

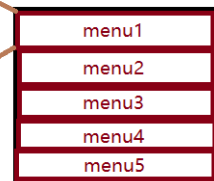
C 语言架构设计

作者:向仔州

单片机菜单架构.....	2
单片机按键架构.....	8
定时器多任务框架.....	13
C 语言实现观察者模式(Observer).....	15
C 语言实现简单工厂模式.....	17
C 语言实现代理模式(Proxy).....	18
C 语言实现命令模式.....	21
C 语言实现装饰器模式, 达到多态效果.....	25
C 语言路由架构.....	26
C 语言实现有限状态机(FSM).....	32
EasyFlash 存储库使用(未完成).....	36

单片机菜单架构

```
struct MenuItem
{
    short MenuCount; | //当前菜单项目数
    char *DisplayString; | //当前项目要显示得字符
    void(*subs)(); | //某一项菜单被选中，执行的功能函数
    struct MenuItem *ChildrenMenus; //当前项目的下级子菜单
    struct MenuItem *ParentMenus; //当前项目上级父菜单
};
```



一行菜单内容如下:

DisplayString: 包含了本菜单项自己的字符显示,

MenuCount: 本菜单项属于当前LCD屏幕的哪一行?

subs: 本菜单项被点击后执行的回调函数

ChildrenMenus: 本菜单点击后, 会不会有子菜单, 如果有子菜单, 将子菜单结构体赋给该指针, 如果没有子菜单填NULL

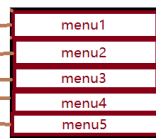
ParentMenus: 本菜单是否是子菜单, 如果是子菜单, 那么久填入上一级父菜单, 如果不是子菜单, 填NULL

我们用这个结构体可以定义主菜单, 也可以用来定义子菜单

```
/*主菜单定义*/
struct MenuItem MainMenu[5] =
{
    { 5, (char *) "menu1", NULL, NULL, NULL },
    { 5, (char *) "menu2", NULL, NULL, NULL },
    { 5, (char *) "menu3", NULL, NULL, NULL },
    { 5, (char *) "menu4", NULL, NULL, NULL },
    { 5, (char *) "menu5", NULL, NULL, NULL },
};
```

主菜单定义5项

注意: 一个界面有5个菜单项,
那么这里一定要填5,
因为这是菜单计数器MenuCount



LCD

```
/*子菜单*/
struct MenuItem Submenu1[3] =
{
    { 3, (char *) "submenu1", NULL, NULL, MainMenu },
    { 3, (char *) "submenu2", NULL, NULL, MainMenu },
    { 3, (char *) "submenu3", NULL, NULL, MainMenu },
};
```

子菜单界面只定义了3项,
那么菜单计数器就填3

子菜单定义3项

因为这是主菜单的子菜单, 所以
要填入主菜单地址, 指定该子菜
单属于哪个主菜单, 后期循环操
作方便调用

菜单定义的数据结构已经讲完, 下面看看怎么调用这些数据结构

```
struct MenuItem *MenuPoint = MainMenu; //开机后当前菜单地址, 一般都是主菜单 第1步
short selectItem; | //当前选择的第几行菜单
```

```
int main(void)
{
    MainMenu[0].ChildrenMenus = Submenu1; //第2步
    //初始化的时候, 要让主界面指定每一项的子菜单, 我这里指定主界面第1项子菜单为Submenu1结构
    //比如又定义了一个子菜单结构Submenu2给主界面第2项
    //那么初始化增加MainMenu[1].ChildrenMenus = Submenu2;
    //子菜单越多, 就按照这个套路定义下去
    selectItem = 1; //开机后菜单初始化选择第1项
}
```

LCD显示得是空屏幕

```
display(MenuPoint, selectItem); //
//因为在初始化的时候*MenuPoint = MainMenu菜单指针指向了主菜单,
//所以这次display函数执行是显示主菜单内容,
//又因为selectItem = 1; 所以display光标会选择主菜单第1行
```

第3步

```
void display(struct MenuItem *MenuP, short selectnum) //传入当前选中的菜单项
{
    int x = 30; //屏幕开始位置设置
    int y = 30; //当前选中的菜单项显示, 用填充来显示
    LCD_Fill(0, y+selectnum*24-24, lcddev.width, y+selectItem*24, BLUE); //这是LCD显示函数, 可以不用管报错主要看细节
    for(int j = 0; j < MenuP->MenuCount; j++) //根据传入的菜单结构, 获取菜单结构里面的菜单计数器, 然后用菜单计数器里
    { //面的数, 循环菜单数组里面的字符, 这里循环5次, 在主界面显示5个菜单
        show_str(x, y, lcddev.width, 24, MenuP[j].DisplayString, 24, 1); //这是LCD显示函数可以不用管报错主要看细节
    } //每次循环一个菜单项, 获取一个字符, 输出给屏幕
}
```

LCD主界面显示



按键扫描程序操作主界面光标

```
int key = 0;

while(1)
{
    key = KEY_Scan(0); // 按键扫描
    switch(key)
    {
        case WKUP_PRES // 按下向上按
        {
            if(selectItem == 1)
            {
                selectItem = MenuPoint->MenuCount;
                // 如果当前行数等于1, 当前行数为当前界面总行数,
                // 就是将光标移到界面最底部
            }
            else
            {
                selectItem--; // 如果当前行数不为1, 那么向上键按下, 菜单光标向上移到一行
            }
        };break;
    }
}

return 0;
```



本来选项在第1行



如果再向上按, 就让选项跳到本界面最大行

```
while(1)
{
    key = KEY_Scan(0); // 按键扫描
    switch(key)
    {
        case WKUP_PRES // 按下向上按
        {
            if(selectItem == 1)
            {
                selectItem = MenuPoint->MenuCount;
                // 如果当前行数等于1, 当前行数为当前界面总行数,
                // 就是将光标移到界面最底部
            }
            else
            {
                selectItem--; // 如果当前行数不为1, 那么向上键按下, 菜单光标向上移到一行
            }
        };break;
        case KEY0_DOWN
        {
            if(selectItem == MenuPoint->MenuCount)
            {
                selectItem = 1;
                // 同样, 如果当前行数为最大行, 如果再按一次向下按钮, 就跳转到本界面第1行
            }
            else
            {
                selectItem++; // 如果不是最大行, 就让光标选项向下滑动一行
            }
        }
    }
}
```

注意, 忘记加分号了

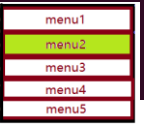


向下滑动一行

如果选择到某一行了, 向进入该行或者该项的子菜单怎么办?

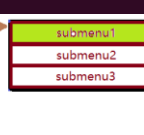
```
case KEY0_DOWN:
{
    if(selectItem == MenuPoint->MenuCount)
    {
        selectItem = 1;
        // 同样, 如果当前行数为最大行, 如果再按一次向下按钮, 就跳转到本界面第1行
    }
    else
    {
        selectItem++; // 如果不是最大行, 就让光标选项向下滑动一行
    }
}
case KEY_ENTER:
{
    if(MainMenu[selectItem-1].ChildrenMenus != NULL)
    {
        MenuPoint = MainMenu[selectItem-1].ChildrenMenus;
        // 如果子菜单不为空, 就证明定义了主菜单选项对应的子菜单的
        selectItem = 1; // 进入子菜单后, 选择第1行
    }
};
```

MenuPoint = MainMenu
本来MenuPoint是指向主菜单的



本来指针指向的主界面

MenuPoint = 子菜单地址



如果按下KEY_ENTER进入子菜单, 那么就需要把子菜单地址给MenuPoint, 后面display执行的时候, 才会把屏幕变成子菜单界面

进入子菜单

```

        case KEY_ENTER:
        {
            if(MainMenu[selectItem-1].ChildrenMenus != NULL)
            {
                MenuPoint = MenuPoint[selectItem - 1].ChildrenMenus;
                //如果子菜单不为空，就证明定义了主菜单选项对应的子菜单的
                selectItem = 1; //进入子菜单后，选择第1行
            }
        };
        case KEY_EXIT:
        {
            if(MainMenu[selectItem-1].ParentMenus != NULL)
            {
                MenuPoint = MenuPoint[selectItem - 1].ParentMenu;
                selectItem = 1;
            }
        };break;
    }

    LCD_Clear(WHITE); //不管进入子菜单还是父菜单，还是选择上选择下，
                      //都需要修改屏幕显示内容，所以每次switch之后要清除一次屏幕

    display(MenuPoint,selectItem);
    //清屏之后，就要根据MenuPoint得到的新菜单地址，也可能是主菜单，子菜单。
    //反要重新设置菜单指针，重新显示。
}

```

这就是整个多级菜单架构，很有用。

代码实例

```

#include <stdio.h>
struct MenuItem
{
    short MenuCount;      //当前菜单项目数
    char *DisplayString;  //当前项目要显示得字符
    void(*subs)();        //某一项菜单被选中，执行的功能函数
    struct MenuItem *ChildrenMenus; //当前项目的下级子菜单
    struct MenuItem *ParentMenus;  //当前项目上级父菜单
};

/*主菜单定义*/
struct MenuItem MainMenu[5] =
{
    { 5,(char *)"menu1",NULL,NULL,NULL},
    { 5,(char *)"menu2",NULL,NULL,NULL},
    { 5,(char *)"menu3",NULL,NULL,NULL},
    { 5,(char *)"menu4",NULL,NULL,NULL},
    { 5,(char *)"menu5",NULL,NULL,NULL}
};

/*子菜单*/
struct MenuItem Submenu1[3] =
{
    { 3,(char *)"submenu1",NULL,NULL,MainMenu},
    { 3,(char *)"submenu2",NULL,NULL,MainMenu},
    { 3,(char *)"submenu3",NULL,NULL,MainMenu}
};

struct MenuItem *MenuPoint = MainMenu; //开机后当前菜单地址，一般都是主菜单
short selectItem;      //当前选择的第几行菜单

void display(struct MenuItem *Menu,short selectnum)
{
    int x = 30;
    int y = 30;
    LCD_Fill(0,y+selectnum*24-24,Lcddev.width,y+selectItem*24,BLUE); //这是 LCD 显示函数，可以不用管报错主要看细节

    for(int j = 0; j < Menu->MenuCount; j++)
    {
        show_str(x,y,Lcddev.width,24,Menu[j].DisplayString,24,1); //这是 LCD 显示函数可以不用管报错主要看细节
    }
}

```

```

int main(void)
{
    MainMenu[0].ChildrenMenus = Submenu1;
    //初始化的时候, 要让主界面指定每一项的子菜单, 我这里指定主界面第 1 项子菜单为 Submenu1 结构
    //比如又定义了一个子菜单结构 Submenu2 给主界面第 2 项
    //那么初始化增加 MainMenu[1].ChildrenMenus = Submenu2;
    //子菜单越多, 就按照这个套路定义下去
    selectItem = 1; //开机后菜单初始化选择第 1 项

    display(MenuPoint,selectItem);
    //因为在初始化的时候*MenuPoint = MainMenu 菜单指针指向了主菜单,
    //所以这次 display 函数执行是显示主菜单内容,
    //又因为 selectItem = 1; 所以 display 光标会选择主菜单第 1 行

    int key = 0;
    while(1)
    {
        key = KEY_Scan(0); //按键扫描
        switch(key)
        {
            case WKUP_PRES: //按下向上按
            {
                if(selectItem == 1)
                {
                    selectItem = MenuPoint->MenuCount;
                    //如果当前行数等于 1, 当前行数为当前界面总行数,
                    //就是将光标移到界面最底部
                }
                else
                    selectItem--; //如果当前行数不为 1, 那么向上键按下, 菜单光标向上移> 到一行

            };break;
            case KEY0_DOWN:
            {
                if(selectItem == MenuPoint->MenuCount)
                {
                    selectItem = 1;
                    //同样, 如果当前行数为最大行, 如果再按一次向下按钮, 就跳转到本界面
                }
                else
                    selectItem++; //如果不是最大行,就让光标选项向下滑动一行
            }
            case KEY_ENTER:
            {
                if(MainMenu[selectItem-1].ChildrenMenus != NULL)
                {
                    MenuPoint = MenuPoint[selectItem - 1].ChildrenMenus;
                    //如果子菜单不为空, 就证明定义了主菜单选项对应的子菜单的
                    selectItem = 1; //进入子菜单后, 选择第 1 行
                }
            };
            case KEY_EXIT:
            {
                if(MainMenu[selectItem-1].ParentMenus != NULL)
                {
                    MenuPoint = MenuPoint[selectItem - 1].ParentMenu;
                    selectItem = 1;
                }
            };break;
        }
        LCD_Clear(WHITE); //不管进入子菜单还是父菜单, 还是选择上选择下,
        //都需要修改屏幕显示内容, 所以每次 switch 之后要清除一次屏幕

        display(MenuPoint,selectItem);
        //清屏之后, 就要根据 MenuPoint 得到的新菜单地址, 也可能是主菜单, 子菜单。
        //反要重新设置菜单指针, 重新显示。
    }
    return 0;
}

