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

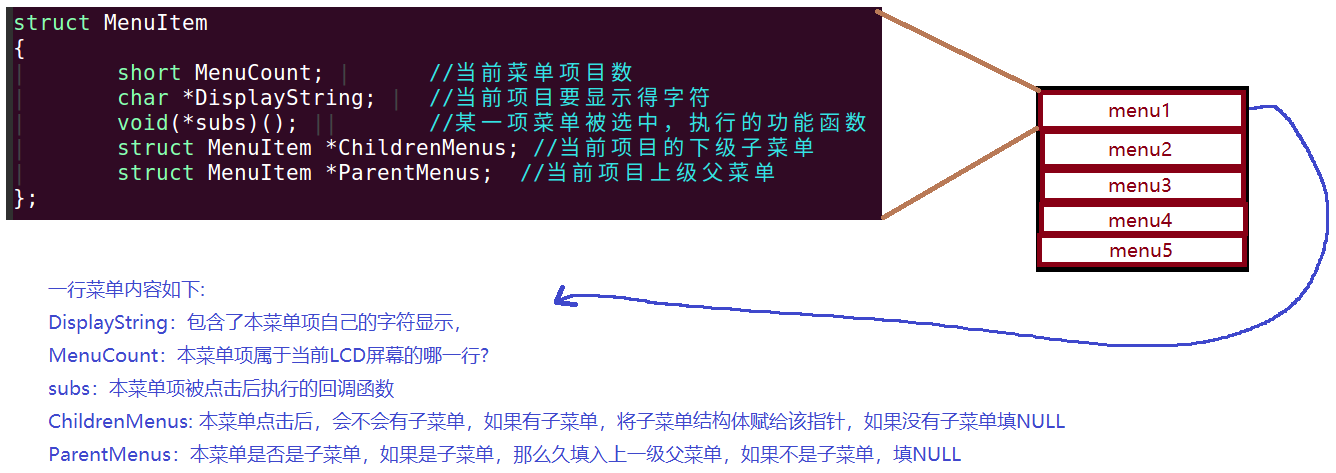
C语言实现装饰器模式，达到多态效果………………………………………………………………………………………...25

C语言路由架构............................................................................................................................................................26

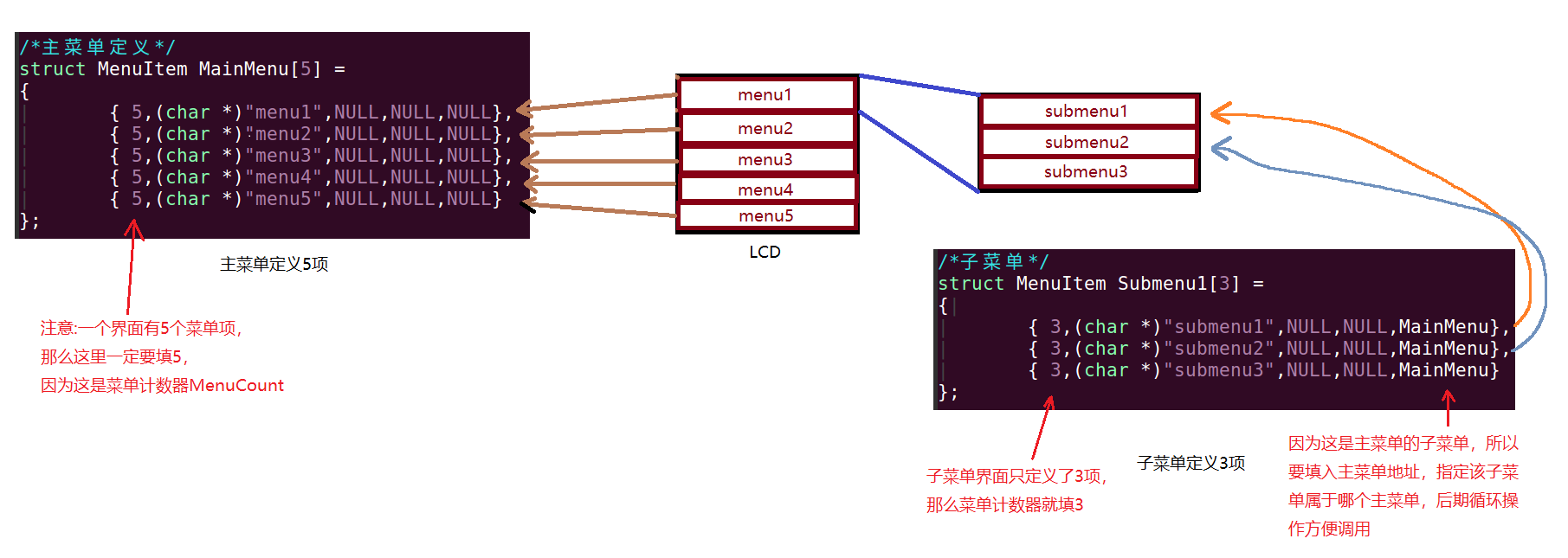
C语言实现有限状态机(FSM)....................................................................................................................................32

EasyFlash存储库使用(未完成).................................................................................................................................36

