSTR_PE4), MP_ROM_PTR(&pin_PE4) },
    { MP_ROM_QSTR(MP_QSTR_PE5), MP_ROM_PTR(&pin_PE5) },
    { MP_ROM_QSTR(MP_QSTR_PE6), MP_ROM_PTR(&pin_PE6) },
    { MP_ROM_QSTR(MP_QSTR_PC13), MP_ROM_PTR(&pin_PC13) },
```

```
    ...
};

MP_DEFINE_CONST_DICT(machine_pin_board_pins_locals_dict, machine_pin_board_pins_locals_
dict_table);
```

- 如果不是已有的 Pin 类实例，也不是数字编号，那么就被当成字符串，送入 pin_find_by_name()函数，同 machine_pin_board_pins_locals_dict 中的 QSTR 进行匹配。在 pin_find_by_name()函数中，如果找到匹配 QSTR 的记录，那么同直接用编号在 machine_pin_board_pins[]执行索引一样，也能映射到一个预先定义好的 Pin 类对象实例，返回给 pin_find()，再向上返回到 make_new()。

5.5　print()和 call()

print()函数和 call()函数是 Python 的通用类属性函数，与操作底层硬件无关，也不影响开发者新创建的类属性方法。但是，考虑它们是 Python 语法现象的一部分，本例还是在 Pin 类的创建过程中实现了这两个方法。

5.5.1　print()方法

Python 类中一个特殊的实例方法，即__ repr __()。该方法用于显示类属性，当通过 print()函数打印一个类对象时，输出该类的属性信息。在 MicroPython 中，当使用 Pin 类时，可以通过 print()打印出一个具体的 Pin 类对象实例的属性信息，见代码 5-19。

代码 5-19　在 Python 中使用 print()方法(a)

```
from machine import Pin

pin1 = Pin('PA2', mode = OUT_PUSHPULL)
print(pin1)
```

此时，在 REPL 的终端中会输出引脚的信息，见代码 5-20。

代码 5-20　在 Python 中使用 print()方法(b)

```
Pin(PA2)
```

这个功能的实现，对应于向 machine_pin_type 中的 call 字段注册 machine_pin_obj_print 函数对象，而 machine_pin_obj_print 函数对象的实现也位于 machine_pin.c 中，见代码 5-21。

代码 5-21　Pin 类对象 print()方法的实现函数

```
/* print(pin). */
STATIC void machine_pin_obj_print(const mp_print_t *print, mp_obj_t o, mp_print_kind_t kind)
{
```

```
    /* o is the machine_pin_obj_t. */
    (void)kind;
    const machine_pin_obj_t *self = MP_OBJ_TO_PTR(o);
    mp_printf(print, "Pin(%s)", qstr_str(self->name));
}
```

当然,这里还可以由开发者自定义,输出显示更多的类属性信息。

5.5.2 call()方法

Python 类中一个特殊的实例方法,即__call__()。该方法的功能类似于类中的重载运算符“()”,使得可以像调用普通函数那样,以“<对象名>()”的形式使用类实例对象。在 MicroPython 中,当使用 Pin 类时,可以通过 pin1()的方式操作引脚,见代码 5-22。

代码 5-22 使用 Pin 对象的 call()方法

```
from machine import Pin

pin1 = Pin('PA2', mode = OUT_PUSHPULL)
pin1(0) # PA2 output low voltage level.
pin1(1) # PA2 output high voltage level.
pin1.value(0) # PA2 output low voltage level.
pin1.value(1) # PA2 output high voltage level.
```

这个功能的实现,对应于 machine_pin_type 中的 call 字段注册 machine_pin_obj_call 函数对象,而 machine_pin_obj_call 函数对象的实现也位于 machine_pin.c 文件中,见代码 5-23。

代码 5-23 Pin 类对象 call()方法的实现函数

```
/* pin.value(val). */
STATIC mp_obj_t machine_pin_value(size_t n_args, const mp_obj_t *args)
{
    /* args[0] is machine_pin_obj_t. */
    return machine_pin_obj_call(args[0], (n_args - 1), 0, args + 1);
}
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(machine_pin_value_obj, 1, 2, machine_pin_value);