```

以上菜单界面只适用于一级界面，多级菜单框架没有错，但是在多级菜单实现的时候，细节还是有些问题

```
struct MenuItem
{
    short MenuCount; //当前菜单项目数
    char *DisplayString; //当前菜单项显示的字符
    void (*subs)(); //某一项菜单被选中，执行功能函数
    struct MenuItem *pChildrenMenus; //当前菜单项的下级子菜单
    struct MenuItem *pParentMenus; //当前菜单项上级父菜单
};
```

```
/*主菜单界面内容*/
struct MenuItem MenuNum[9] =
{
    {9, (char*) "menu1", Menu1Dispfunc, 0, 0},
    {9, (char*) "menu2", Menu2Dispfunc, 0, 0},
    {9, (char*) "menu3", Menu3Dispfunc, 0, 0},
    {9, (char*) "menu4", Menu4Dispfunc, 0, 0},
    {9, (char*) "menu5", Menu5Dispfunc, 0, 0},
    {9, (char*) "menu6", Menu6Dispfunc, 0, 0},
    {9, (char*) "menu7", Menu7Dispfunc, 0, 0},
    {9, (char*) "menu8", Menu8Dispfunc, 0, 0},
    {9, (char*) "menu9", Menu9Dispfunc, 0, 0}
};
```

比如我定义了菜单一级界面

```
/*
 * 菜单5 子菜单, 状态显示设置, 设置站号
 */
struct MenuItem Menu5subNode =
{
    1,
    (char*) "menu5subnode",
    Menu5subNodefunc,
    0,
    &MenuNum[3] //本子菜单上一级父菜单单位
};
```

我二级菜单直接用结构体实现，这样可以吗？

```
/******
 * 菜单显示事件循环
 *****/
void display(struct MenuItem *Menu, short selectnum)
{
    Menu[selectnum].subs();
}
```

在菜单循环中，
我们是根据
selectnum 的编
号，循环菜单数
组里面的内容

那么子菜单也必须以数组形式实现，不能直接用单个结构体实现。

```
struct MenuItem *pMenuPoint = MenuNum; //初始化将主菜单数组地址赋值给结构体指针
short Menuselect = 0; //当前显示菜单号(不区分主菜单还是子菜单)

MenuNum[3].pChildrenMenus = Menu4subNode; //将主菜单4的子菜单赋值给主菜单4
```

下面在按键循环中，注意菜单结构体数组的传参方式。

```
Menu_Key_pro(pMenuPoint, MenuNum, &gKey_Event, &Menuselect); //按键菜单切换处理
display(pMenuPoint, Menuselect); //菜单显示
```



```

void Menu_Key_pro(struct MenuItem *MenuPoint, struct MenuItem MenuStruct[], Key_eStatus
{

    switch(*pKey_eSta)
    {
        case eKEY_MENU_DOWN: //按键SET 设置菜单按下
        {

            if(MenuPoint[*pSelect].pChildrenMenus != NULL)
            {
                memset(OLED_CharGRAM, 0, sizeof(OLED_CharGRAM)); //清空显示缓存
                MenuPoint[0] = MenuPoint[3].pChildrenMenus[0]; //结构体数组除了初始化 //可以直接赋值给结构体指针
                //MenuPoint[0] = Menu4subNode[0];这样是可以的 //在运行过程中, 结构体数组赋值 //给结构体指针必须都用下标
            }
            *pSelect = 0; //子菜单选择第1行
        }
    }
}

```

单片机按键架构

multi_button.c 实现

```
#include "multi_button.h"
#define EVENT_CB(ev)  if(handle->cb[ev])handle->cb[ev]((Button*)handle)

//按钮句柄链表
static struct Button* head_handle = NULL;

/*
 * 初始化按钮结构
 * handle: 传入按钮句柄结构
 * pin_level: 读取 GPIO 管脚电平, 需要读取 IO 的实现硬件回调函数, 然后传入进来
 * active_level: 默认 IO 电平, 一般填 0
 */
void button_init(struct Button* handle, uint8_t(*pin_level)(), uint8_t active_level)
{
    memset(handle, 0, sizeof(struct Button));
    handle->event = (uint8_t)NONE_PRESS;
    handle->hal_button_Level = pin_level;
    handle->button_level = handle->hal_button_Level();
    handle->active_level = active_level;
}

/*
 * 注册按钮按下, 或者松开事件回调函数, 一般按下设置一个回调, 松开实现一个回调
 * handle: 传入按钮句柄结构
 * event: 事件触发类型, 就是该回调函数是支持按钮按下执行吗, 还是按钮松开执行
 * cb: 回调函数
 */
void button_attach(struct Button* handle, PressEvent event, BtnCallback cb)
{
    handle->cb[event] = cb;
}

PressEvent get_button_event(struct Button* handle)
{
    return (PressEvent)(handle->event);
}

void button_handler(struct Button* handle)
{
    uint8_t read_gpio_level = handle->hal_button_Level();

    //ticks counter working..
    if((handle->state) > 0) handle->ticks++;

    /*-----button debounce handle-----*/
    if(read_gpio_level != handle->button_level) { //not equal to prev one
        //continue read 3 times same new level change
        if(++(handle->debounce_cnt) >= DEBOUNCE_TICKS) {
            handle->button_level = read_gpio_level;
            handle->debounce_cnt = 0;
        }
    }
}
```



```

} else { //leved not change ,counter reset.
    handle->debounce_cnt = 0;
}

/*-----State machine-----*/
switch (handle->state) {
case 0:
    if(handle->button_level == handle->active_level) { //start press down
        handle->event = (uint8_t)PRESS_DOWN;
        EVENT_CB(PRESS_DOWN);
        handle->ticks = 0;
        handle->repeat = 1;
        handle->state = 1;
    } else {
        handle->event = (uint8_t)NONE_PRESS;
    }
    break;

case 1:
    if(handle->button_level != handle->active_level) { //released press up
        handle->event = (uint8_t)PRESS_UP;
        EVENT_CB(PRESS_UP);
        handle->ticks = 0;
        handle->state = 2;

    } else if(handle->ticks > LONG_TICKS) {
        handle->event = (uint8_t)LONG_PRESS_START;
        EVENT_CB(LONG_PRESS_START);
        handle->state = 5;
    }
    break;

case 2:
    if(handle->button_level == handle->active_level) { //press down again
        handle->event = (uint8_t)PRESS_DOWN;
        EVENT_CB(PRESS_DOWN);
        handle->repeat++;
        EVENT_CB(PRESS_REPEAT); // repeat hit
        handle->ticks = 0;
        handle->state = 3;
    } else if(handle->ticks > SHORT_TICKS) { //released timeout
        if(handle->repeat == 1) {
            handle->event = (uint8_t)SINGLE_CLICK;
            EVENT_CB(SINGLE_CLICK);
        } else if(handle->repeat == 2) {
            handle->event = (uint8_t)DOUBLE_CLICK;
            EVENT_CB(DOUBLE_CLICK); // repeat hit
        }
        handle->state = 0;
    }
    break;

case 3:
    if(handle->button_level != handle->active_level) { //released press up
        handle->event = (uint8_t)PRESS_UP;
        EVENT_CB(PRESS_UP);

```

```

        if(handle->ticks < SHORT_TICKS) {
            handle->ticks = 0;
            handle->state = 2; //repeat press
        } else {
            handle->state = 0;
        }
    }
    break;

case 5:
    if(handle->button_level == handle->active_level) {
        //continue hold trigger
        handle->event = (uint8_t)LONG_PRESS_HOLD;
        EVENT_CB(LONG_PRESS_HOLD);

    } else { //releasd
        handle->event = (uint8_t)PRESS_UP;
        EVENT_CB(PRESS_UP);
        handle->state = 0; //reset
    }
    break;
}
}

```

```

int button_start(struct Button* handle)
{
    struct Button* target = head_handle;
    while(target) {
        if(target == handle) return -1;    //already exist.
        target = target->next;
    }
    handle->next = head_handle;
    head_handle = handle;
    return 0;
}

```

```

void button_stop(struct Button* handle)
{
    struct Button** curr;
    for(curr = &head_handle; *curr; ) {
        struct Button* entry = *curr;
        if (entry == handle) {
            *curr = entry->next;
            free(entry);
        } else
            curr = &entry->next;
    }
}

```

```

void button_ticks()
{
    struct Button* target;
    for(target=head_handle; target; target=target->next) {
        button_handler(target);
    }
}

```

这就是整个实现过程。主函数调用就行

multi_button.h 头文件实现

```
#ifndef _MULTI_BUTTON_H_
#define _MULTI_BUTTON_H_

#include "stdint.h"
#include "string.h"

//According to your need to modify the constants.
#define TICKS_INTERVAL    5    //ms
#define DEBOUNCE_TICKS    3    //MAX 8
#define SHORT_TICKS       (300 /TICKS_INTERVAL)
#define LONG_TICKS        (1000 /TICKS_INTERVAL)

typedef void (*BtnCallback)(void*);

typedef enum {
    PRESS_DOWN = 0,
    PRESS_UP,
    PRESS_REPEAT,
    SINGLE_CLICK,
    DOUBLE_CLICK,
    LONG_PRESS_START,
    LONG_PRESS_HOLD,
    number_of_event,
    NONE_PRESS
}PressEvent;

typedef struct Button {
    uint16_t ticks;
    uint8_t repeat : 4;
    uint8_t event : 4;
    uint8_t state : 3;
    uint8_t debounce_cnt : 3;
    uint8_t active_level : 1;
    uint8_t button_level : 1;
    uint8_t (*hal_button_Level)(void);
    BtnCallback cb[number_of_event];
    struct Button* next;
}Button;

#ifdef __cplusplus
extern "C" {
#endif

void button_init(struct Button* handle, uint8_t(*pin_level)(), uint8_t active_level);
void button_attach(struct Button* handle, PressEvent event, BtnCallback cb);
PressEvent get_button_event(struct Button* handle);
int button_start(struct Button* handle);
void button_stop(struct Button* handle);
```

```
void button_ticks(void);
```

```
#ifdef __cplusplus  
{  
#endif
```

```
#endif
```

主函数调用方式

```
#include "multi_button.h"
```

```
/*底层按键采集函数实现*/
```

```
uint8_t read_P16button_GPIO()  
{  
    return 0/1; //按键按下返回 0, 按键松开返回 1  
}
```

```
//button1 按钮按下事件回调函数
```

```
void button1_press_down_Handler(void* btn)  
{  
    printf("---> key1 press down! <---\n");  
}
```

```
//button1 松开事件回调函数
```

```
void button1_press_up_Handler(void* btn)  
{  
    printf("***> key1 press up! <***\n");  
}
```

```
int main( void )
```

```
{  
    struct Button button1; //定义一个按钮  
    /*按钮 GPIO 初始化函数*/  
    button_init(&button1, read_P16button_GPIO, 0); //将硬件按钮检测函数与自定义的 button1 句柄连接起来。
```

```
    //注册按钮事件回调函数
```

```
    button_attach(&button1, PRESS_DOWN, button1_press_down_Handler); //按钮按下事件  
    button_attach(&button1, PRESS_UP, button1_press_up_Handler); //按钮松开事件  
    button_start(&button1); //启动按钮
```

```
    while(1)
```

```
{  
    //每隔 5ms 调用一次后台处理函数  
    button_ticks();  
    delay_ms(5); //延时 5ms, 可以考虑把 button_ticks()放入定时器处理。  
}
```

```
}
```