**单片机菜单架构**



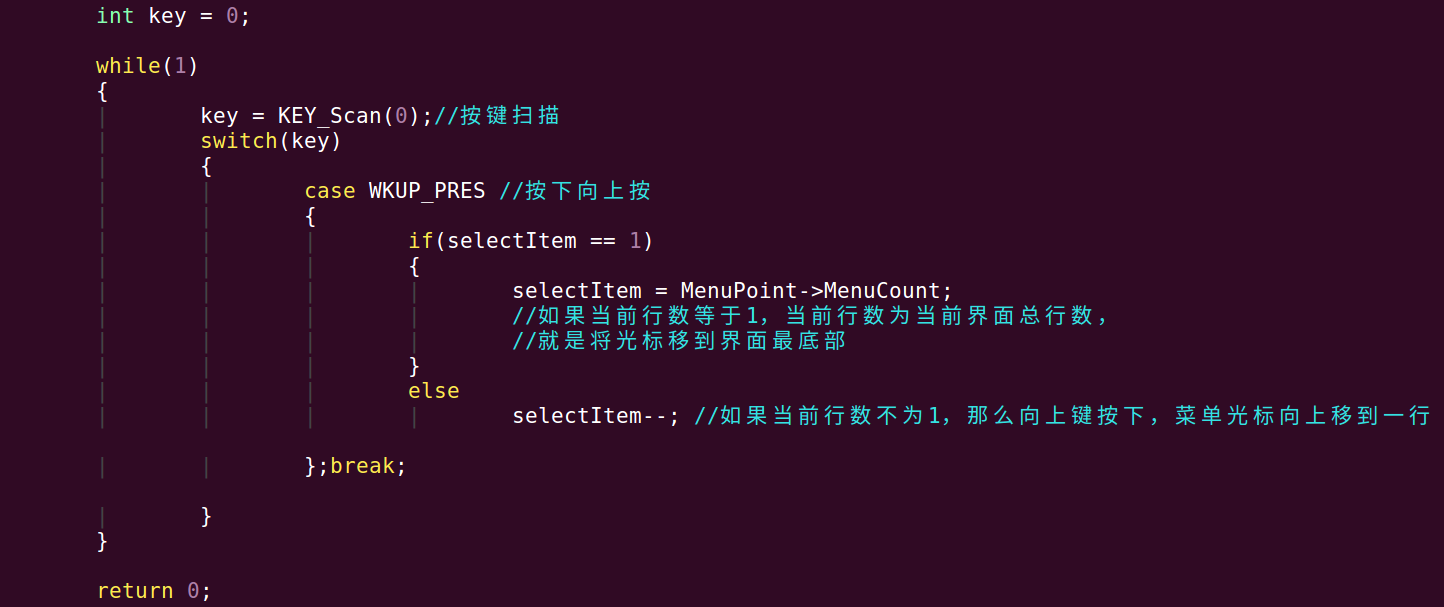
我们用这个结构体可以定义主菜单，也可以用来定义子菜单

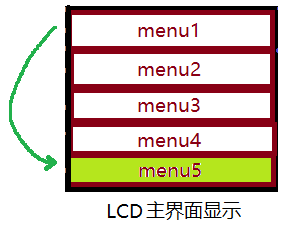


菜单定义的数据结构已经讲完，下面看看怎么调用这些数据结构



按键扫描程序操作主界面光标





如果再向上按，就让选项跳到本届面最大行



本来选项在第1行

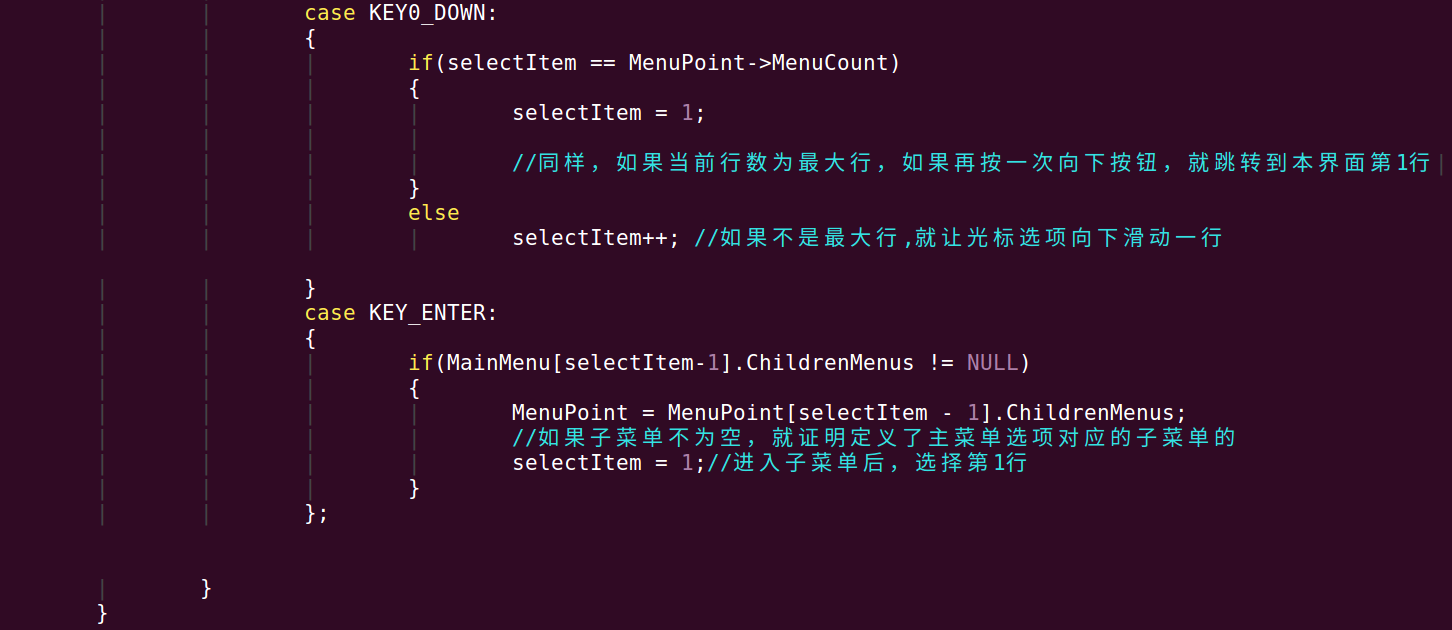


向下滑动一行



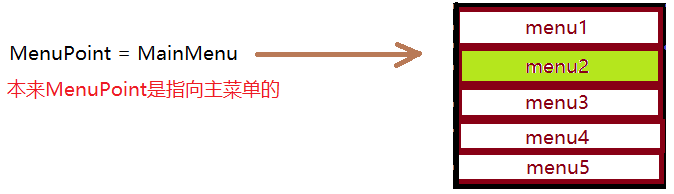
注意，忘记加分号了

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?

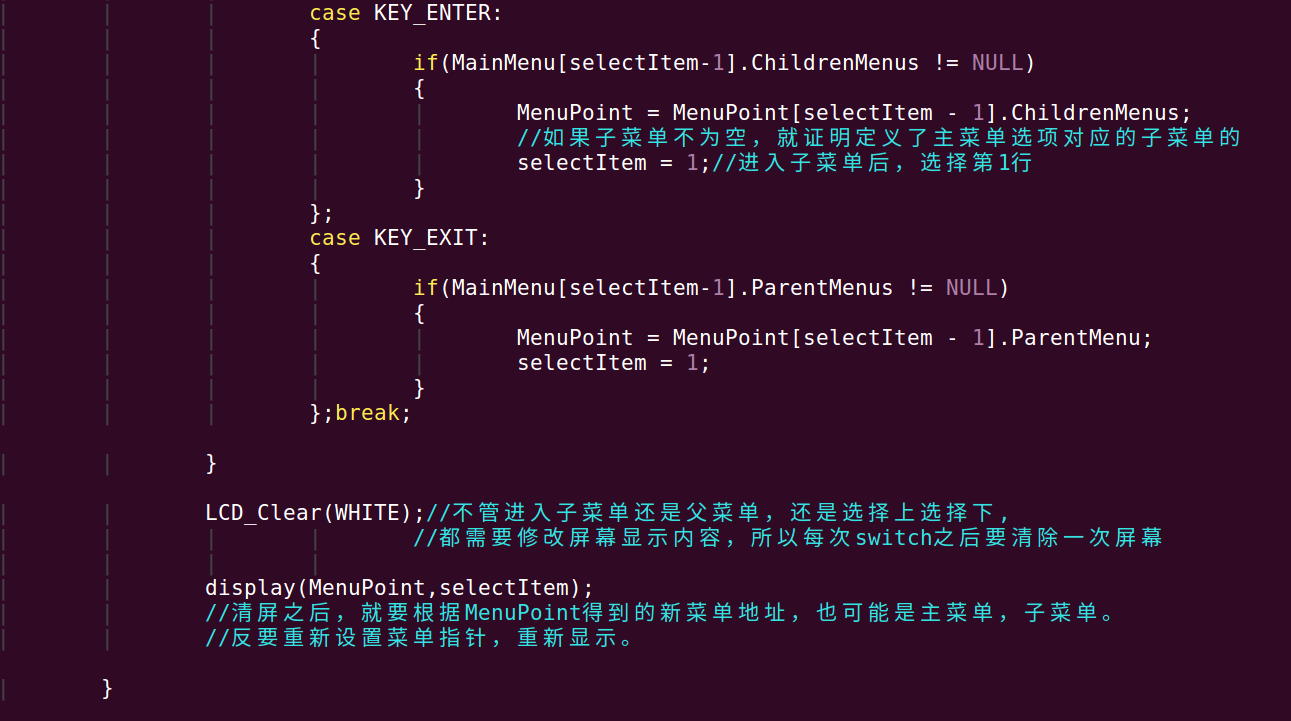




进入子菜单



本来指针指向的主界面



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 \*Menup,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 < Menup->MenuCount; j++)

{

show\_str(x,y,lcddev.width,24,Menup[j].DispalyString,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。

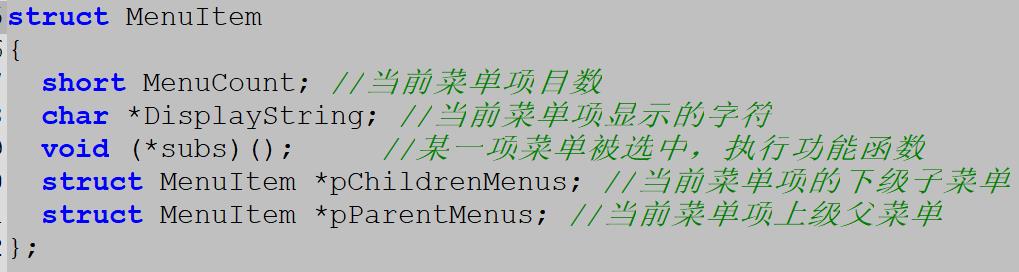
//反要重新设置菜单指针，重新显示。

}

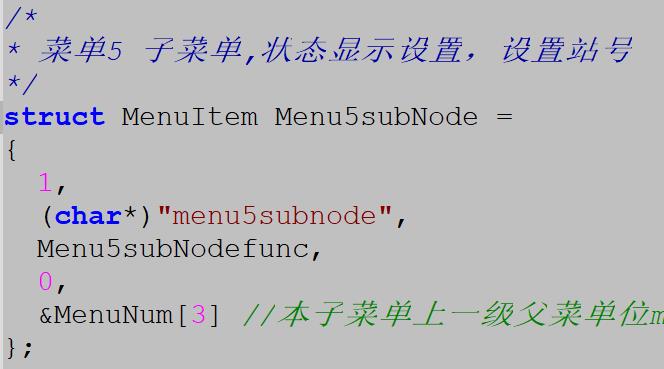
return 0;

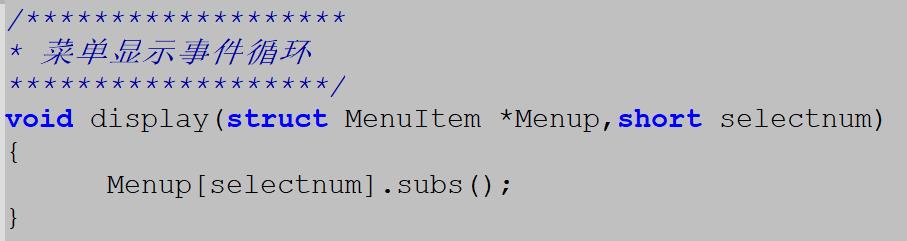
}

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



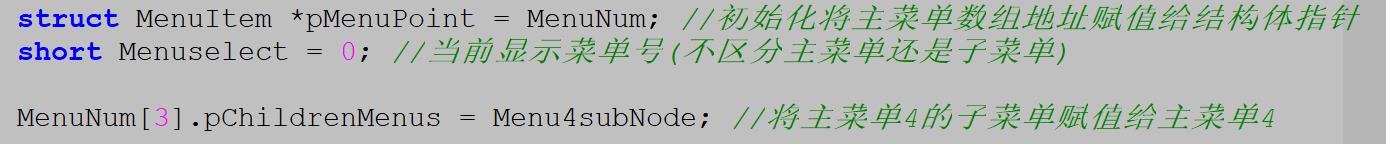
比如我定义了菜单一级界面

我二级菜单直接用结构体实现，这样可以吗?

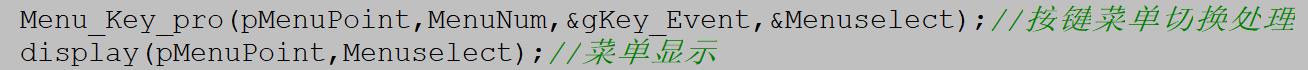


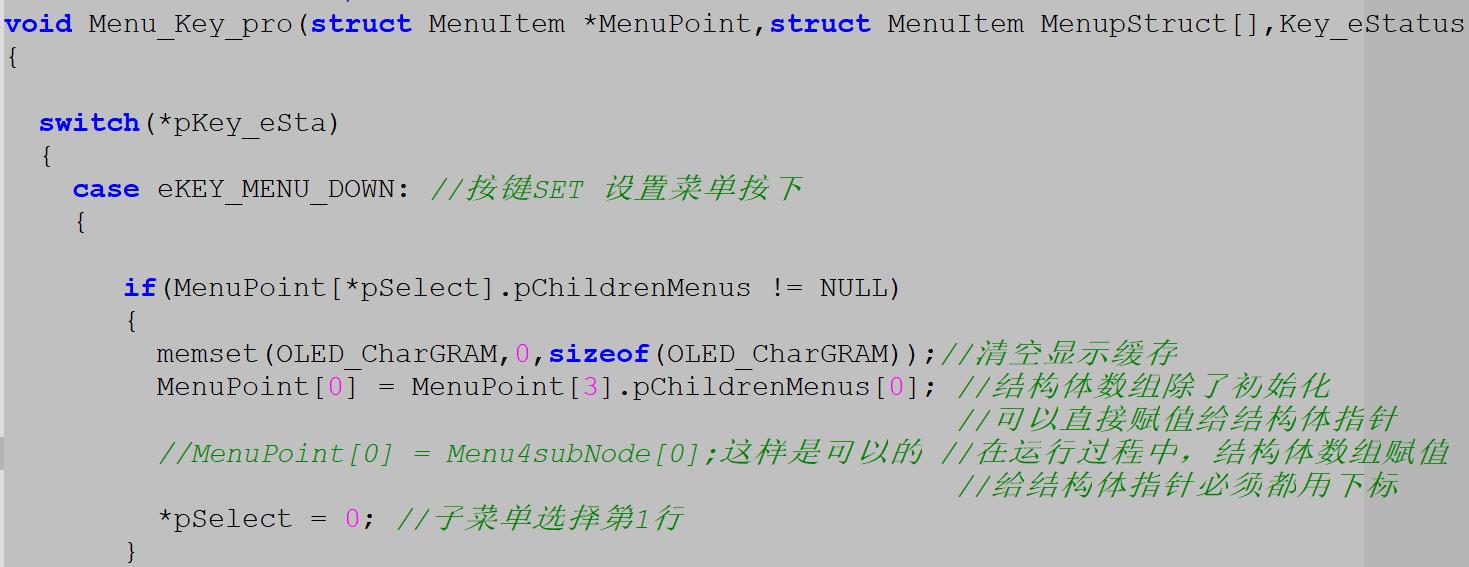
在菜单循环中，我们是根据selectnum的编号，循环菜单数组里面的内容

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。



下面在按键循环中，注意菜单结构体数组的传参方式。





**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\_Struct,\*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);