/* pin(val). */
STATIC mp_obj_t machine_pin_obj_call(mp_obj_t self_in, mp_uint_t n_args, mp_uint_t n_kw,
const mp_obj_t *args)
{
    /* self_in is machine_pin_obj_t. */
    mp_arg_check_num(n_args, n_kw, 0, 1, false);
    machine_pin_obj_t *self = self_in;

    if ( n_args == 0 )
    {
        //return MP_OBJ_NEW_SMALL_INT(mp_hal_pin_read(self));
```

```
        return MP_OBJ_NEW_SMALL_INT(GPIO_ReadInDataBit(self->gpio_port, 1u << self->
gpio_pin) ? 1u: 0u);
    }
    else
    {
        //mp_hal_pin_write(self, mp_obj_is_true(args[0]));
        GPIO_WriteBit(self->gpio_port, 1u << self->gpio_pin, mp_obj_is_true(args[0]) ?
1u : 0u);
        return mp_const_none;
    }
}
```

这里考虑到 machine_pin_value()函数和 machine_pin_obj_call()都会操作引脚，为了确保对硬件操作的一致性，使用 machine_pin_obj_call()函数直接操作硬件，而 machine_pin_value()函数调用 machine_pin_obj_call()函数间接操作硬件。

5.5.3　其他基础类属性函数

既然能向 machine_pin_type 注册__repr__()和__call__()函数，可以想见，还有更多的基础类属性函数可以接受注册。在 obj.h 文件中找到_mp_obj_type_t 结构体的定义，果然，除了 make_new()、print()、call()，还有 getiter 对应__iter__()，iternext 对应__next__()等等，但大多嵌入式应用相关性不强，因此本例没有实现。关于_mp_obj_type_t 结构体类型的定义，位于 obj.h 文件中，见代码 5-24。

代码 5-24　obj.h 文件中定义的_mp_obj_type_t 结构体类型

```
struct _mp_obj_type_t {
    // A type is an object so must start with this entry, which points to mp_type_type.
    mp_obj_base_t base;

    // Flags associated with this type.
    uint16_t flags;

    // The name of this type, a qstr.
    uint16_t name;

    // Corresponds to __repr__ and __str__ special methods.
    mp_print_fun_t print;

    // Corresponds to __new__ and __init__ special methods, to make an instance of the type.
    mp_make_new_fun_t make_new;

    // Corresponds to __call__ special method, ie T(...).
    mp_call_fun_t call;

    // Implements unary and binary operations.
    // Can return MP_OBJ_NULL if the operation is not supported.
    mp_unary_op_fun_t unary_op;
```

```
    mp_binary_op_fun_t binary_op;

    // Implements load, store and delete attribute.
    //
    // dest[0] = MP_OBJ_NULL means load
    //  return: for fail, do nothing
    //            for attr, dest[0] = value
    //            for method, dest[0] = method, dest[1] = self
    //
    // dest[0,1] = {MP_OBJ_SENTINEL, MP_OBJ_NULL} means delete
    // dest[0,1] = {MP_OBJ_SENTINEL, object} means store
    //  return: for fail, do nothing
    //            for success set dest[0] = MP_OBJ_NULL
    mp_attr_fun_t attr;

    // Implements load, store and delete subscripting:
    //   - value = MP_OBJ_SENTINEL means load
    //   - value = MP_OBJ_NULL means delete
    //   - all other values mean store the value
    // Can return MP_OBJ_NULL if operation not supported.
    mp_subscr_fun_t subscr;

    // Corresponds to __iter__ special method.
    // Can use the given mp_obj_iter_buf_t to store iterator object,
    // otherwise can return a pointer to an object on the heap.
    mp_getiter_fun_t getiter;

    // Corresponds to __next__ special method.  May return MP_OBJ_STOP_ITERATION
    // as an optimisation instead of raising StopIteration() with no args.
    mp_fun_1_t iternext;

    // Implements the buffer protocol if supported by this type.
    mp_buffer_p_t buffer_p;

    // One of disjoint protocols (interfaces), like mp_stream_p_t, etc.
    const void *protocol;

    // A pointer to the parents of this type:
    //   - 0 parents: pointer is NULL (object is implicitly the single parent)
    //   - 1 parent: a pointer to the type of that parent
    //   - 2 or more parents: pointer to a tuple object containing the parent types
    const void *parent;