定时器多任务框架

/*任务处理结构体*/

typedef struct _Task_Struc

{

char isRun; //表示任务是否在运行

unsigned int TimerSlice; //分配给任务的时间片

unsigned SliceNumber; //时间片的个数，在 TimerSlice 为 0 时，将其赋值给 TimerSlice 重新计数

void (*TaskPointer)(void* parameter); //任务的函数指针

}Task_Struc,*Task_Struct_Pointer;

/*创建需要的任务*/

void task1(void* parameter)

{

printf("LED is blinking.\n");

}

void task2(void* parameter)

{

printf("LCD is Running.\n");

}

/*创建好的任务需要初始化，我这里是两个任务*/

Task_Struct tasks[] =

{

{0,200,200,task1}, //用各个任务的函数名初始化

{0,60,60,task2},

};

unsigned long task_count = sizeof(tasks) / sizeof(Task_Struct); //保存所有任务的执行数量

//定时器中断处理函数，放入定时器中断内

void IRQ_Task_Process(void)

{

unsigned char i = 0;

for (i=0; i < task_count; ++i) //遍历任务数组

{

if (tasks[i].TimerSlice) //判断时间片是否到了

{

--tasks[i].TimerSlice;

if (0 == tasks[i].TimerSlice) //时间片到了

{

tasks[i].isRun = 0x01; //置位 表示任务可以执行

tasks[i].TimerSlice = tasks[i].SliceNumber; //重新加载时间片值，为下次做准备

}

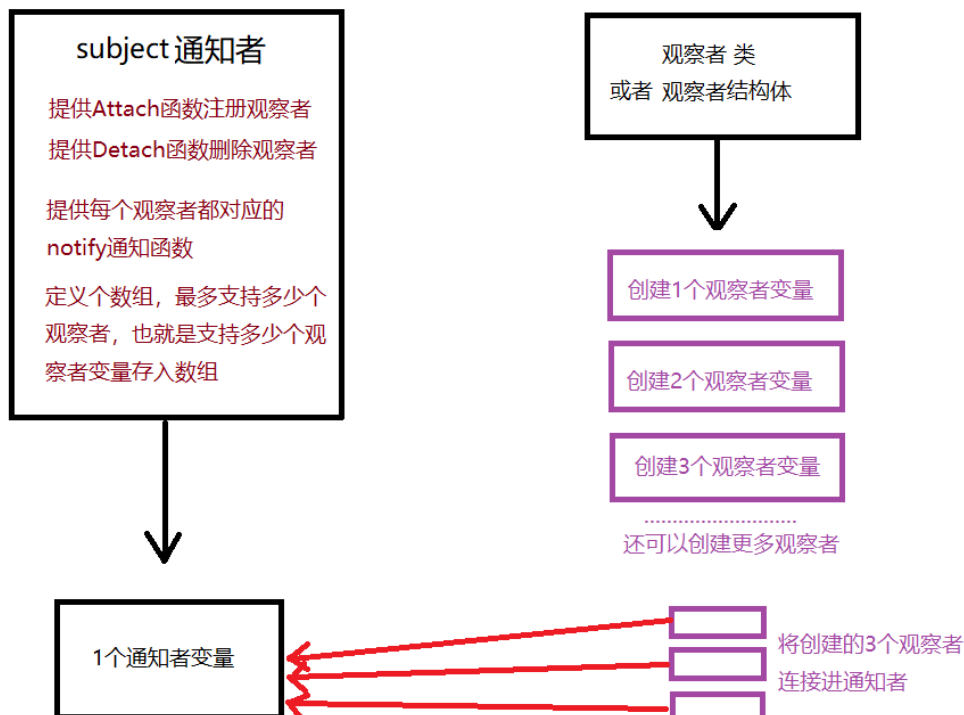
}

}

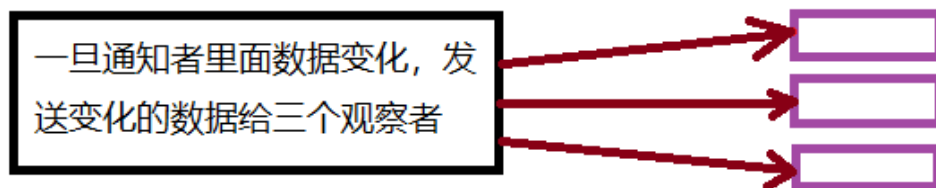
}

```
// 定时器任务处理函数
void Task_Process()
{
    unsigned char i = 0;
    for (i=0; i < task_count; ++i) //遍历任务数组
    {
        if (tasks[i].isRun) //若任务可执行，则执行任务
        {
            tasks[i].TaskPointer(0);
            tasks[i].isRun = 0; //将标志位清零
        }
    }
}
```

C 语言实现观察者模式(Observer)



通知者循环检测和循环发送



下面以明细和粉丝为例, 来说明观察者模式的应用

```
struct Fans {  
    struct Fengjie* fengjie;  
    void (*update) (int value);  
    void (*joinfans)(struct Fans fans);  
    void (*quitfans)();  
};
```

观察者 类
或者 观察者结构体

//粉丝获得凤姐的新状态, 被动获得
//粉丝有权选择成为凤姐粉丝
//粉丝有权选择退出凤姐粉丝圈

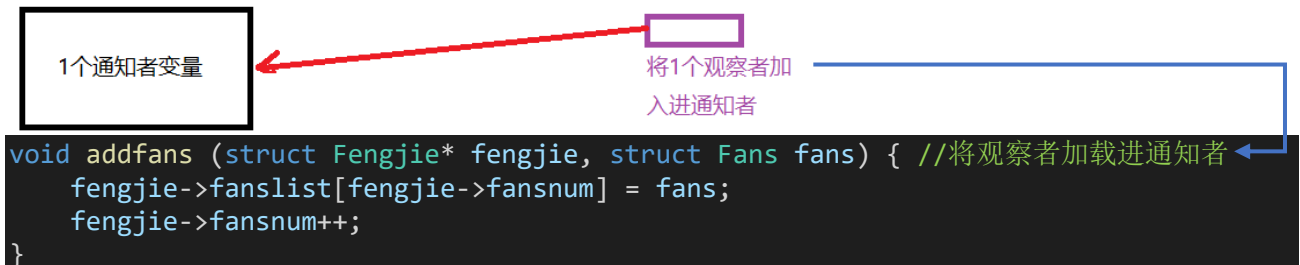
```
struct Fengjie {  
    int fansnum;  
    int weight;  
    struct Fans fanslist[100]; //负责通知的粉丝的数量,最多 100 个粉丝(subject 内容)  
    void (*notify) (struct Fengjie* fengjie); //由凤姐发出更新通知(subject 内容)  
    void (*addfans)(struct Fengjie* fengjie, struct Fans fans); //凤姐这边也可以  
    // void (*delfans)(); //to be done  
};
```

subject 通知者

- 提供Attach函数注册观察者
- 提供Detach函数删除观察者
- 提供每个观察者都对应的


```
void joinfans(struct Fans fans) {
    fans.fengjie->addfans(fans.fengjie, fans);    //粉丝这边直接调用凤姐的函数来把该粉丝加入粉丝圈
}
```

观察者就是 struct Fans fans 接收的 fans 变量，将观察者 addfans 加载进通知者



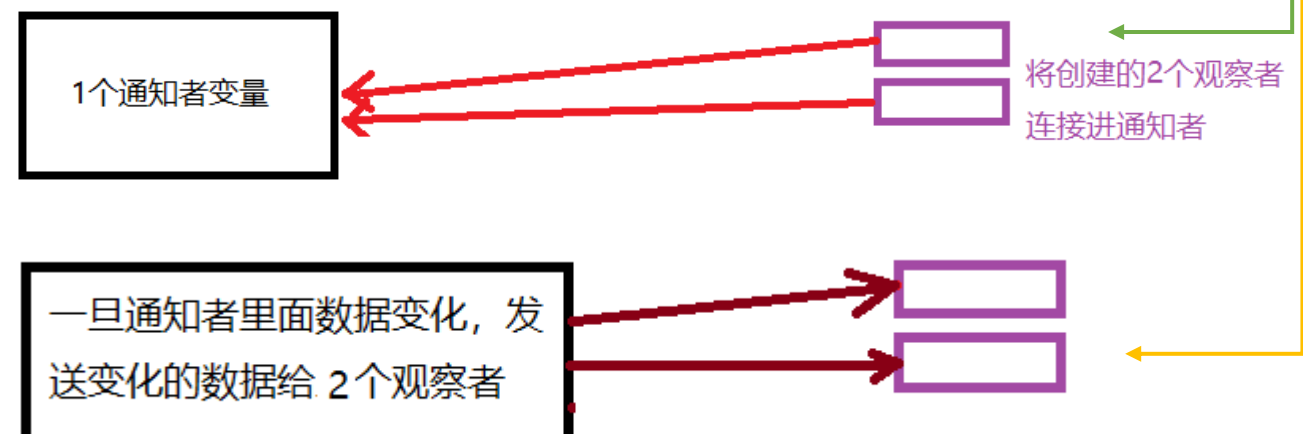
*fengjie 是通知者地址，fans 是观察者

```
void notify (struct Fengjie* fengjie){
    int i;
    for (i = 0; i < fengjie->fansnum; i++)
        fengjie->fanslist[i].update(fengjie->weight); //循环执行每个通知者嵌入的观察者
}
```

```
int main(void)
{
    struct Fengjie fengjie;
    fengjie.notify = notify,    //notify 函数实现传入通知者
    fengjie.addfans = addfans, //加入新的观察者函数传入通知者
    fengjie.fansnum = 0, //for 从 0 开始循环累加，通知到每个观察者
    fengjie.weight = 60; //向所有观察者发送的数据

    struct Fans fan1, fan2; //创建两个观察者
    fan1.fengjie = &fengjie, fan1.joinfans = joinfans, fan1.update = fan1update;
    fan2.fengjie = &fengjie, fan2.joinfans = joinfans, fan2.update = fan2update;
    fan1.joinfans (fan1); //将观察者 1 放入通知者
    fan2.joinfans (fan2); //将观察者 2 放入通知者

    fengjie.notify (&fengjie); //通知者循环执行每一个 Fans 创建的观察者
}
```



C 语言实现简单工厂模式

一句话理解

一个工厂，根据输入类型不一样，产生不同类型的结果，factory 中使用 switch 或者 if else 来做区分。

缺点：每次增加一个新产品，需要对 factor 进行修改，之前对 factory 的测试要重新进行

优点：针对小型程序，设计比较简单、代码耦合度低

```
typedef struct _Shoe
{
    int type;
    void (*print_shoe)(struct _Shoe*);
}Shoe;

void print_leather_shoe(struct _Shoe* pShoe)
{
    assert(NULL != pShoe);
    printf("This is a leather show!\n");
}

void print_rubber_shoe(struct _Shoe* pShoe)
{
    assert(NULL != pShoe);
    printf("This is a rubber shoe!\n");
}

#define LEATHER_TYPE 0x01
#define RUBBER_TYPE 0x02

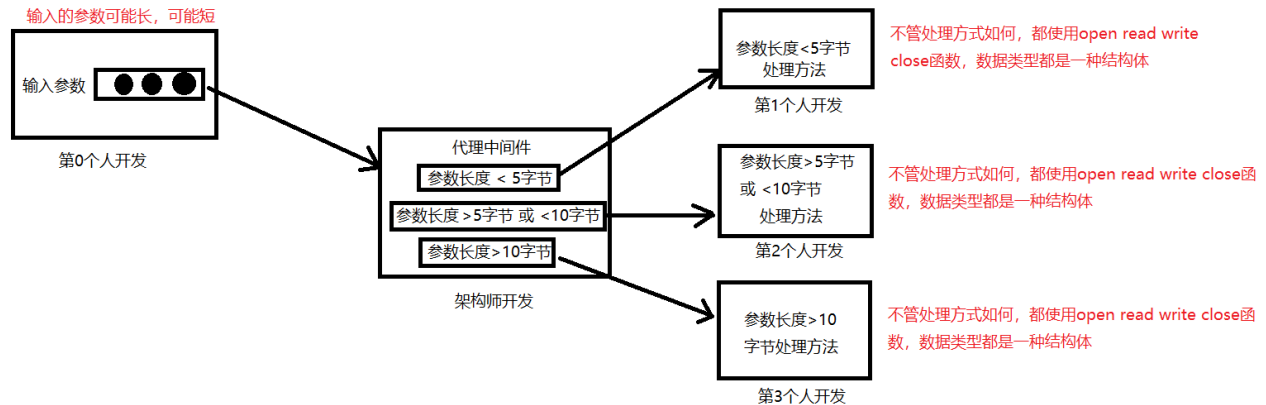
Shoe* manufacture_new_shoe(int type) //根据输入的类型返回不同的对象和函数
{
    assert(LEATHER_TYPE == type || RUBBER_TYPE == type);

    Shoe* pShoe = (Shoe*)malloc(sizeof(Shoe));
    assert(NULL != pShoe);

    memset(pShoe, 0, sizeof(Shoe));
    if(LEATHER_TYPE == type)
    {
        pShoe->type == LEATHER_TYPE;
        pShoe->print_shoe = print_leather_shoe;
    }
    else
    {
        pShoe->type == RUBBER_TYPE;
        pShoe->print_shoe = print_rubber_shoe;
    }

    return pShoe;
}
```