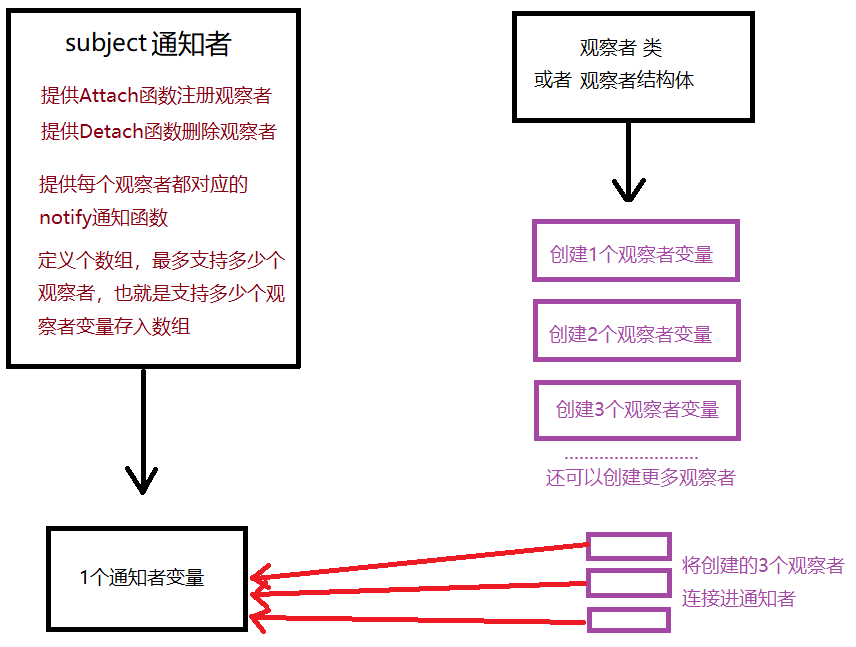
tasks[i].isRun = 0;//将标志位清零

}

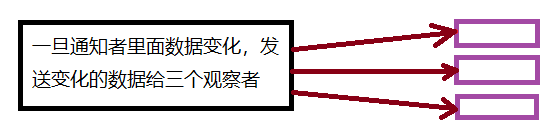
}

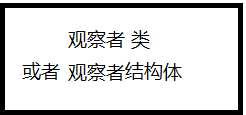
}

**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);                    //粉丝获得凤姐的新状态，被动获得

    void ( \*joinfans)(struct Fans fans);            //粉丝有权选择成为凤姐粉丝

    void ( \*quitfans)();                            //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);    //凤姐这边也可以增加删除粉丝(subject内容)

 //   void ( \*)delfans)();                            //to be done

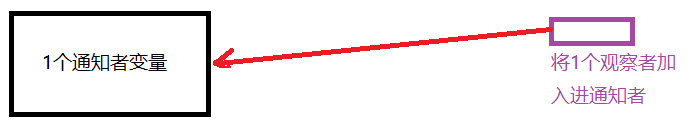
};

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



void addfans (struct Fengjie\* fengjie, struct Fans fans) { //将观察者加载进通知者

    fengjie->fanslist[fengjie->fansnum] = fans;

    fengjie->fansnum++;

}

\*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;

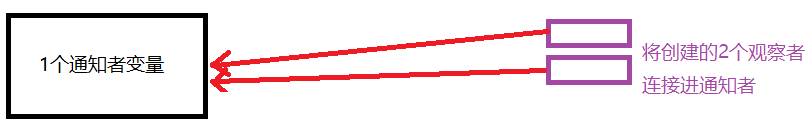
    fan2.fengjie = &fengjie, fan2.joinfans = joinfans, fan2.update = fan2update;

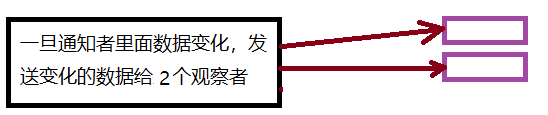
    fan1.joinfans (fan1); //将观察者1放入通知者

    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;

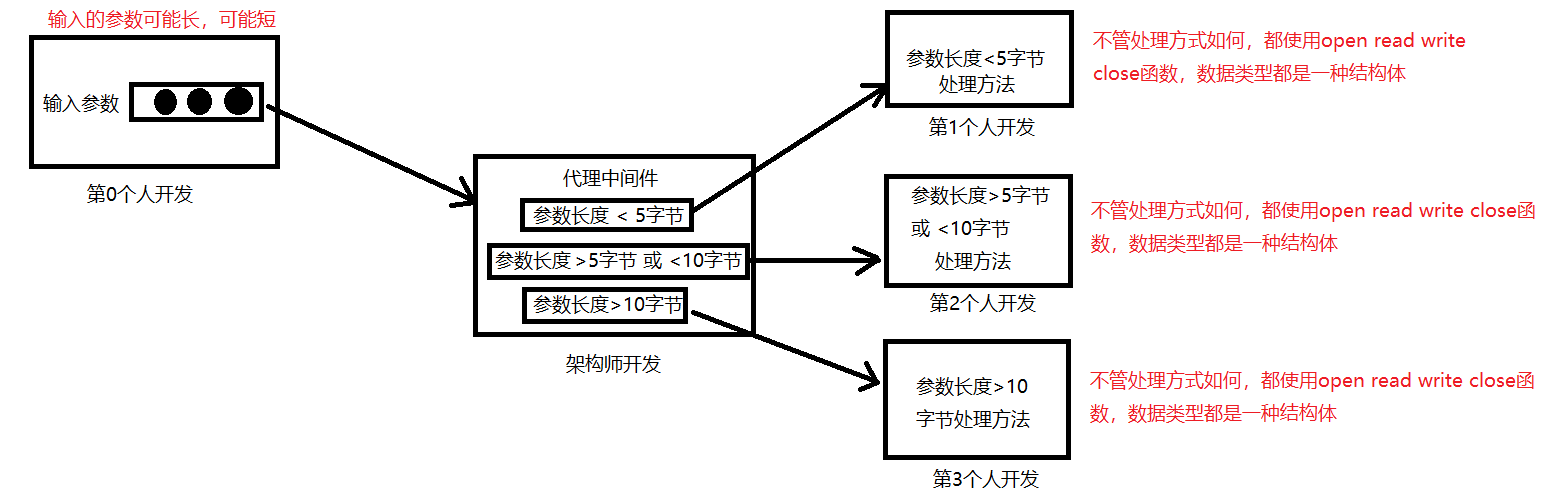
        pShoe->print\_shoe = print\_rubber\_shoe;

    }

    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个人定义的相同数据

    //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，经纪人在这里控制了一下

   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，经纪人在这里控制了一下

    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，经纪人在这里控制了一下

    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**



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 != xxx) && (tail != xxx) && (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 != xxx) && (tail != xxx) && (crc != (buffer[length - 3]) << 8 | buffer[length - 2]))

  {

    return 0;

  }

  for(entry = package\_items; entry->handle != NULL; ++entry)

  {

    if(frame->cmd == entry->cmd)

    {

    entry->handle(frame->data);

    break;