    // A dict mapping qstrs to objects local methods/constants/etc.
    struct _mp_obj_dict_t *locals_dict;
};
```

应特别注意，protocol 字段是使用 stream 流传输模型专用的字段，当实现与 UART 或 SPI 类似的串行通信外设类时，会按照流模型的设计要求，简单实现发送单个数据单元的函

数注册到流模型实例，然后再将流模型实例注册到对象类中，就可以使用 write()、read()等的标准读写操作访问硬件外设了。关于流模型的工作机制和设计方法，在后续章节中会详细讲解。

5.6　实验

重新编译 MicroPython 项目，创建 firmware.elf 文件，并下载到 PLUS-F3270 开发板。

在计算机上启用终端软件，配置成 UART 串口通信，使用 115 200bps 的波特率。复位开发板，在终端里与 MicroPython 的 REPL 通信，执行 Python 脚本。

5.6.1　向引脚输出电平控制小灯亮灭

本实验将验证 Pin 模块能够正常工作，通过输出电平控制开发板上小灯的亮灭。

PLUS-F3270 开发板上设计了 LED 小灯的电路，如图 5-1 所示。

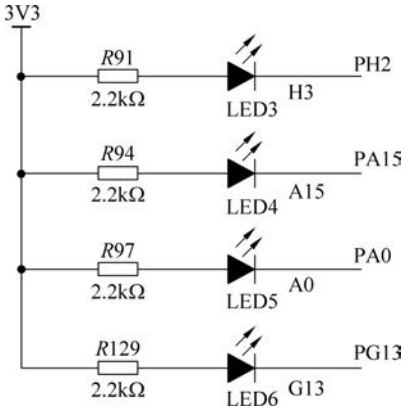

图 5-1　LED 小灯电路原理图

PH2、PA15、PA0、PG13 引脚控制 4 个 LED 小灯。在 REPL 中输入 Python 脚本，先试着点亮 PH2 引脚控制的 LED 小灯，见代码 5-25。

代码 5-25　编写 Python 脚本控制 LED 小灯亮灭

```
>>> from machine import Pin
>>> led0 = Pin('PH2', mode = Pin.OUT_PUSHPULL, value = 1)
>>> dir(led0)
['value', 'AF_OPENDRAIN', 'AF_PUSHPULL', 'IN_ANALOG', 'IN_FLOATING', 'IN_PULLDOWN', 'IN_PULLUP',
'OUT_OPENDRAIN', 'OUT_PUSHPULL', 'high', 'init', 'low']
>>> led0(0)
>>> led0(1)
>>> led0.low()
>>> led0.high()
>>>
```

其中，

- 首先导入 machine 类中的 Pin 子类。

- 之后创建了 Pin 类模块的实例化对象 led0，并绑定到 PH2 引脚上，配置其为推挽输出模式，指定初值为 1。
- 用 dir()命令查看对象 led0 的属性方法。此时可以看到 Pin 类的各种属性常量，例如 OUT_PUSHPULL，以及属性方法，例如 high 和 low。
- 通过对象名方法，先使用 led0(0)指定 led0 的输出为 0，可以观察到开发板上 LED3 小灯亮；再使用 led0(1)指定 led0 的输出为 1，可以观察到开发板上的 LED3 小灯灭。
- 通过 Pin 类模块的属性方法，先使用 led0.low()指定 led0 的输出为 0，可以观察到开发板上 LED3 小灯亮；再使用 led0.high()指定 led0 的输出为 1，可以观察到开发板上的 LED3 小灯灭。

5.6.2 读取引脚电平获取按键值

本实验进一步验证 Pin 模块能够正常工作，通过读取开发板上的按键输入电平信号，再控制输出电平控制开发板上小灯的亮灭。

PLUS-F3270 开发板上设计了独立按键的电路，如图 5-2 所示。

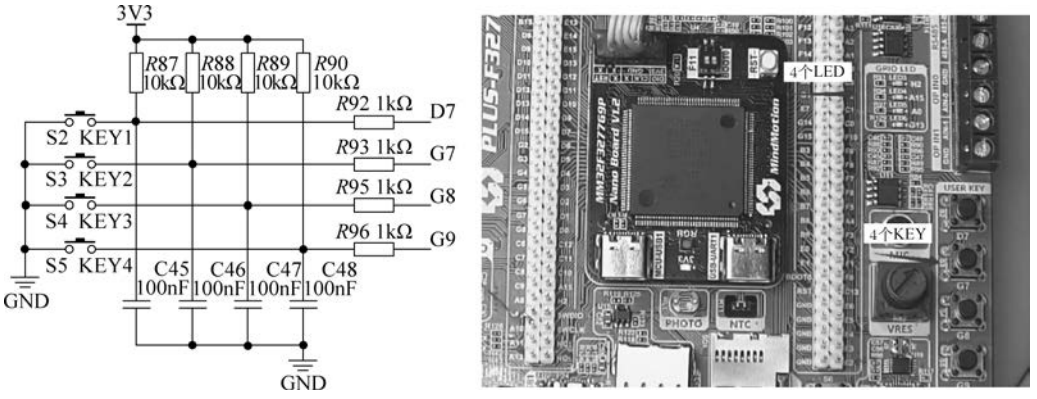

图 5-2 按键电路原理图和实物图

PD7、PG7、PG8、PG9 引脚连接 4 个按键。本实验将使用按键 PD7 控制上述实验中的 led0。在 REPL 中继续输入 Python 脚本，使用 key0 控制 led0，见代码 5-26。

代码 5-26 编写 Python 脚本使用按键控制 LED 小灯亮灭