C 语言实现代理模式(Proxy)



这就是代理模式，5 个人开发一个功能，将功能分解之后的实现方法。

```
//定义接口
typedef struct{
    //签名
    char* (*sign)(void*);
    //拍电影
    void (*MakeFilm)(void*);
    //拍广告
    void (*MakeCommercials)(void*);
}BehaviourInterface_t;
```

5 个人都使用同一个接口实现同样的函数，只是函数里面的内容根据输入参数长度不同，而不同

```
//周杰伦的实际处理
static char* zhoujielunsign(void* v)
{
    printf("sign ok...\n");
    return (char*)"Hello";
}
//周杰伦的实际处理
static void zhoujielunMakeFilm(void* v)
{
    printf("MakeFilm ok...\n");
}
//周杰伦的实际处理
static void zhoujielunMakeCommercials(void* v)
{
    printf("MakeCommercials ok...\n");
}
```

第 1 个人处理方法

```
//实例创建
typedef struct{
    //代理对象相当于经纪人
    BehaviourInterface_t Proxy; //代理中间件
    //实际的对象周杰伦
    BehaviourInterface_t zhoujielun; //代理中间件和第 1 个人定义的数据
    //控制对象的变量
    int age;
    int sex;
    char CompanyName[32];
}BehaviourProxyNew_t;
```

代理中间件，要用同样的数据结构，定义代理人和第 1 个人

```

//经纪的实际处理
static char* Proxysign(void* v)
{
    BehaviourProxyNew_t* p = (BehaviourProxyNew_t*)v;
    //如果小于 18 岁或者是男的话就不给你签名，经纪人在这控制了一下
    if(p->age < 18 || p->sex ==1){
        printf("You are too young or you are man,no sign thank you...\n");
        return (char*)"You are too young or you are man,no sign thank you";
    }
    else{
        return p->zhoujielun.sign(v);
    }
    return NULL;
}

//经纪的实际处理
static void ProxyMakeFilm(void* v)
{
    BehaviourProxyNew_t* p = (BehaviourProxyNew_t*)v;
    //如果是小公司的话就不拍，我的艺人就是这么豪横，经纪人在这控制了一下
    if(strcmp(p->CompanyName,"small")==0){
        printf("sorry you company is so small...\n");
    }
    else if(strcmp(p->CompanyName,"big")==0){
        p->zhoujielun.MakeFilm(v);
    }
}

//经纪的实际处理
static void ProxyCommercials(void* v)
{
    BehaviourProxyNew_t* p = (BehaviourProxyNew_t*)v;
    //如果是小公司的话就不怕，我的艺人就是这么豪横，经纪人在这控制了一下
    if(strcmp(p->CompanyName,"small")==0){
        printf("sorry you company is so small...\n");
    }
    else if(strcmp(p->CompanyName,"big")==0){
        p->zhoujielun.MakeCommercials(v);
    }
}

//代理创建
static BehaviourProxyNew_t* NewProxy(int age,int sex,const char* CompanyName)
{
    BehaviourProxyNew_t *p = (BehaviourProxyNew_t*)malloc(sizeof(BehaviourProxyNew_t));

    //经纪人实现接口 相当于代理对象
    p->Proxy.sign = Proxysign;
    p->Proxy.MakeFilm = ProxyMakeFilm;
    p->Proxy.MakeCommercials = ProxyCommercials;

    //周杰伦实现接口 相对于实际对象
    p->zhoujielun.sign = zhoujielunsign;
    p->zhoujielun.MakeFilm = zhoujielunMakeFilm;
    p->zhoujielun.MakeCommercials = zhoujielunMakeCommercials;
}

```

```

//经纪人独有的控制变量
p->age      = age;
p->sex      = sex;
memcpy(p->CompanyName,CompanyName,strlen(CompanyName));

return p;
}

int main(void)
{
    //第一个代理处理名为 hansen 的人 男性 27 岁 大公司的人
    BehaviourInterface_t *HansenProxy = (BehaviourInterface_t*)NewProxy(27,1,"big")
; //1: male
    //第二个人名为 Maria 的人，女性 22 岁，小公司的人
    BehaviourInterface_t *MariaProxy  = (BehaviourInterface_t*)NewProxy(22,0,"small
"); //0: female
    printf("hansenProxy-----\n");
    HansenProxy->sign(HansenProxy);
    HansenProxy->MakeFilm(HansenProxy);
    HansenProxy->MakeCommercials(HansenProxy);
    printf("\n\nMariaProxy-----\n");
    MariaProxy->sign(MariaProxy);
    MariaProxy->MakeFilm(MariaProxy);
    MariaProxy->MakeCommercials(MariaProxy);
}

```

C 语言实现命令模式

好处：让代码清晰明了，容易添加和删除，易维护。

哪些地方会用到命令模式？（列出几个常见的例子）

- 1、按键处理，每个按键按下得到一个索引（指的就是命令），一个按键对应一个处理函数。[按键处理命令模式](#)
- 2、协议解析（串口，网口，CAN，等等）；以串口为例简单说明一下，比如有如下协议：[http类型解析](#) (html, jpg, jpeg...)

帧头	命令	数据长度	数据内容	校验	帧尾
1字节	1字节	2字节	n字节	2字节	1字节

命令1: 0x01 温度

命令2: 0x02 湿度

命令3: 0x03 光照强度

传统代码实现方式，是用 switch case

```
static uint8_t parse(char *buffer, uint16_t length) //接收串口数据到 buffer
{
    uint8_t head = buffer[0];
    uint8_t cmd = buffer[1];
    uint16_t len = (buffer[2] << 8) | buffer[3];
    uint16_t crc = CRCCheck(buffer, length - 3);
    uint8_t tail = buffer[length - 1];

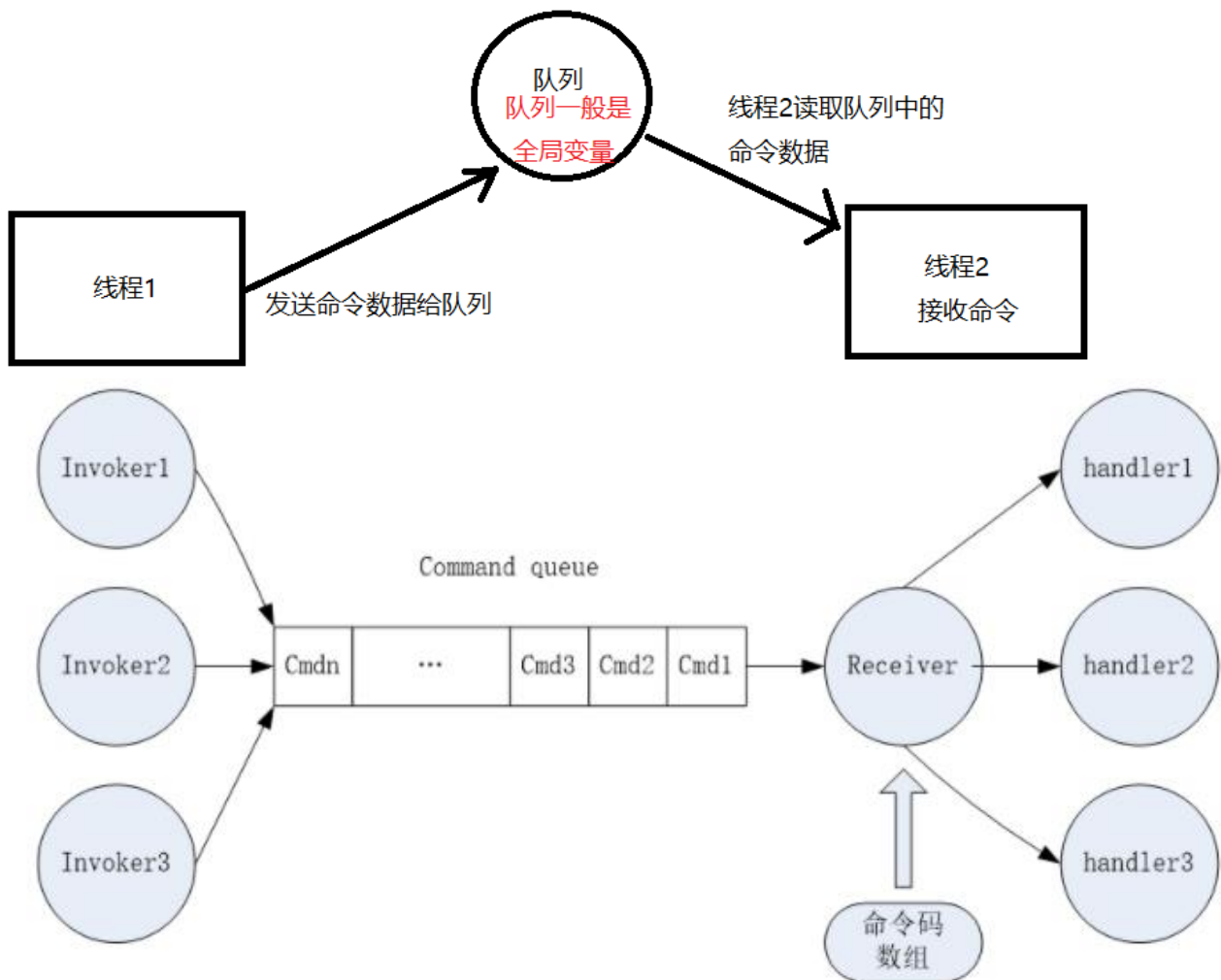
    if((head != 0x01) && (tail != 0x01) && (crc != ((buffer[length - 3] << 8) | buffer[length - 2])))
    {
        return 0;
    }
    switch(cmd)
    {
        case 0x01:
            int temperatue = *(int *)&buffer[4];
            printf("temperatue = %d\n", temperatue);
            break;
        case 0x02:
            int humidity = *(int *)&buffer[4];
            printf("humidity = %d\n", humidity);
            break;
        case 0x03:
            int illumination= *(int *)&buffer[4];
            printf("illumination = %d\n", illumination);
            break;
        default:
            printf("parse error\n");
            break;
    }
    return 1;
}
```