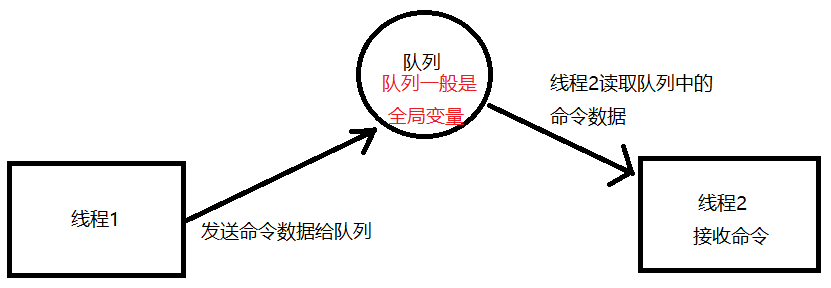
    }

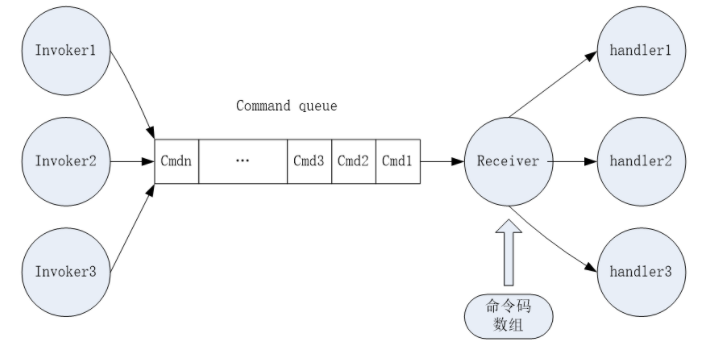
  }

  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

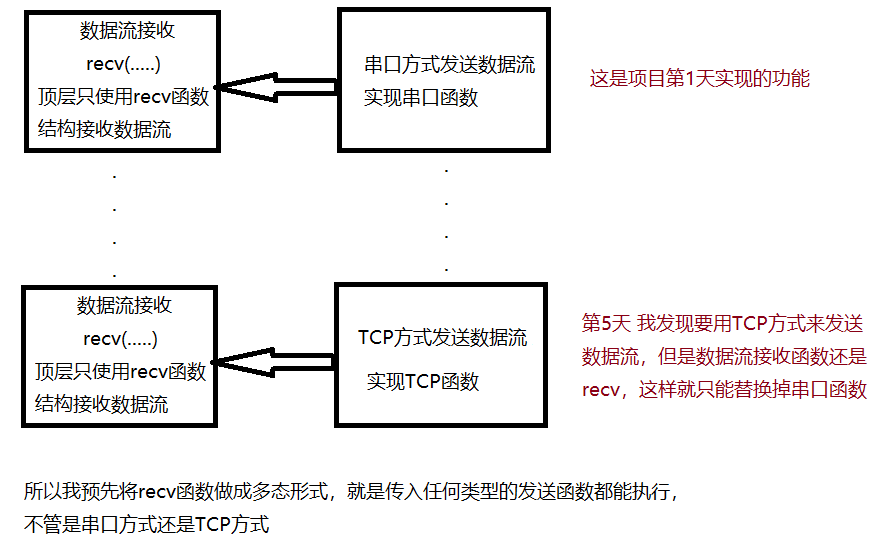
    invoker2();//发送命令消息到队列

    cmd\_receiver();//接受命令消息

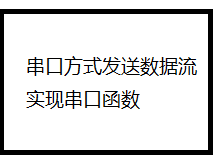
    return 0;

}

**C语言实现装饰器模式，达到多态效果**



#include <stdio.h>



/\*第1天实现的串口接收数据方式，将两个数据加起来\*/

int UARTrecv( int num1, int num2 )

{

    int sum = num1 + num2;

    printf( "UART %d+%d=%d\n", num1, num2, sum );

    return sum;

}



/\*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" );

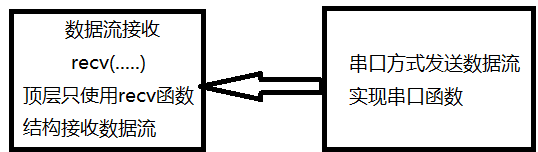
    //类似回调函数嘛，函数前后 可以进行装饰或者其他逻辑处理

    func( num1, num2 );

    printf( "走出装饰器\n" );

}

int main( int argc, char \*argv[] )



{

    Recv( UARTrecv, 20, 30 );

    return 0;

}

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\*/

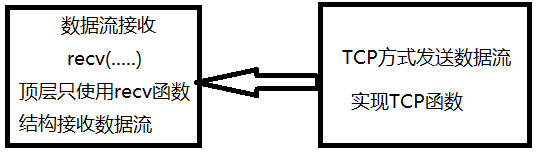
void Recv( int( \*func )( int, int ), int num1, int num2 )

{

    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[] )

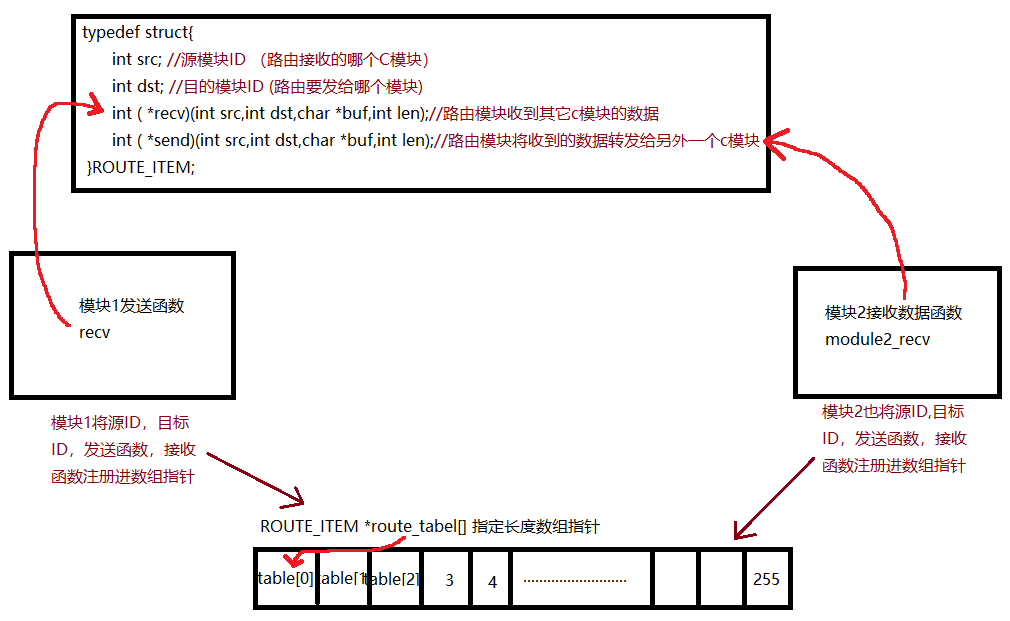
{

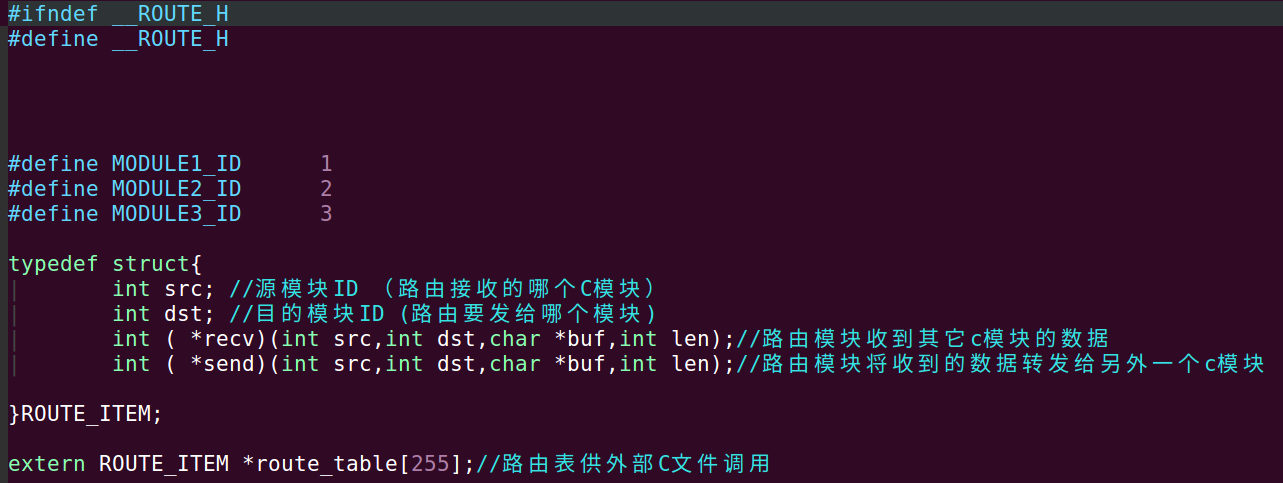
    Recv( TCPrecv, 20, 30 ); //同样的Recv接收函数不变，形参也不变

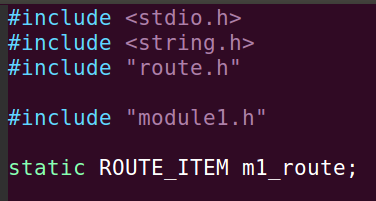
    return 0;