```
>>> key0 = Pin('PD7', mode = Pin.IN_FLOATING)
>>> while True:
      led0(key0())
      ...
```

其中：

- 创建了一个 Pin 类对象 key0，并绑定到引脚 PD7 上，指定其工作模式为输入。
- 在一个无限循环中，不断轮询 key0()方法的值，作为控制前文例子中创建 led0 的输出值。

之后，随时按下连接 PD7 引脚的按键，LED3 小灯亮；松开按键，LED3 小灯灭。

注意，因为程序执行进入无限循环，已经不再返回 REPL，必须通过复位开发板，才能重启 REPL 接收新的命令。

5.7　本章小结

Pin 类是本书讲述移植 MicroPython 实现的第一个操作硬件外设的类（在移植 minimal 时向 REPL 适配 UART 的过程是通过函数映射实现的，不是通过定义类实现的），通过 Pin 类，可以真正在 MicroPython 中对硬件进行编程，通过在 REPL 中输入 Python 脚本控制小灯亮灭以展示 Python 语言可以控制硬件电路的现象，这确实是开发 MicroPython 的一个里程碑式的成果。但同时，通过设计和实现第一个实现操作硬件外设的类，已经建立了在 MicroPython 中设计外设硬件相关类模块的框架，摸索出了一套标准开发流程。后续在设计更多硬件外设类模块时，大体都遵循 Pin 类的基本设计规范。

在本章中，通过阅读源代码，分析了一些关键技术点的实现原理：

- 定义 machine_pin_obj_t 结构体用于表示类对象，在其中定义的字段用于表示该对象实例的私有属性信息。如果需要在实例化（初始化）一个类对象的过程中向该对象写入一些属性或者创建状态信息，或者在自定义的类方法需要使用类实例内部的信息参与操作，则在 machine_pin_obj_t 结构体中为它定义一个字段。
- 在创建新对象时，使用静态存储取代动态内存分配，向 Python 用户隐藏了配置引脚的更多烦琐的技术细节，同时规避了动态内存分配可能在运行时产生的内存溢出的风险。在 Flash 中存放预先填好实例私有属性信息的对象，存放成对象数组，用查询过程代替动态分配过程，在程序执行效率上也有一定的提升。
- 实现了通过字符串（QSTR）、引脚编号以及已有的 Pin 对象等多种标识方式创建新的 Pin 对象实例。这里用到了“Python 中一切皆对象”的思想及其实现机制，通过匹配结构体中 type 字段值的方式，判定传入参数的对象类型，进而能够分别处理。通过这个设计向用户开放了非常灵活的调用接口，并且为后续设计使用对象作为函数传参的实现提供了范例。

虽然目前只有一个 Pin 类模块用来访问硬件外设，但已经可以开始在微控制器上实现很多有趣的设计了。