通过这段伪代码可以看出代码结构的一些问题，如果要添加更多的命令，势必需要向 switch case 语句中加入更多的 case 语句。使得解析函数越来越臃肿。

使用命令模式来解决 switch case 问题，命令模式只需要一个命令列表就搞定，增加新功能的命令和函数，只需要填入命令表就行

```
// 当心字节对齐的问题
typedef struct
{
    uint8_t head;
    uint8_t cmd;
    uint16_t length;
    uint8_t data[1];
} package_t;
static int parse_temperature(char *buffer)
{
    int value = *(int *)buffer;
    printf("temperature = %d\n", value);
}
static int parse_humidity(char *buffer)
{
    int value = *(int *)buffer;
    printf("humidity = %d\n", value);
}
static int parse_illumination(char *buffer)
{
    int value = *(int *)buffer;
    printf("illumination = %d\n", value);
}
typedef struct
{
    uint8_t cmd;
    void (* handle)(char *buffer);
} package_entry_t;
static const package_entry_t package_items[] =
{
    {0x01, parse_temperature},
    {0x02, parse_humidity},
    {0x03, parse_illumination},
    {0xFF, NULL},
};
static uint8_t parse(char *buffer, uint16_t length)
{
    package_t *frame = (package_t *)buffer;
    uint16_t crc = CRCCheck(buffer, length - 3);
    uint8_t tail = buffer[length - 1];
    const package_entry_t *entry;
    if((frame->head != 0xFF) && (tail != 0xFF) && (crc != (buffer[length - 3]) << 8 | b
uffer[length - 2]))
    {
        return 0;
    }
    for(entry = package_items; entry->handle != NULL; ++entry)
    {
        if(frame->cmd == entry->cmd)
        {
            entry->handle(frame->data);
            break;
        }
    }
    return 1;
}
```


命令模式的 C 语言实现也是非常显性的。命令发送方不通过直接调用的方式，而是通过发一个命令消息给接收方，让接收方执行操作。C 语言里采用命令模式的最常见的原因是核间通信，进程间交互。如果是核间通信，通常是把命令按协定的格式封装在消息数据包里。如果是进程间通信，通常封装成一个结构体，把参数带过去。命令的通道通常是队列。



```

#define CMD_1 0
#define CMD_2 1
#define CMD_MAX 2

#define CMD_LEN 256
struct cmd_msg
{
    int cmd_code;
    char buf[CMD_LEN]; //如果是不同环境的，只能用 buffer 数组，否则可以用指针
};

int cmd1_handler(char *buf)
{
    printf("cmd1  \n");
    return 0;
}

int cmd2_handler(char *buf)
{
    printf("cmd2  \n");
    return 0;
}

```

```

typedef int (*cmd_func)(char *buf);
cmd_func cmd_table[] =
{
    cmd1_handler,
    cmd2_handler,
};

/*有部分伪代码*/
int invoker1()
{
    struct cmd_msg cmd1_case;
    memset(&cmd1_case, 0, sizeof(cmd1_case));
    cmd1_case.cmd_code = CMD_1;
    //send cmd1_case to queue
    //发送命令给队列，队列应该是全局变量
    return 0;
}

int invoker2()
{
    struct cmd_msg cmd1_case;
    memset(&cmd1_case, 0, sizeof(cmd1_case));
    cmd1_case.cmd_code = CMD_2;
    //send cmd2_case to queue
    //发送命令给队列，队列应该是全局变量
    return 0;
}

int cmd_receiver()
{
    struct cmd_msg *cmd_case;

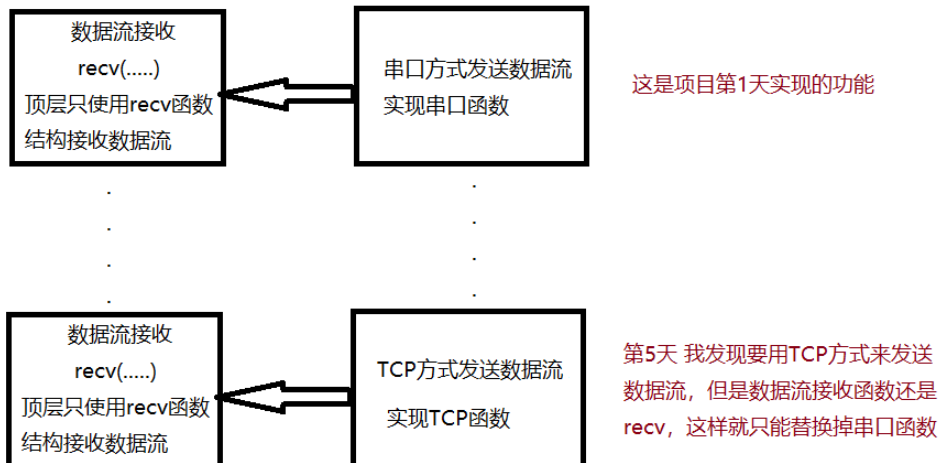
    while(1)
    {
        //get cmd_case from queue while queue is not empty
        //消息接受函数接受队列的命令，进行选择执行，所以队列应该是个全局变量
        (*cmd_table[cmd_case->cmd_code])(cmd_case->buf);
    }
    return 0;
}

int main(void)
{
    invoker1();//发送命令消息到队列
    invoker2();//发送命令消息到队列

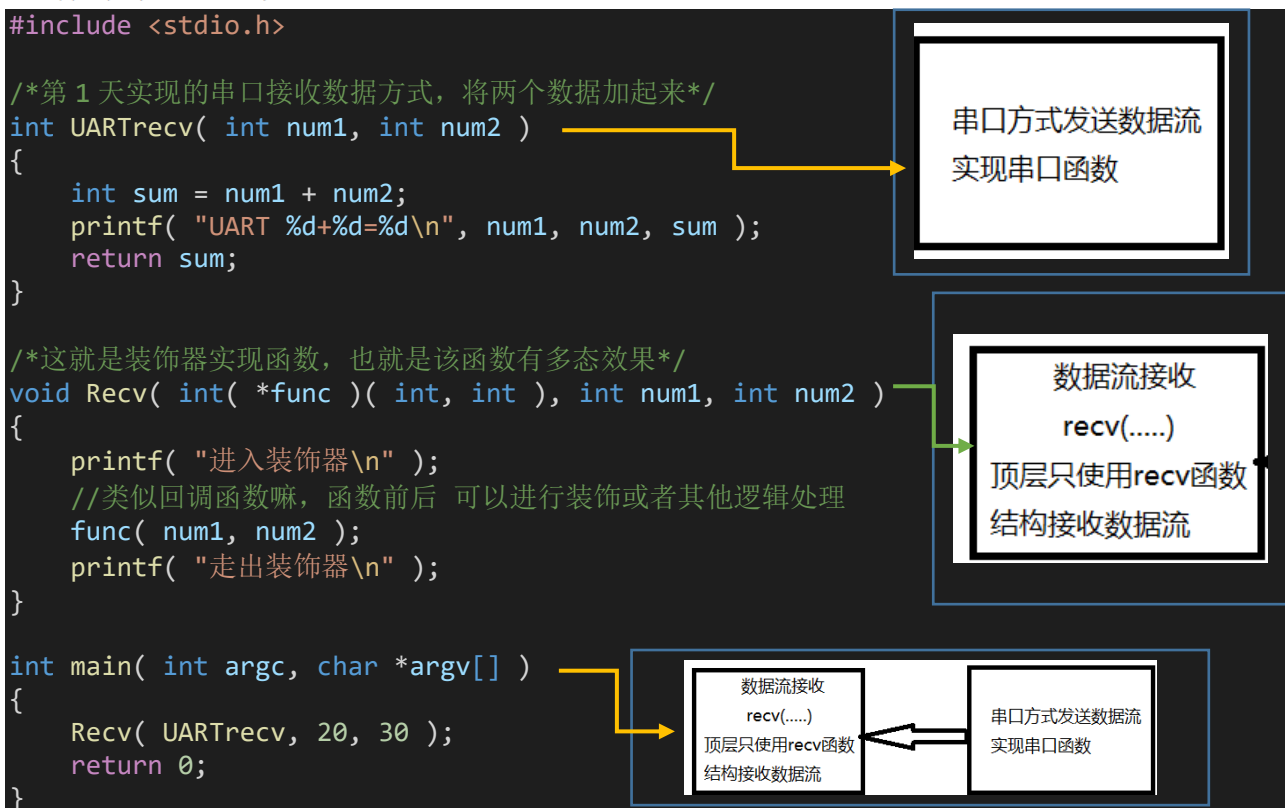
    cmd_receiver();//接受命令消息
    return 0;
}

```

C 语言实现装饰器模式，达到多态效果



所以我预先将recv函数做成多态形式，就是传入任何类型的发送函数都能执行，不管是串口方式还是TCP方式



下面我想加入 TCP 接收数据流方式

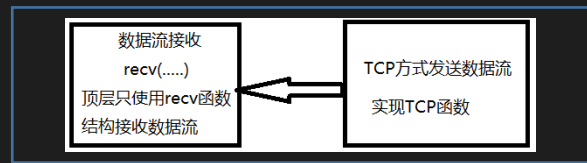
```
/*第5天将串口接收数据方式改成TCP接收数据方式，将两个数减掉*/
int TCPrecv( int num1, int num2 )
{
    int sub = num1 - num2;
    printf( "TCP %d-%d=%d\n", num1, num2, sub );
    return sub;
}
```

```

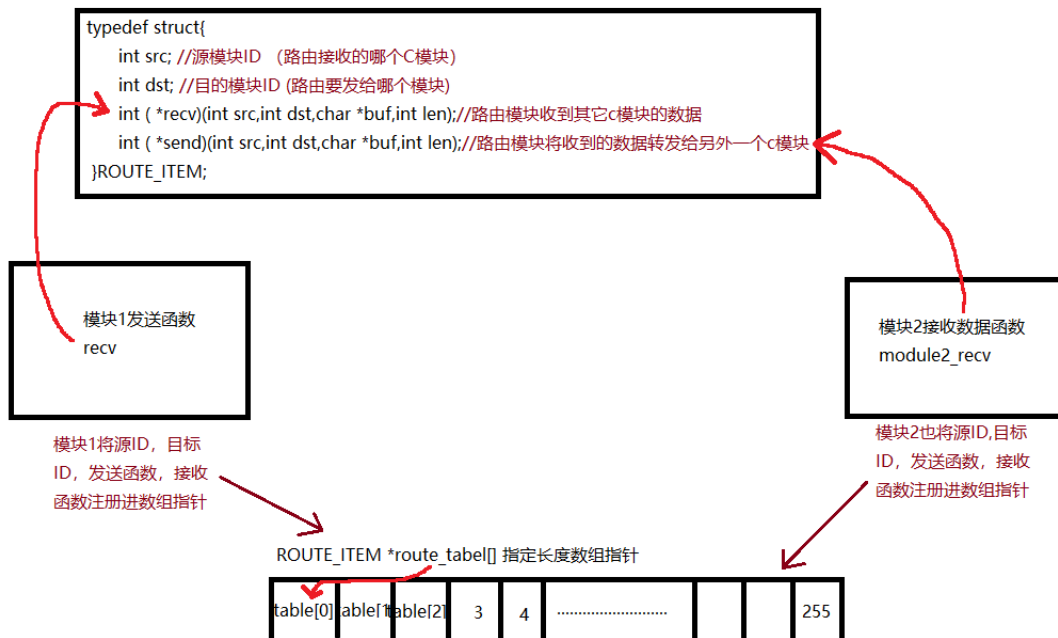
/*这就是装饰器实现函数，也就是该函数有多态效果*/
void Recv( int( *func )( int, int ), int num1, int num2 )
{
    printf( "进入装饰器\n" );
    //类似回调函数嘛，函数前后 可以进行装饰或者其他逻辑处理
    func( num1, num2 );
    printf( "走出装饰器\n" );
}

int main( int argc, char *argv[] )
{
    Recv( TCPrecv, 20, 30 ); //同样的 Recv 接收函数不变，形参也不变
    return 0;
}

```



C 语言路由架构



```

#ifndef __ROUTE_H
#define __ROUTE_H

#define MODULE1_ID 1
#define MODULE2_ID 2
#define MODULE3_ID 3

typedef struct{
    int src; //源模块ID (路由接收的哪个C模块)
    int dst; //目的模块ID (路由要发给哪个模块)
    int ( *recv)(int src,int dst,char *buf,int len); //路由模块收到其它c模块的数据
    int ( *send)(int src,int dst,char *buf,int len); //路由模块将收到的数据转发给另外一个c模块
}ROUTE_ITEM;

extern ROUTE_ITEM *route_table[255]; //路由表供外部C文件调用