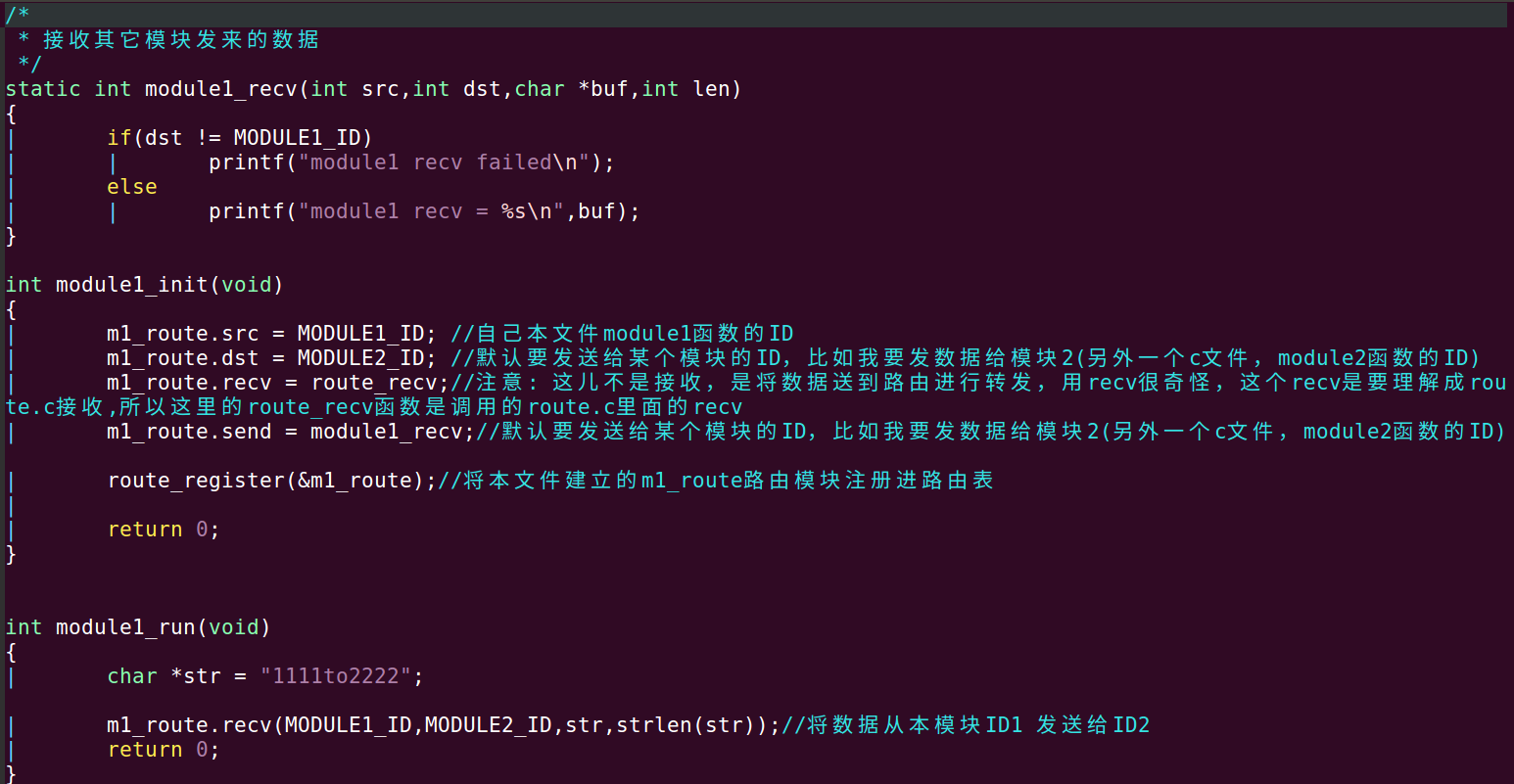
}

**C语言路由架构**



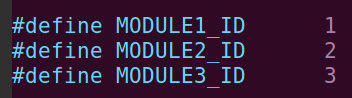


将定义的本模块结构放入数组，里面有本模块的ID，目标模块ID….



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

本模块要发送数据就调用路由的route\_recv回调函数



一共有多少个模块，必须预先在route..h文件中写号

module1.c

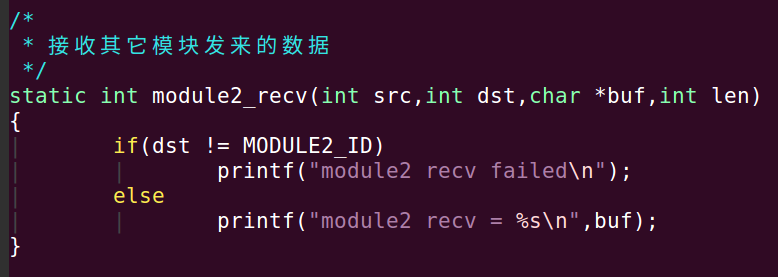


间接调用模块2.c里面的recv函数

其实本质就是模块1调用模块2的recv函数执行



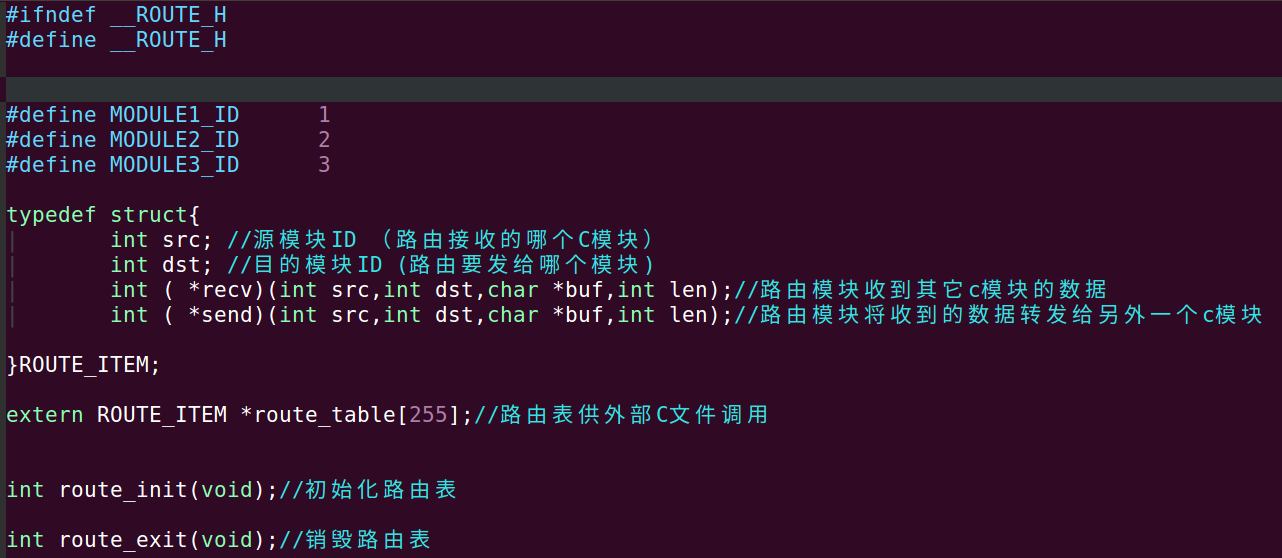
route.c

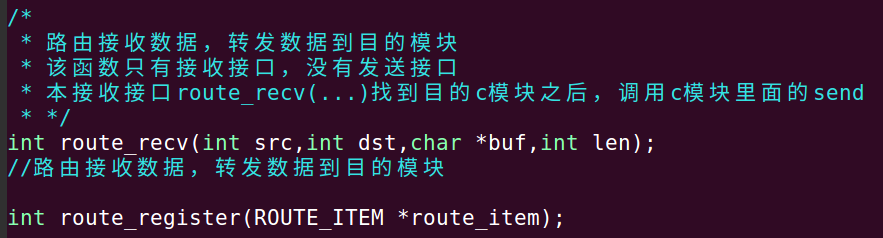


module2.c

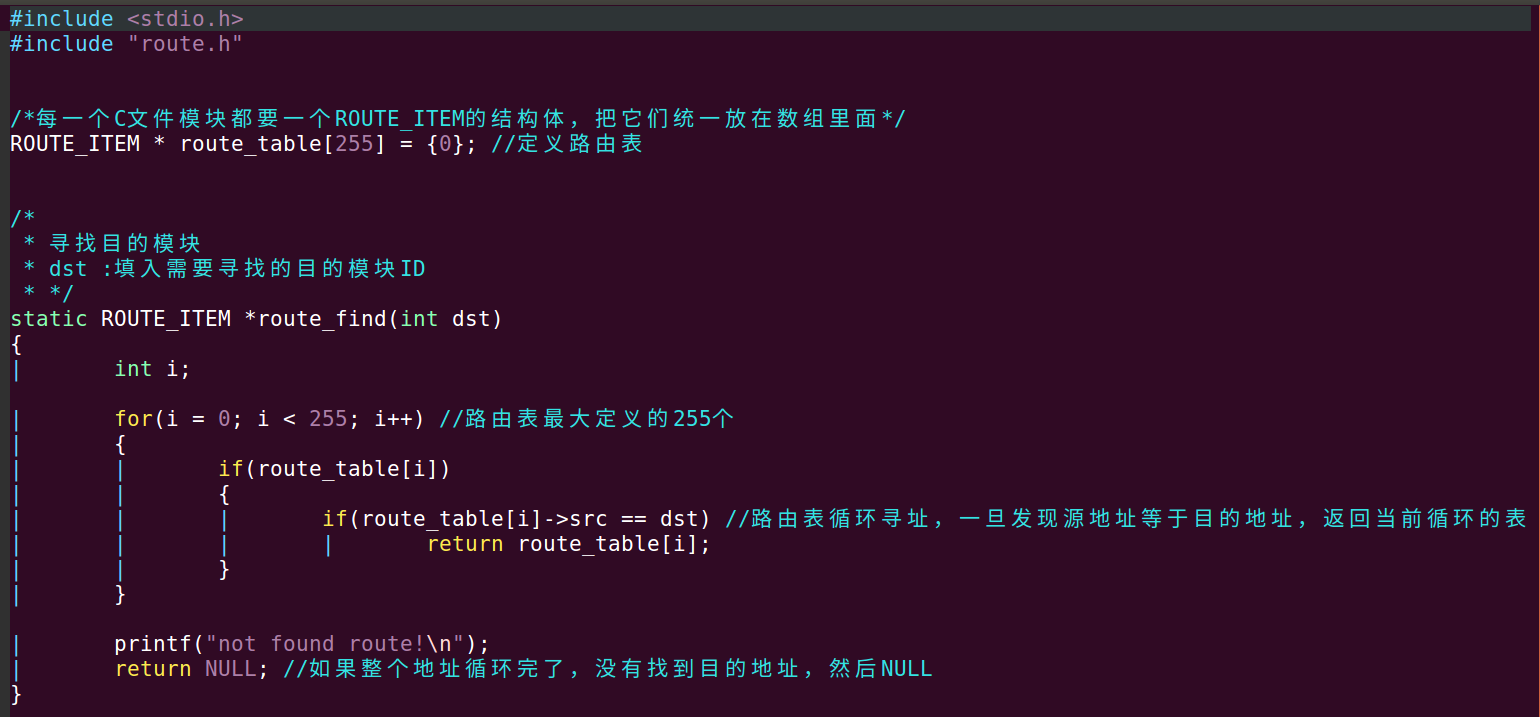
整个代码结构如下:

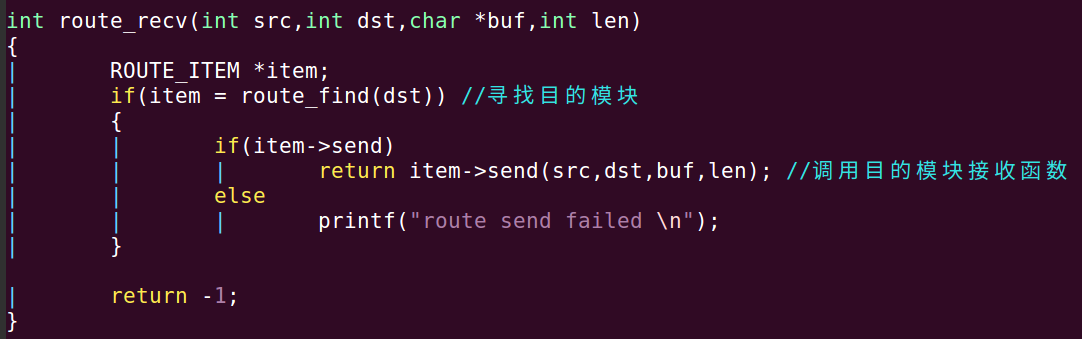
route.h



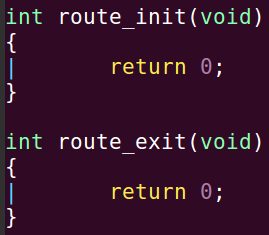


route.c



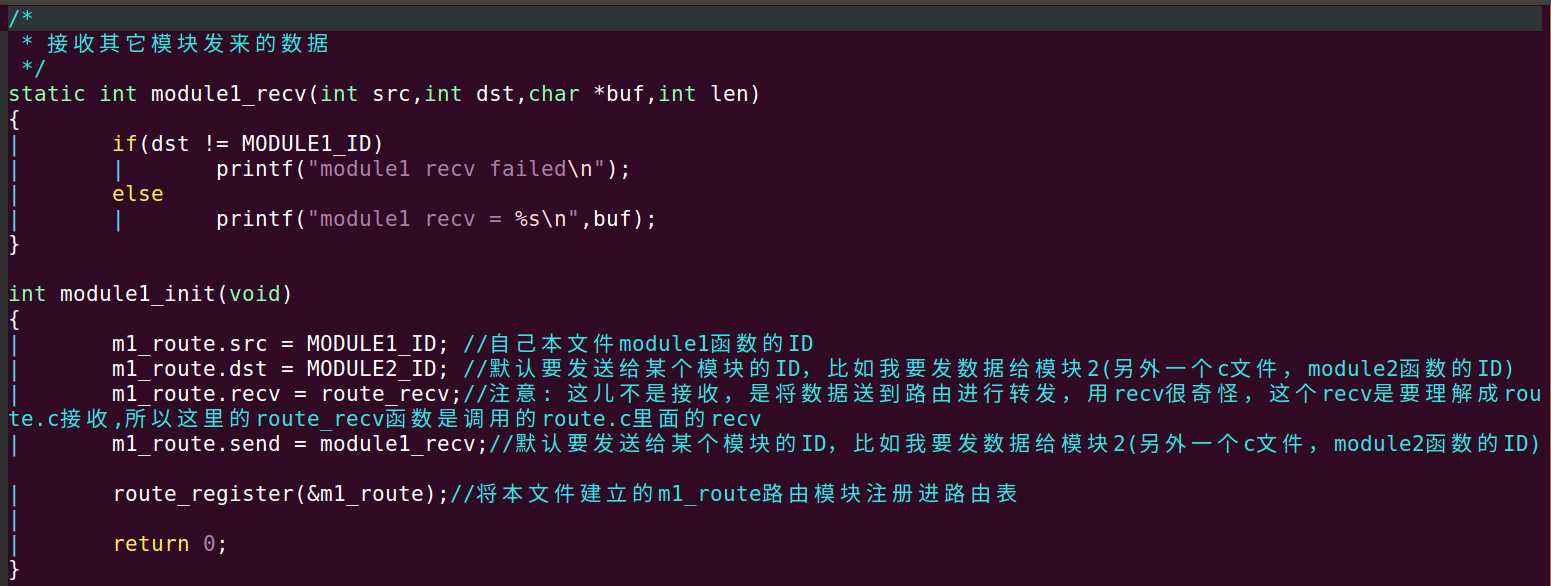




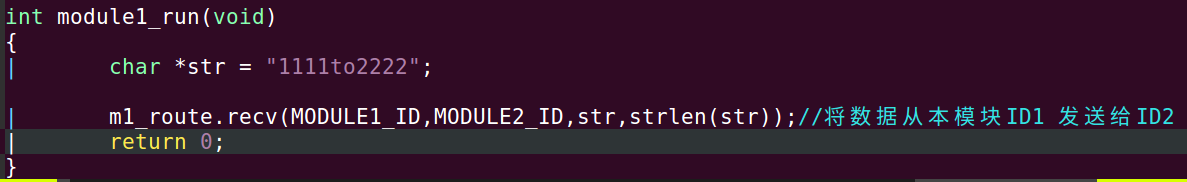
init和exit是用来初始化和销毁模块的，一般没用

下面介绍模块如何使用路由

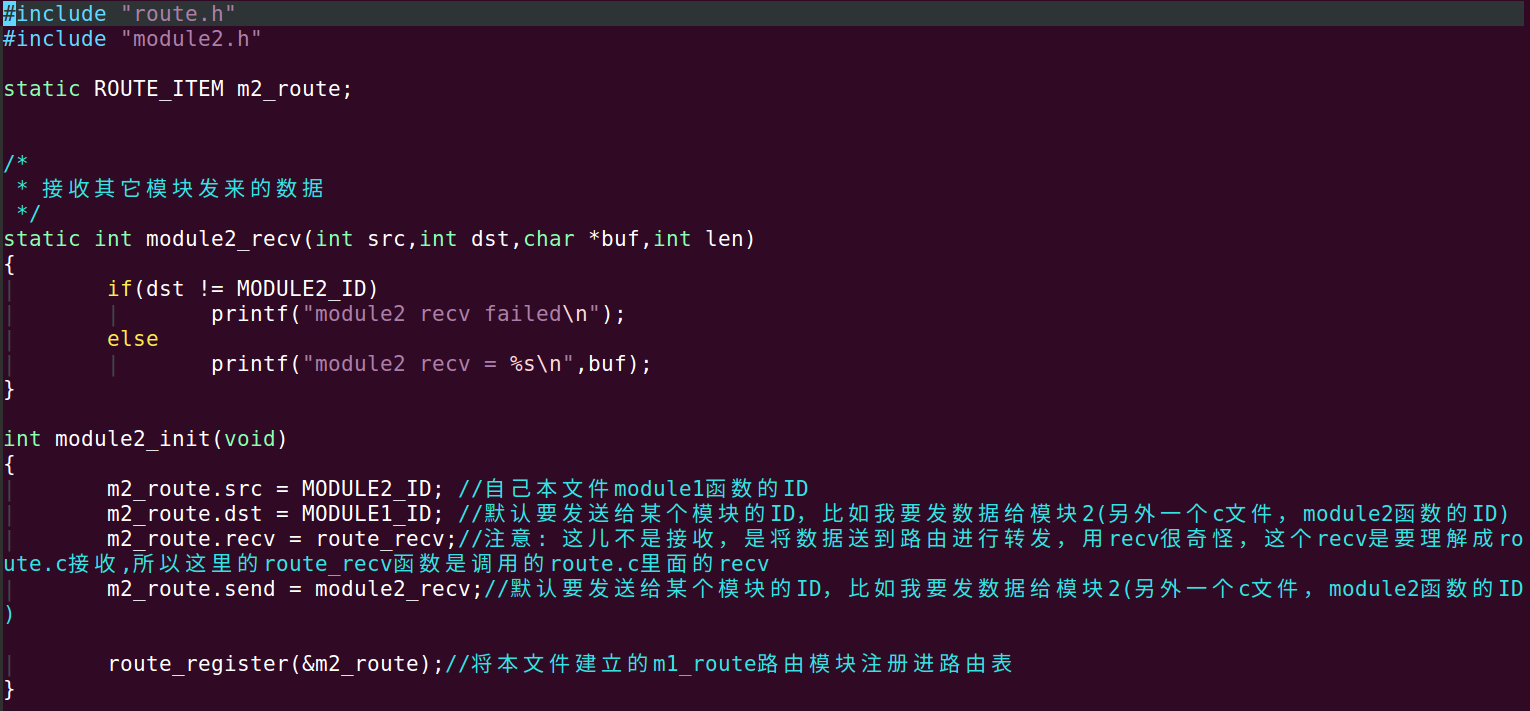
模块1



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

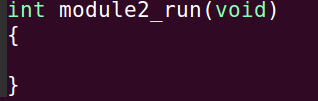


数流修改为模块2为源ID

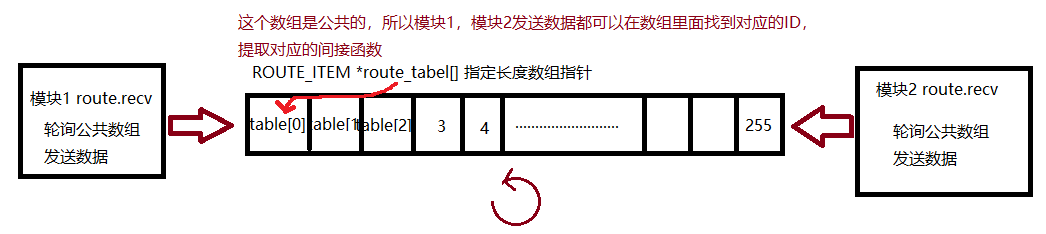
模块1位接收目的ID

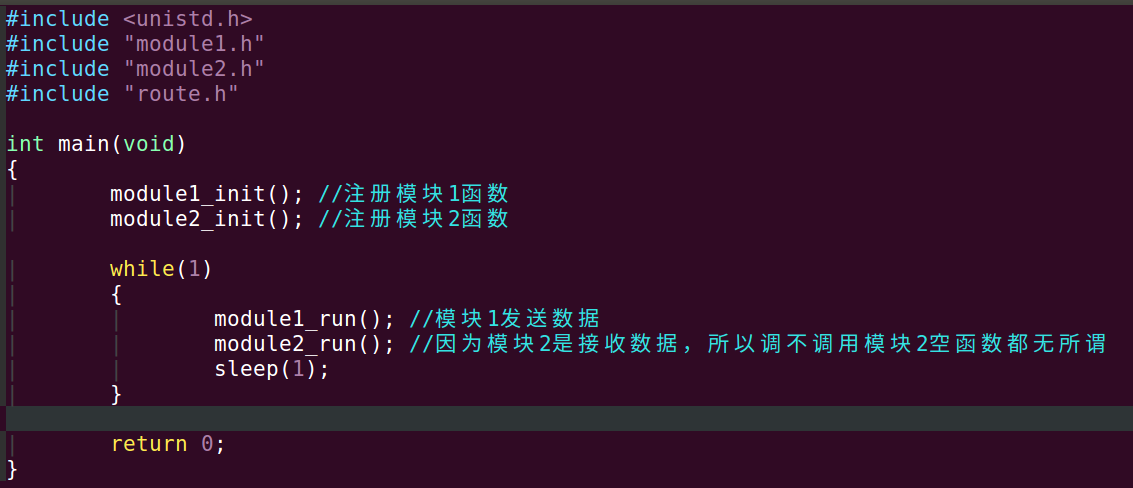
模块2的代码结构基本和模块1是一样的

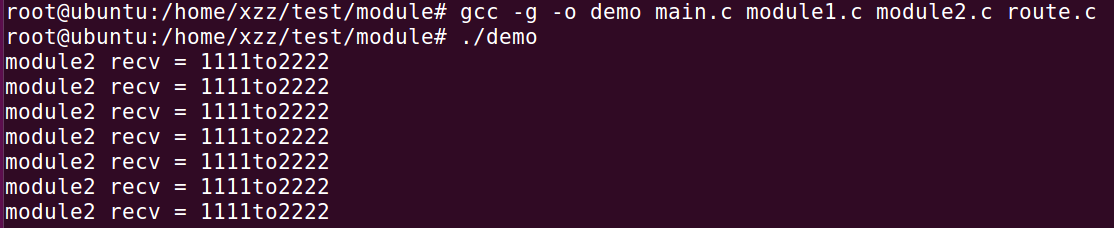
只是这里接收数据要判断是不是模块2得到了数据

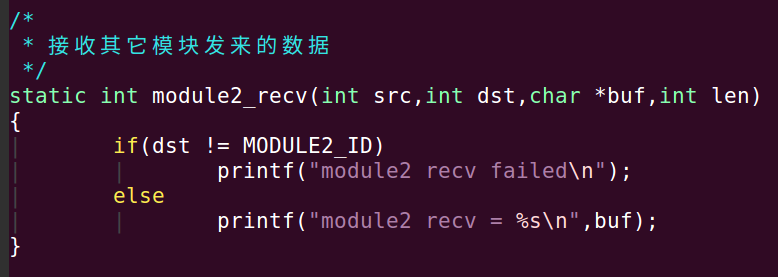


模块2现在做的接收模块1数据，所以空函数空转







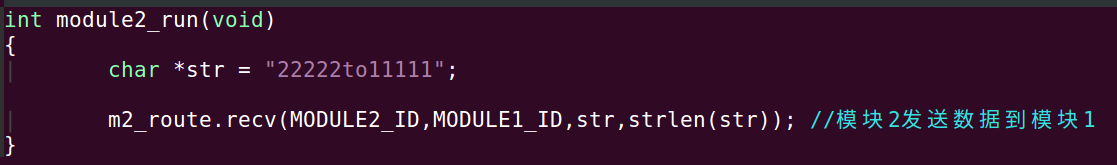


模块2接收到模块1发来的数据。说白了就是模块1循环数组，调用模块2 module2\_recv

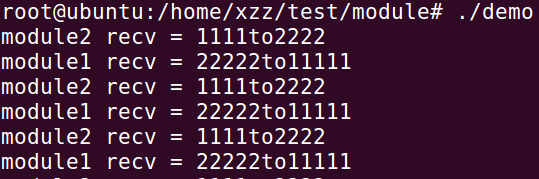