```

```
#include <stdio.h>
#include <string.h>
#include "route.h"

#include "module1.h"

static ROUTE_ITEM m1_route;
```

将定义的本模块结构放入数组，里面有本模块的 ID，目标模块 ID....

```
/*
 * 接收其它模块发来的数据
 */
static int module1_rcv(int src,int dst,char *buf,int len)
{
    if(dst != MODULE1_ID)
        printf("module1 rcv failed\n");
    else
        printf("module1 rcv = %s\n",buf);
}

int module1_init(void)
{
    m1_route.src = MODULE1_ID; //自己本文件module1函数的ID
    m1_route.dst = MODULE2_ID; //默认要发送给某个模块的ID，比如我要发数据给模块2(另外一个c文件， module2函数的ID)
    m1_route.rcv = route_rcv; //注意：这儿不是接收，是将数据送到路由进行转发，用rcv很奇怪，这个rcv是要理解成 route.c接收，所以这里的 route_rcv函数是调用的 route.c 里面的 rcv
    m1_route.send = module1_rcv; //默认要发送给某个模块的ID，比如我要发数据给模块2(另外一个c文件， module2函数的ID)

    route_register(&m1_route); //将本文件建立的m1_route路由模块注册进路由表

    return 0;
}

int module1_run(void)
{
    char *str = "1111to2222";

    m1_route.rcv(MODULE1_ID,MODULE2_ID,str,strlen(str)); //将数据从本模块ID1 发送给ID2
    return 0;
}
```

```
#define MODULE1_ID 1
#define MODULE2_ID 2
#define MODULE3_ID 3
```

一共有多少个模块，必须预先在 route.h 文件中写号

本模块要发送数据就调用路由的 route_rcv 回调函数

调用方式就是这样，写入本模块 ID，目标模块 ID，要发送的数据，数据长度

module1.c

```
int route_rcv(int src,int dst,char *buf,int len)
{
    ROUTE_ITEM *item;
    if(item = route_find(dst)) //寻找目的模块
    {
        if(item->send)
            return item->send(src,dst,buf,len); //调用目的模块接收函数
        else
            printf("route send failed \n");
    }

    return -1;
}
```



间接调用模块 2.c 里面的 rcv 函数
其实本质就是模块 1 调用模块 2 的 rcv 函数执行

route.c

```
/*
 * 接收其它模块发来的数据
 */
static int module2_rcv(int src,int dst,char *buf,int len)
{
    if(dst != MODULE2_ID)
        printf("module2 rcv failed\n");
    else
        printf("module2 rcv = %s\n",buf);
}
```

module2.c

整个代码结构如下:

route.h

```
#ifndef __ROUTE_H
#define __ROUTE_H

#define MODULE1_ID    1
#define MODULE2_ID    2
#define MODULE3_ID    3

typedef struct{
    int src; //源模块ID (路由接收的哪个C模块)
    int dst; //目的模块ID (路由要发给哪个模块)
    int ( *recv)(int src,int dst,char *buf,int len); //路由模块收到其它c模块的数据
    int ( *send)(int src,int dst,char *buf,int len); //路由模块将收到的数据转发给另外一个c模块
}ROUTE_ITEM;

extern ROUTE_ITEM *route_table[255]; //路由表供外部C文件调用

int route_init(void); //初始化路由表
int route_exit(void); //销毁路由表
/*
 * 路由接收数据，转发数据到目的模块
 * 该函数只有接收接口，没有发送接口
 * 本接收接口route_recv(...)找到目的c模块之后，调用c模块里面的send
 */
int route_recv(int src,int dst,char *buf,int len);
//路由接收数据，转发数据到目的模块

int route_register(ROUTE_ITEM *route_item);
```

route.c

```
#include <stdio.h>
#include "route.h"

/*每一个C文件模块都要一个ROUTE_ITEM的结构体，把它们统一放在数组里面*/
ROUTE_ITEM * route_table[255] = {0}; //定义路由表

/*
 * 寻找目的模块
 * dst :填入需要寻找的目的模块ID
 */
static ROUTE_ITEM *route_find(int dst)
{
    int i;

    for(i = 0; i < 255; i++) //路由表最大定义的255个
    {
        if(route_table[i])
        {
            if(route_table[i]->src == dst) //路由表循环寻址，一旦发现源地址等于目的地址，返回当前循环的表
            {
                return route_table[i];
            }
        }
    }

    printf("not found route!\n");
    return NULL; //如果整个地址循环完了，没有找到目的地址，然后NULL
}

int route_recv(int src,int dst,char *buf,int len)
{
    ROUTE_ITEM *item;
    if(item = route_find(dst)) //寻找目的模块
    {
        if(item->send)
        {
            return item->send(src,dst,buf,len); //调用目的模块接收函数
        }
        else
        {
            printf("route send failed \n");
        }
    }

    return -1;
}
```

```

/*
 *其它模块C文件就是用该函数将本文件的模块注册进路由表
 */
int route_register(ROUTE_ITEM *route_item)
{
    int i;
    for(i = 0; i < 255; i++)
    {
        if(route_table[i] == NULL)
        {
            route_table[i] = route_item;
            break;
        }
    }

    if(i == 255)
    {
        printf("Route table full!!");
        return -1;
    }

    return 0;
}

```

```

int route_init(void)
{
    return 0;
}

int route_exit(void)
{
    return 0;
}

```

init 和 exit 是用来初始化和销毁模块的，一般没用

下面介绍模块如何使用路由

模块 1

```

/*
 * 接收其它模块发来的数据
 */
static int module1_rcv(int src,int dst,char *buf,int len)
{
    if(dst != MODULE1_ID)
    {
        printf("module1 rcv failed\n");
    }
    else
    {
        printf("module1 rcv = %s\n",buf);
    }
}

int module1_init(void)
{
    m1_route.src = MODULE1_ID; //自己本文件module1函数的ID
    m1_route.dst = MODULE2_ID; //默认要发送给某个模块的ID, 比如我要发数据给模块2(另外一个c文件, module2函数的ID)
    m1_route.rcv = route_rcv; //注意: 这儿不是接收, 是将数据送到路由进行转发, 用rcv很奇怪, 这个rcv是要理解成 route.c接收, 所以这里的 route_rcv函数是调用的 route.c里面的 rcv
    m1_route.send = module1_rcv; //默认要发送给某个模块的ID, 比如我要发数据给模块2(另外一个c文件, module2函数的ID)

    route_register(&m1_route); //将本文件建立的m1_route路由模块注册进路由表

    return 0;
}

int module1_run(void)
{
    char *str = "1111to2222";

    m1_route.rcv(MODULE1_ID,MODULE2_ID,str,strlen(str)); //将数据从本模块ID1 发送给ID2
    return 0;
}

```

主要是在 main 函数初始化的时候注册模块，一旦模块被注册，其它 c 文件就可以间接调用 xxx_route.send

模块 2

```
#include "route.h"
#include "module2.h"

static ROUTE_ITEM m2_route;

/*
 * 接收其它模块发来的数据
 */
static int module2_rcv(int src,int dst,char *buf,int len)
{
    if(dst != MODULE2_ID)
        printf("module2 rcv failed\n");
    else
        printf("module2 rcv = %s\n",buf);
}

int module2_init(void)
{
    m2_route.src = MODULE2_ID; //自己本文件module1函数的ID
    m2_route.dst = MODULE1_ID; //默认要发送给某个模块的ID, 比如我要发数据给模块2(另外一个c文件, module2函数的ID)
    m2_route.rcv = route_rcv; //注意: 这儿不是接收, 是将数据送到路由进行转发, 用rcv很奇怪, 这个rcv是要理解成route.c接收, 所以这里的route_rcv函数是调用的route.c里面的rcv
    m2_route.send = module2_rcv; //默认要发送给某个模块的ID, 比如我要发数据给模块2(另外一个c文件, module2函数的ID)

    route_register(&m2_route); //将本文件建立的m1_route路由模块注册进路由表
}

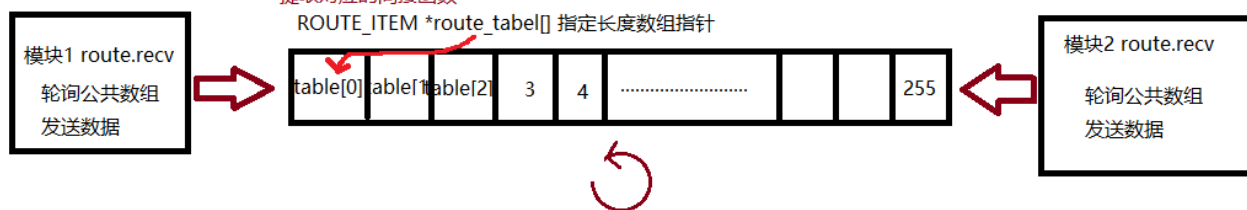
int module2_run(void)
{
}
```

模块 2 的代码结构基本和模块 1 是一样的
只是这里接收数据要判断是不是模块 2 得到了数据

数据流修改为模块 2 为源 ID
模块 1 位接收目的 ID

模块 2 现在做的接收模块 1 数据, 所以
空函数空转

这个数组是公共的, 所以模块1, 模块2发送数据都可以在数组里面找到对应的ID,
提取对应的间接函数



```
#include <unistd.h>
#include "module1.h"
#include "module2.h"
#include "route.h"

int main(void)
{
    module1_init(); //注册模块1函数
    module2_init(); //注册模块2函数

    while(1)
    {
        module1_run(); //模块1发送数据
        module2_run(); //因为模块2是接收数据, 所以调不调用模块2空函数都无所谓
        sleep(1);
    }

    return 0;
}
```

root@ubuntu:/home/xzz/test/module# gcc -g -o demo main.c module1.c module2.c route.c

root@ubuntu:/home/xzz/test/module# ./demo

```
module2 rcv = 1111to2222
module2 rcv = 1111to2222
module2 rcv = 1111to2222
module2 rcv = 1111to2222
module2 rcv = 1111to2222
module2 rcv = 1111to2222
module2 rcv = 1111to2222
```

/*
 * 接收其它模块发来的数据
 */
static int module2_rcv(int src,int dst,char *buf,int len)
{
 if(dst != MODULE2_ID)
 printf("module2 rcv failed\n");
 else
 printf("module2 rcv = %s\n",buf);
}

模块 2 接收到模块 1 发来的数据。说白了就是模块 1 循环数组, 调用模块 2 module2_rcv

模块 1, 模块 2 双向通信

route.c 和 route.h 不变

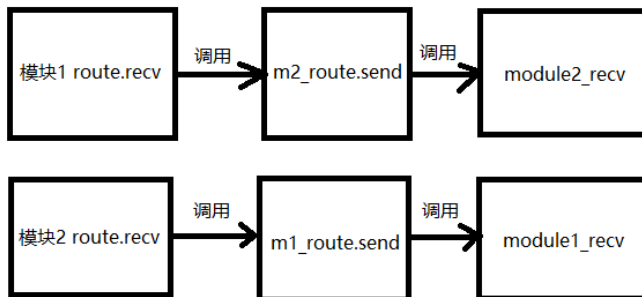
只需要修改 module2.c 就可以了

```
int module2_run(void)
{
    char *str = "22222to11111";

    m2_route.recv(MODULE2_ID,MODULE1_ID,str,strlen(str)); //模块2发送数据到模块1
}
```

加入发送数据函数

基本思想就是两个模块的接收函数相互调用



```
root@ubuntu:/home/xzz/test/module# ./demo
module2 recv = 1111to2222
module1 recv = 2222to11111
module2 recv = 1111to2222
module1 recv = 2222to11111
module2 recv = 1111to2222
module1 recv = 2222to11111
```