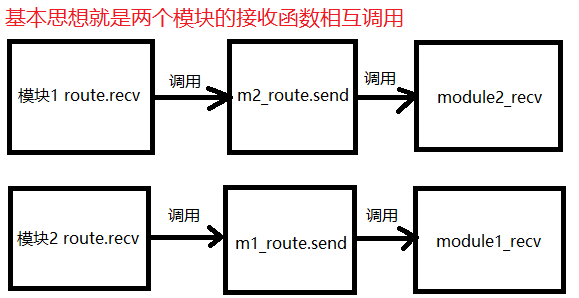
模块1，模块2双向通信

route.c 和route.h不变

只需要修改module2.c就可以了

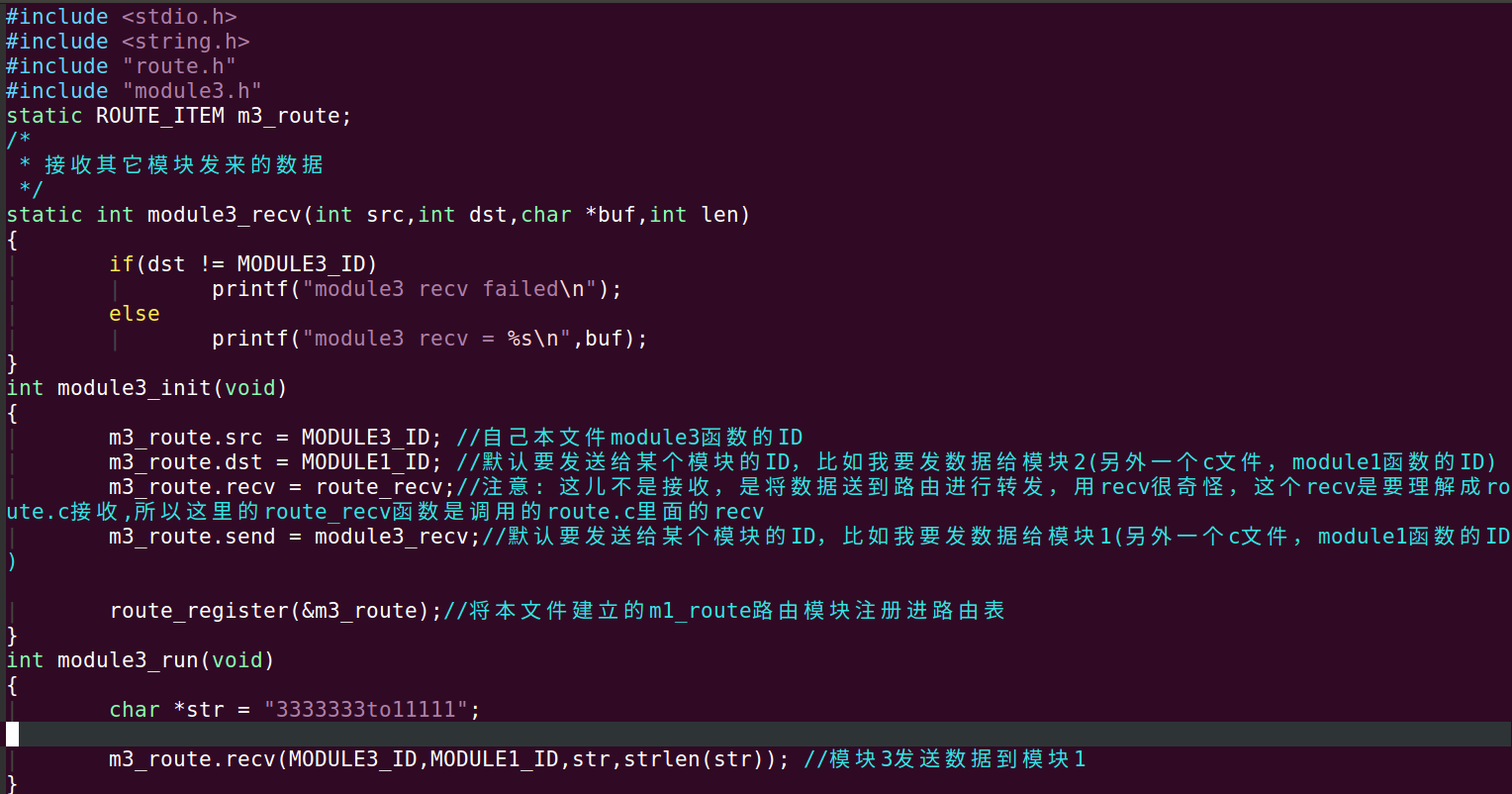


加入发送数据函数



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

增加一个模块3的C文件

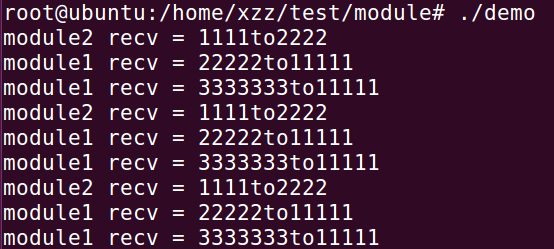


main函数执行module3\_run，相当于主程序只需要加入模块3的init和run就可以了

模块3数据传输路径

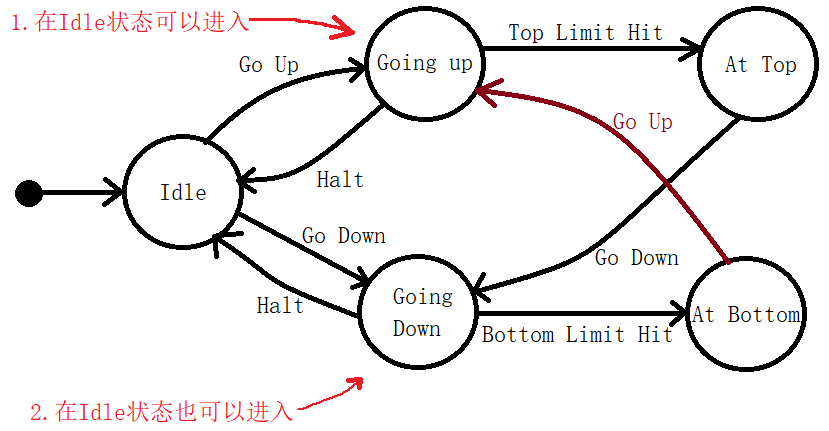
增加模块3数组节点

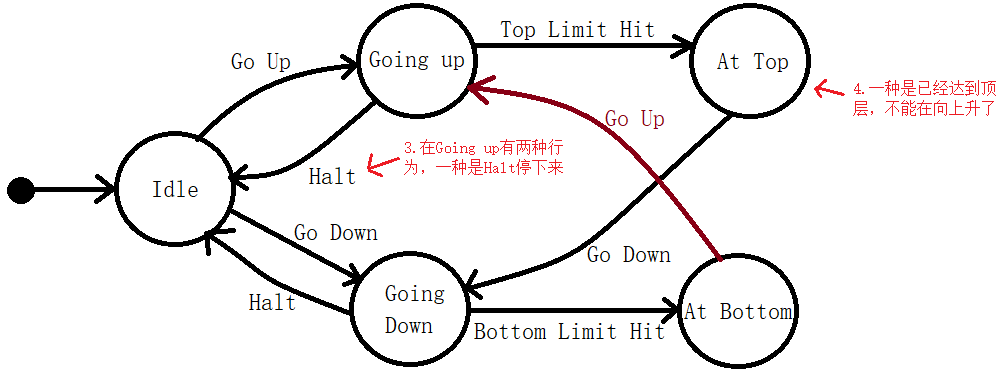
修改模块3接受函数名

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



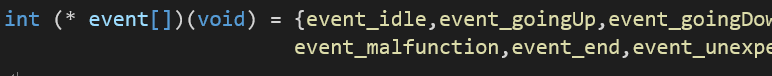


#include <stdio.h>

每个状态对应一个函数

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

每个状态函数执行完之后，返回的下一个状态

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

}

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)