现在突然有个工程师, 想加入个模块 3 怎么办? 要求接受模块 1 的数据

cp module2.c module3.c

增加一个模块 3 的 C 文件

```
#include <stdio.h>
#include <string.h>
#include "route.h"
#include "module3.h"
static ROUTE_ITEM m3_route;
/*
 * 接收其它模块发来的数据
 */
static int module3_recv(int src,int dst,char *buf,int len)
{
    if(dst != MODULE3_ID)
        printf("module3 recv failed\n");
    else
        printf("module3 recv = %s\n",buf);
}
int module3_init(void)
{
    m3_route.src = MODULE3_ID; //自己本文件module3函数的ID
    m3_route.dst = MODULE1_ID; //默认要发送给某个模块的ID, 比如我要发数据给模块2(另外一个c文件, module1函数的ID)
    m3_route.recv = route_recv; //注意: 这儿不是接收, 是将数据送到路由进行转发, 用recv很奇怪, 这个recv是要理解成 route.c接收, 所以这里的route_recv函数是调用的route.c里面的recv
    m3_route.send = module3_recv; //默认要发送给某个模块的ID, 比如我要发数据给模块1(另外一个c文件, module1函数的ID)

    route_register(&m3_route); //将本文件建立的m1_route路由模块注册进路由表
}
int module3_run(void)
{
    char *str = "3333333to11111";

    m3_route.recv(MODULE3_ID,MODULE1_ID,str,strlen(str)); //模块3发送数据到模块1
}
```

```
#include <unistd.h>
#include "module1.h"
#include "module2.h"
#include "module3.h"
#include "route.h"

int main(void)
{
    module1_init(); //注册模块1函数
    module2_init(); //注册模块2函数
    module3_init(); //注册模块3函数

    while(1)
    {
        module1_run(); //模块1发送数据
        module2_run(); //模块2发数据到模块1
        module3_run(); //模块3发数据到模块1

        sleep(1);
    }

    return 0;
}
```

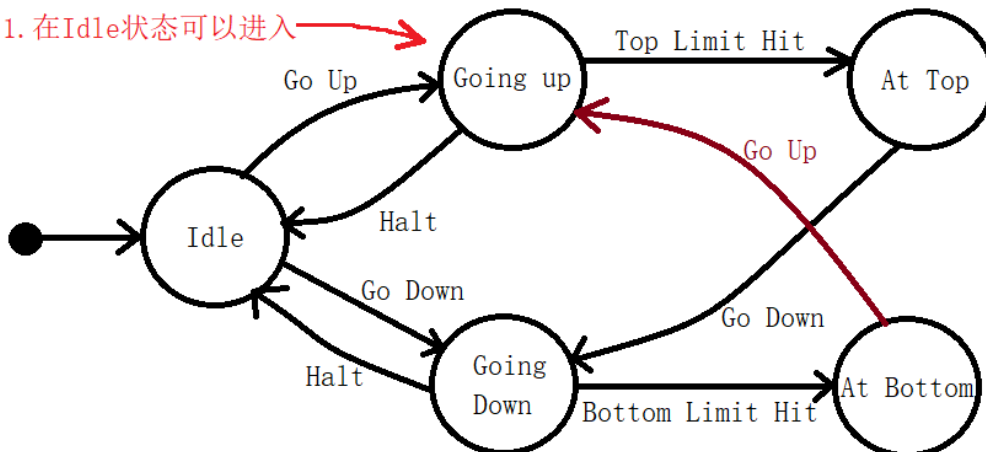
```
root@ubuntu:/home/xzz/test/module# ./demo
module2 recv = 1111to2222
module1 recv = 2222to11111
module1 recv = 3333333to11111
module2 recv = 1111to2222
module1 recv = 2222to11111
module1 recv = 3333333to11111
module2 recv = 1111to2222
module1 recv = 2222to11111
module1 recv = 3333333to11111
```

其实路由就是相互调用

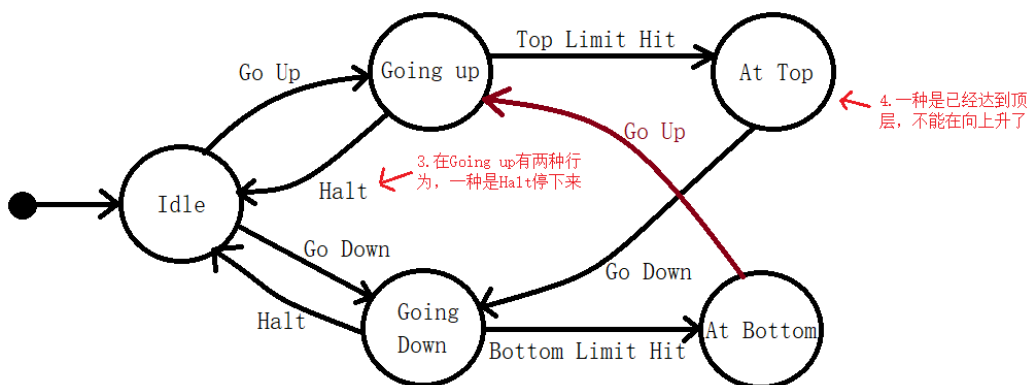
C 语言实现有限状态机(FSM)

以电梯状态机为例，实现 FSM

1. 在Idle状态可以进入



2. 在Idle状态也可以进入



```
#include <stdio.h>

enum state_codes {_idle = 0, _goingup = 1, _goingDown = 2, _AtTop = 3, _AtBottom = 4,
                 _malfunction = 5, _unexpected = 6, _end = 7}; //定义多个不同的状态(系统默认都是 int)

enum ret_codes {up = 0, down = 1, halt = 2, top = 3, bottom = 4, fail = 5, quit = 6}; //定义操作类型

enum state_codes x = _idle; //初始化枚举变量为_idle 模式

int target_floor_number = 8; //电梯默认目标设置
int current_floor_number = 1; //电梯当前默认在第1层
int accumulated_floor_number = 0;

#define TOP_FLOOR 9 //电梯顶层，一共9层
#define BOTTOM_FLOOR 1 //电梯底层，第1层

int event_idle(void)
{
    //target_floor_number = 得到一个动作
    if(current_floor_number < target_floor_number) //如果得到的动作是3，当前层数 current_floor_number = 0，表示电梯向上运行
    {
        return up; //返回电梯向上操作
    }
    else if(current_floor_number > target_floor_number) //如果得到的动作是3，当前层数 current_floor_number = 4.电梯执行向下操作
    {
        return down; //返回电梯向下操作
    }
    else if(current_floor_number == target_floor_number) //电梯就在当前层
    {
        return halt; //电梯停止运行
    }
    else //什么都不操作
    {
        return quit + 1; // 返回7
    }
}
```

每个状态对应一个函数

int (* event[])(void) = {event_idle, event_goingUp, event_goingDown, event_malfunction, event_end, event_unexpected};

每个状态函数执行完之后，返回的下一个状态

```

int event_goingUp(void)
{
    if(accumulated_floor_number > 100)
        return fail;
    else if(TOP_FLOOR == current_floor_number) //如果当前电梯在顶层 TOP_FLOOR, 返回 top
        return top;
    else if(BOTTOM_FLOOR == current_floor_number) //如果当前电梯在底层 BOTTOM_FLOOR, 返回 bottom
        return bottom;
    else if(current_floor_number < target_floor_number) //如果当前电梯层 < 目标设置的层,电梯向上
        return up;
    else if(current_floor_number == target_floor_number) //电梯就在当前层
        return halt; //电梯停止运行
    else
        return quit;
}

int event_goingDown(void)
{
    if(accumulated_floor_number > 100)
        return fail;
    else if(TOP_FLOOR == current_floor_number) //如果当前电梯在顶层 TOP_FLOOR, 返回 top
        return top;
    else if(BOTTOM_FLOOR == current_floor_number) //如果当前电梯在底层 BOTTOM_FLOOR, 返回 bottom
        return bottom;
    else if(current_floor_number > target_floor_number) //如果当前电梯层 > 目标设置的层,电梯向下
        return down; //返回电梯向下操作
    else if(current_floor_number == target_floor_number) //电梯就在当前层
        return halt; //电梯停止运行
    else
        return quit;
}

int event_atTop(void)
{
    printf("atTop\n");//当前电梯在顶层
    //target_floor_number = 得到一个动作
    if(current_floor_number > target_floor_number) //如果当前电梯层 > 目标设置的层,电梯向下
        return down; //返回电梯向下操作
    else if(current_floor_number == target_floor_number) //电梯就在当前层
        return halt; //电梯停止运行
}

int event_atBottom(void)
{
    printf("atBottom\n");
}

int event_malfunction(void)
{
    printf("malfunction\n");
}

int event_end(void)
{
    printf("end\n");
}

int event_unexpected(void)
{
    printf("unexpected\n");
}

int (* event[])(void) = {event_idle,event_goingUp,event_goingDown,event_atTop,event_atBottom,
    event_malfunction,event_end,event_unexpected}; //函数指针数组, 存放多个函数地址

/*状态迁移表*/
int lookup_transitions[][7] = {
    // up      down      halt      top      bottom      fail      quit
    [_idle]   = {_goingup, _goingDown, _idle, _unexpected, _unexpected, _malfunction, _end},
    [_goingup] = {_goingup, _unexpected, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_goingDown] = {_unexpected, _goingDown, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_AtTop]   = {_unexpected, _goingDown, _AtTop, _unexpected, _unexpected, _malfunction, _end},
    [_AtBottom] = {_goingup, _goingDown, _AtBottom, _unexpected, _unexpected, _malfunction, _end},
    [_malfunction] = {_end, _end, _end, _end, _end, _end, _end},
    [_unexpected] = {_end, _end, _end, _end, _end, _end, _end},
};

```

不同状态执行不同的函数, 在主循环轮询使用

比如当前执行函数是_idle 状态对应的函数, 那么_idle 函数返回之后, 下一个要执行的函数在这个表里面找

```

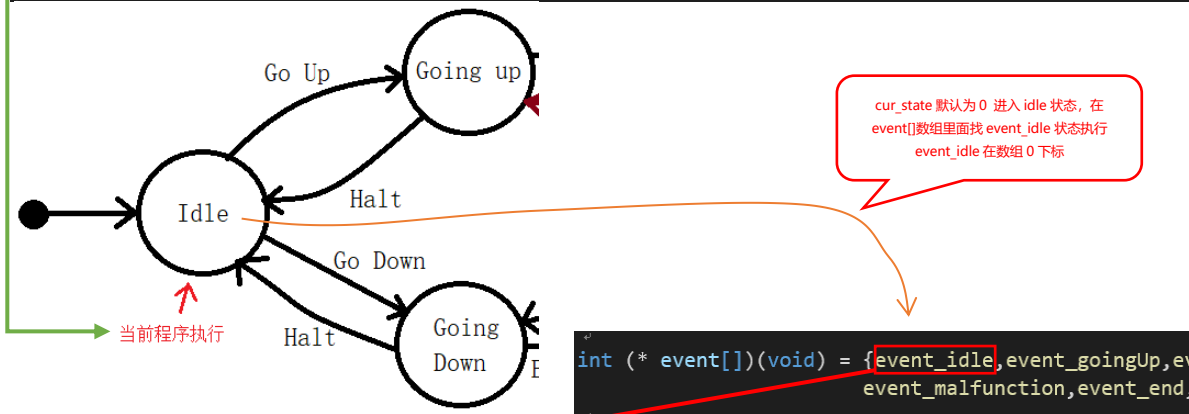
int main()
{
    enum state_codes cur_state = _idle; //状态机初始化为_idle 0 状态
    enum ret_codes rc; //接收返回值, 表示下一次要执行哪个 event 事件函数
    int (* state_func)(void); //定义函数指针, 用来承接要执行的函数

    while (1)
    {
        state_func = event[cur_state]; //根据 lookup...返回的 cur_state 状态, 决定下一个要执行的函数赋值给 state_func
        rc = state_func();
        if(_end == cur_state) //如果当前状态是_end, 退出死循环
            break;
        cur_state = lookup_transitions[cur_state][rc];
    }

    return 0;
}

```

cur_state 默认为 0



```

int event_idle(void)
{
    //target_floor_number = 得到一个动作
    if(current_floor_number < target_floor_number) //如果得到的动作是 3, 当前层数 current_floor_number = 0, 表示电梯向上运行
        return up; //返回电梯向上操作
    else if(current_floor_number > target_floor_number) //如果得到的动作是 3, 当前层数 current_floor_number = 4. 电梯执行向下操作
        return down; //返回电梯向下操作
    else if(current_floor_number == target_floor_number) //电梯就在当前层
        return halt; //电梯停止运行
    else //什么都不操作
        return quit; // 返回 6
}