{

不同状态执行不同的函数，在主循环轮询使用

    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},

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srare\_func

        rc = state\_func();

        if(\_end == cur\_state) //如果当前状态是\_end，退出死循环

cur\_state默认为0

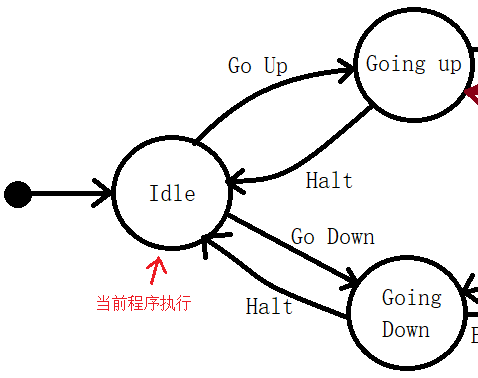
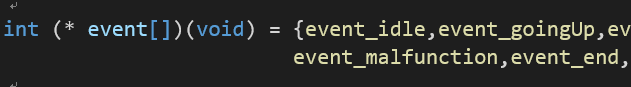
            break;

        cur\_state = lookup\_transitions[cur\_state][rc];

    }

    return 0;

}

cur\_state默认为0 进入idle状态，在event[]数组里面找event\_idle状态执行

event\_idle在数组0下标

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

如果没有操作，返回6

else //什么都不操作

return quit ; // 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srare\_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;

[0][7]在状态表就是 \_end

        cur\_state = lookup\_transitions[cur\_state][rc]; //[0][6] 也就是[\_idle][6]

    }

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srare\_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},

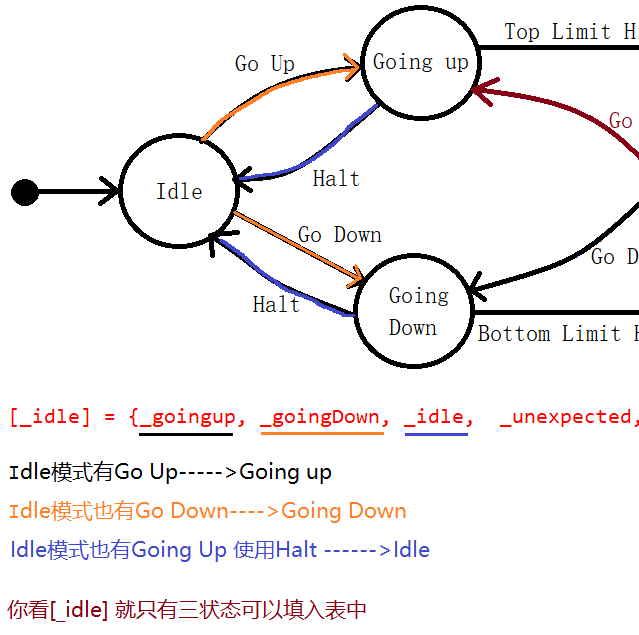
    [\_goingDown] = {\_unexpected, \_goingDown,  \_idle,     \_AtTop,       \_AtBottom,   \_malfunction, \_end},

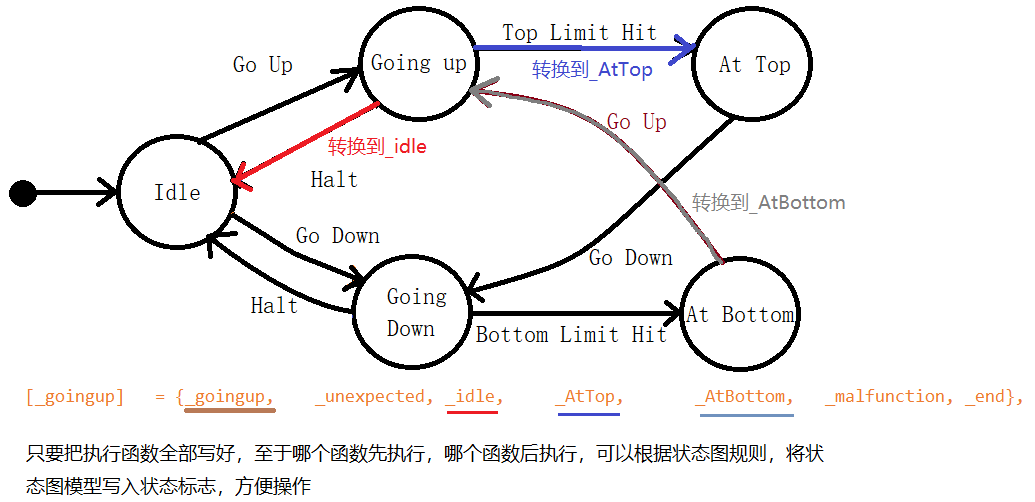
    [\_AtTop]     = {\_unexpected, \_goingDown,  \_AtTop,    \_unexpected,  \_unexpected, \_malfunction, \_end},

    [\_AtBottom]  = {\_goingup ,   \_goingDown,  \_AtBottom, \_unexpected, \_unexpected, \_malfunction , \_end},

    [\_malfunction] = {\_end , \_end , \_end , \_end , \_end, \_end , \_end},

    [\_unexpected] = {\_end , \_end , \_end , \_end , \_end, \_end , \_end},



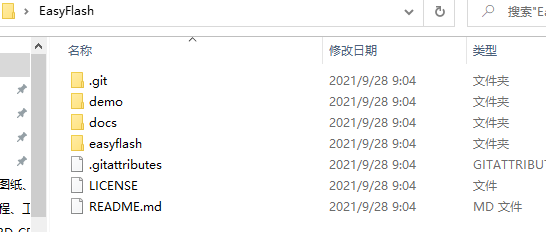


这就是FSM有限状态机，为了编程方便而设计

**EasyFlash存储库使用**

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

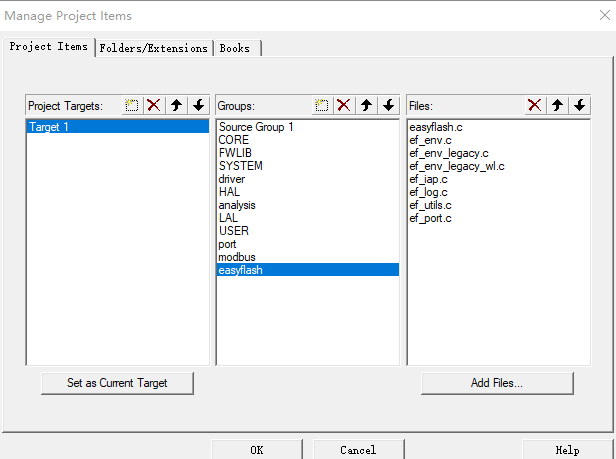


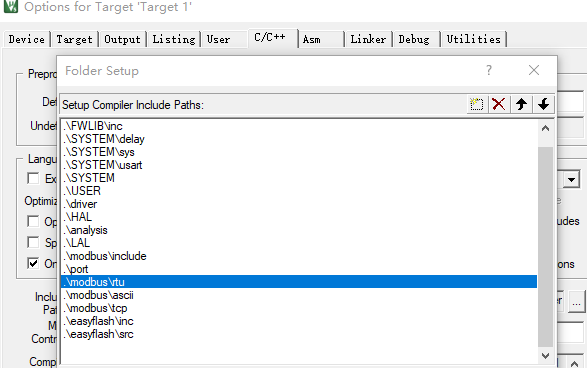


移植easyflash目录到STM32工程

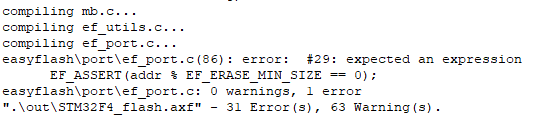
添加下面4个文件到工程

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



编译出现很多错误

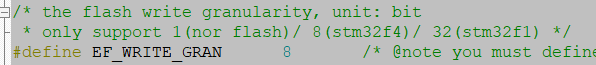
打开ef\_cfg.h文件，修改里面的配置

 添加版本号，我随便写的0

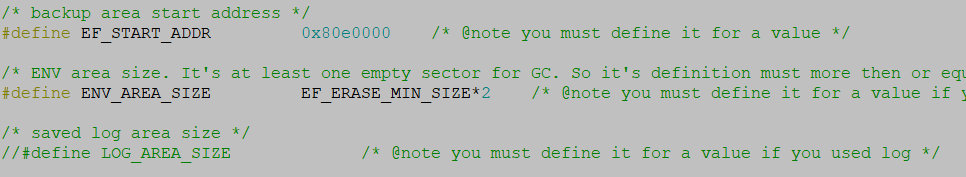
根据实际情况填写最小擦除粒度, 我是用的是F103rb所以我写的是1024, 如果你使用的是F103ze就应该填2048,根据实际情况进行填写

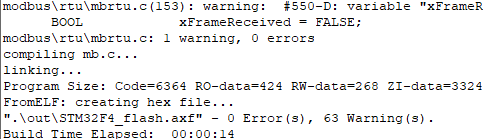


EF\_WRITE\_GRAN上面有注释,如果是stm32f4就要填8



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)





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………………….