```

如果没有操作, 返回 6

```

while (1)
{
    state_func = event[cur_state]; //根据 lookup...返回的 cur_state 状态, 决定下一个要执行的函数赋值给 state_func
    rc = state_func(); //rc 得到 6
    if(_end == cur_state) //如果当前状态是_end, 退出死循环
        break;
    cur_state = lookup_transitions[cur_state][rc]; // [0][6] 也就是 [_idle][6]
}

```

[0][6]在状态表就是 _end

```

int lookup_transitions[][7] = {
    // up      down      halt      top      bottom      fail      quit
    [_idle]   = {_goingup, _goingDown, _idle, _unexpected, _unexpected, _malfunction, _end},
    [_goingup] = {_goingup, _unexpected, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_goingDown] = {_unexpected, _goingDown, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_AtTop]   = {_unexpected, _goingDown, _AtTop, _unexpected, _unexpected, _malfunction, _end},
    [_AtBottom] = {_goingup, _goingDown, _AtBottom, _unexpected, _unexpected, _malfunction, _end},
    [_malfunction] = {_end, _end, _end, _end, _end, _end, _end},
    [_unexpected] = {_end, _end, _end, _end, _end, _end, _end},
};

```

下一次循环

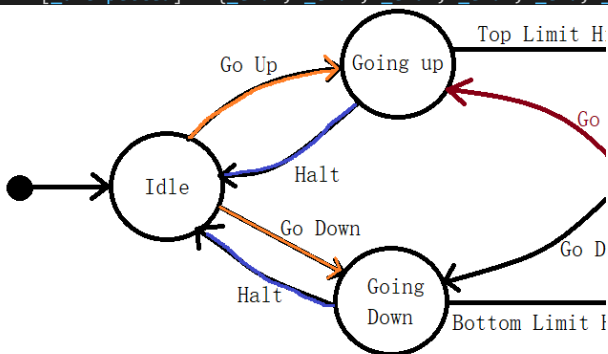
```
while (1)
{
    //因为 cur_state = 6 就是_end
    state_func = event[cur_state]; //根据 lookup...返回的 cur_state 状态，决定下一个要执行的函数赋值给 spare_func
    rc = state_func(); //rc 得到 6
    if(_end == cur_state) //如果当前状态是_end，退出死循环
        break;
    cur_state = lookup_transitions[cur_state][rc]; //[0][6] 也就是[_idle][7]
}
```

```
enum state_codes {_idle = 0, _goingup = 1, _goingDown = 2, _AtTop = 3, _AtBottom = 4, _malfunction = 5, _unexpected = 6, _end = 7};

int (* event[])(void) = {event_idle, event_goingUp, event_goingDown, event_atTop, event_atBottom,
    event_malfunction, event_end, event_unexpected};
```

所以每次循环的代码都是在状态迁移表里面去寻找，状态迁移表的设计，就决定了状态机的逻辑。

```
int lookup_transitions[][7] = {
    // up      down      halt      top      bottom      fail      quit
    [_idle]   = {_goingup, _goingDown, _idle, _unexpected, _unexpected, _malfunction, _end},
    [_goingup] = {_goingup, _unexpected, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_goingDown] = {_unexpected, _goingDown, _idle, _AtTop, _AtBottom, _malfunction, _end},
    [_AtTop]   = {_unexpected, _goingDown, _AtTop, _unexpected, _unexpected, _malfunction, _end},
    [_AtBottom] = {_goingup, _goingDown, _AtBottom, _unexpected, _unexpected, _malfunction, _end},
    [_malfunction] = {_end, _end, _end, _end, _end, _end, _end},
    [_unexpected] = {_end, _end, _end, _end, _end, _end, _end},
}
```



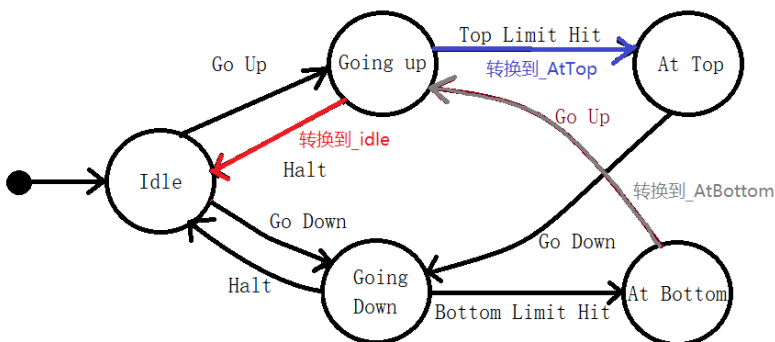
```
[_idle] = {_goingup, _goingDown, _idle, _unexpected,
```

Idle模式有Go Up----->Going up

Idle模式也有Go Down---->Going Down

Idle模式也有Going Up 使用Halt ----->Idle

你看[_idle] 就只有三状态可以填入表中



```
[_goingup] = {_goingup, _unexpected, _idle, _AtTop, _AtBottom, _malfunction, _end},
```

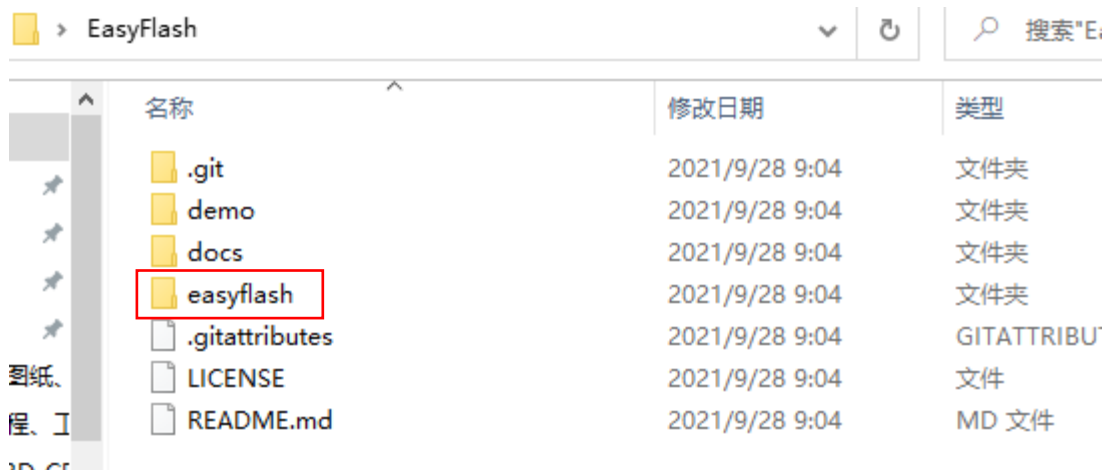
只要把执行函数全部写好，至于哪个函数先执行，哪个函数后执行，可以根据状态图规则，将状态图模型写入状态标志，方便操作

这就是 FSM 有限状态机，为了编程方便而设计

EasyFlash 存储库使用

源代码下载地址: <https://gitee.com/Armink/EasyFlash>

demo	【删除】多余的断言检查。	3年前
docs	【更新】图片素材	11个月前
easyflash	[修复]注释符不匹配	1年前



移植 easyflash 目录到 STM32 工程

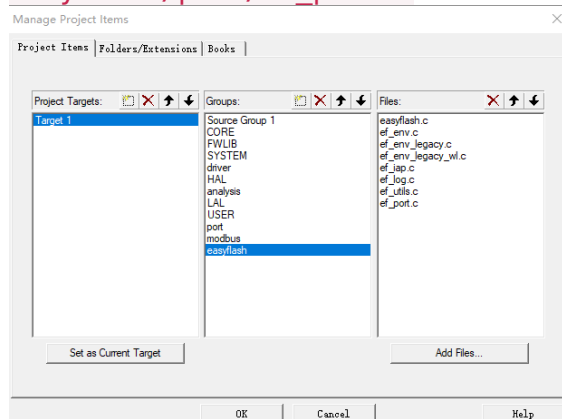
添加下面 4 个文件到工程

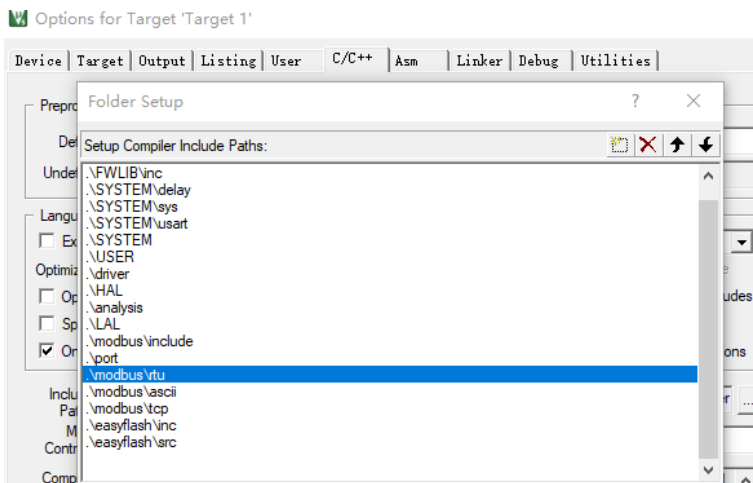
easyflash/src/easyflash.c

easyflash/src/ef_env.c

easyflash/src/ef_utils.c

easyflash/port/ef_port.c





加入 easyflash\inc 目录和 easyflash\src 目录

```
compiling mb.c...
compiling ef_utils.c...
compiling ef_port.c...
easyflash\port\ef_port.c(86): error: #29: expected an expression
    EF_ASSERT(addr % EF_ERASE_MIN_SIZE == 0);
easyflash\port\ef_port.c: 0 warnings, 1 error
".\out\STM32F4_flash.axf" - 31 Error(s), 63 Warning(s).
```

编译出现很多错误

打开 ef_cfg.h 文件, 修改里面的配置

```
#define EF_ENV_VER_NUM 0 /* @not
```

添加版本号, 我随便写的 0

根据实际情况填写最小擦除粒度, 我是用的是 F103rb 所以我写的是 1024, 如果你使用的是 F103ze 就应该填 2048, 根据实际情况进行填写

```
#define EF_ERASE_MIN_SIZE 512
```

EF_WRITE_GRAN 上面有注释, 如果是 stm32f4 就要填 8

```
/* the flash write granularity, unit: bit
 * only support 1(nor flash)/ 8(stm32f4)/ 32(stm32f1) */
#define EF_WRITE_GRAN 8 /* @note you must define
```

填写 easyflash 的开始地址, 我划分的分区是后面 64K, 所以我填的是 (64 * EF_ERASE_MIN_SIZE + 0x08000000)

```
/* backup area start address */
#define EF_START_ADDR 0x80e0000 /* @note you must define it for a value */

/* ENV area size. It's at least one empty sector for GC. So it's definition must more then or equal to EF_ERASE_MIN_SIZE */
#define ENV_AREA_SIZE EF_ERASE_MIN_SIZE*2 /* @note you must define it for a value if you used log */

/* saved log area size */
// #define LOG_AREA_SIZE /* @note you must define it for a value if you used log */
```

```
modbus\rtu\mbrtu.c(153): warning: #550-D: variable "xFrameR
      BOOL          xFrameReceived = FALSE;
modbus\rtu\mbrtu.c: 1 warning, 0 errors
compiling mb.c...
linking...
Program Size: Code=6364 RO-data=424 RW-data=268 ZI-data=3324
FromELF: creating hex file...
".\out\STM32F4_flash.axf" - 0 Error(s), 63 Warning(s).
Build Time Elapsed: 00:00:14
```

编译成功

填充 ef_port.c 硬件操作内容

ef_port_init(如果使用 FAL, 则需要配置,本例程不需要配置)

ef_port_read

ef_port_erase

ef_port_write

修改 ef_port.c 的向量表, 添加要使用的变量

未完后续补充.....

