

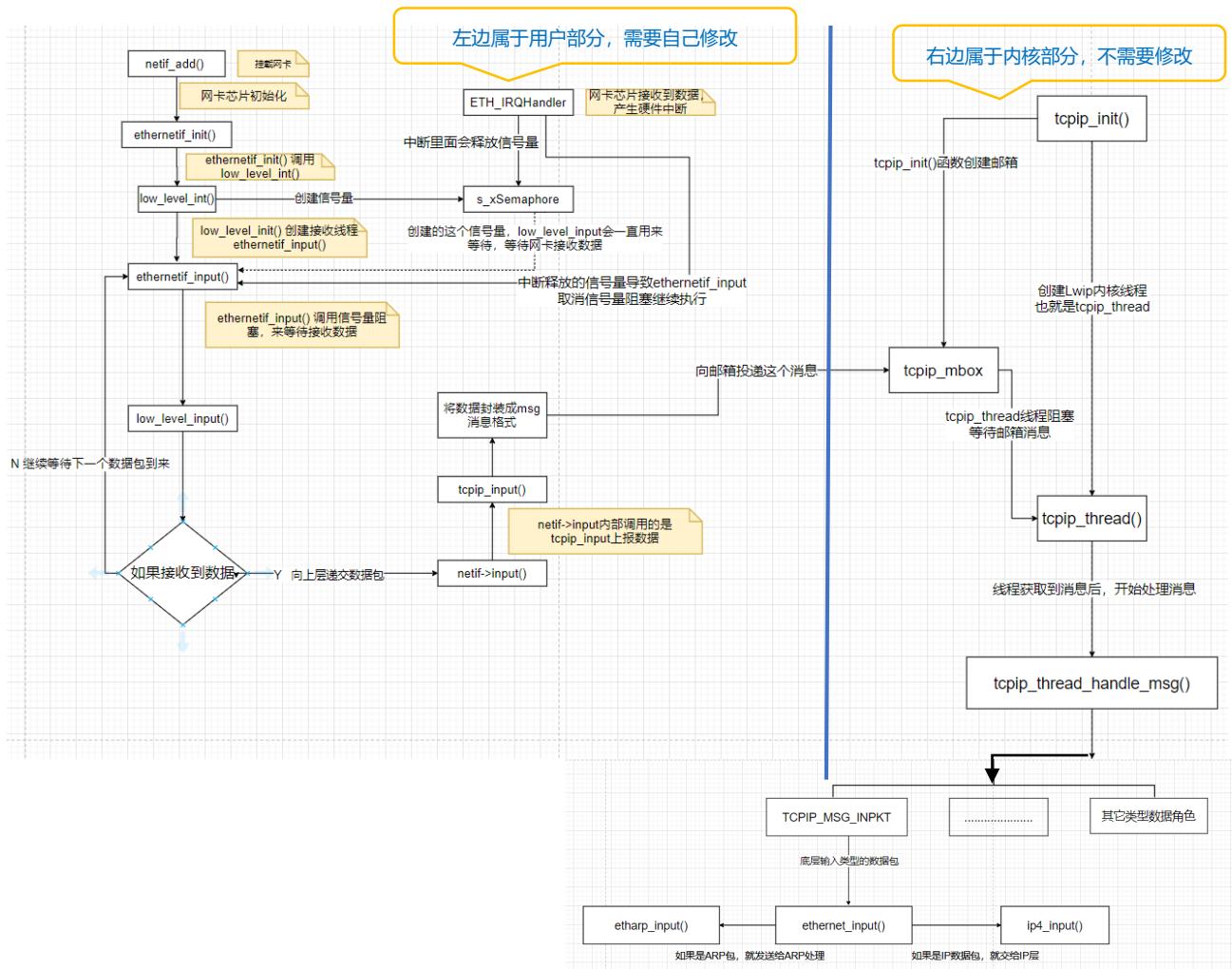
LWIP 使用指南

作者:向仔州

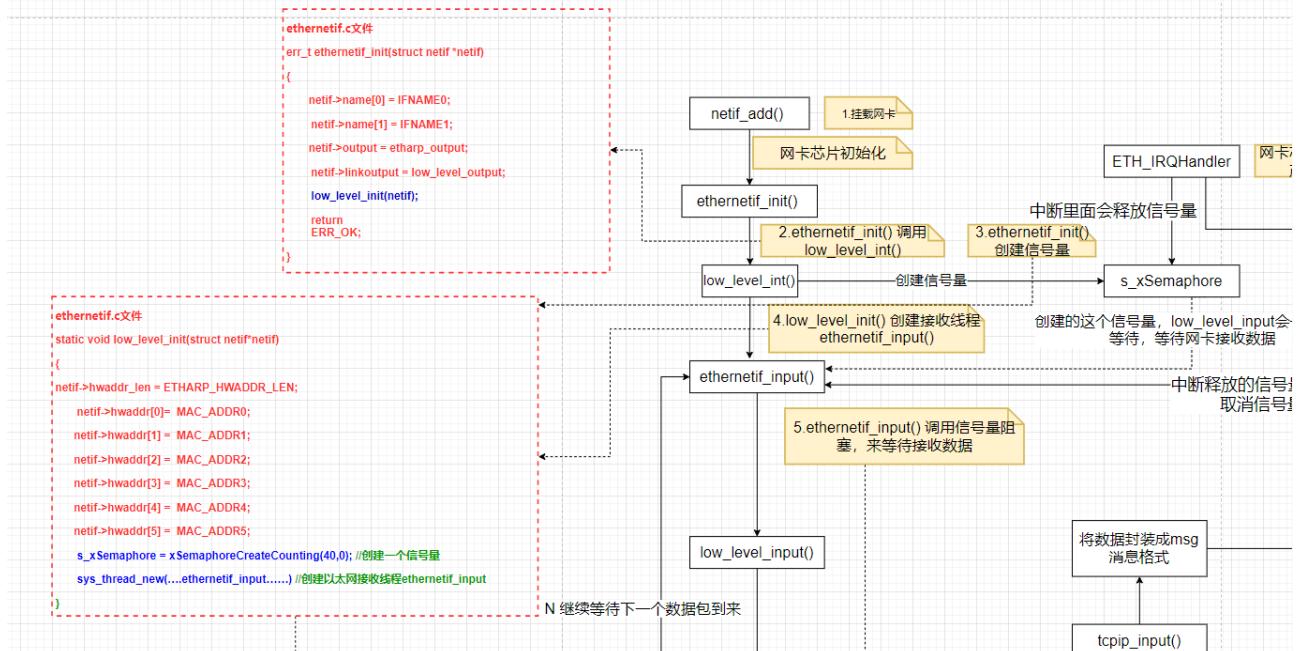
目录

LWIP 基本流程, 以 FreeRTOS 操作系统移植 LWIP 为例.....	2
ARP 协议.....	5
LWIP 内存管理.....	7
LWIP 无操作系统初始化流程.....	7
LWIP 无操作系统 UDP 通信.....	10
LWIP 无操作系统 TCP 通信	16
LWIP 带 FreeRTOS RAW API 网络通信.....	25
LwIP SOCKET 网络接口使用	35
RAW API MQTT 实现.....	47
最简单的 MQTT 连接服务器 CONNECT 程序.....	50

LWIP 基本流程，以 FreeRTOS 操作系统移植 LWIP 为例



下面对流程中的每个模块，进行代码解读

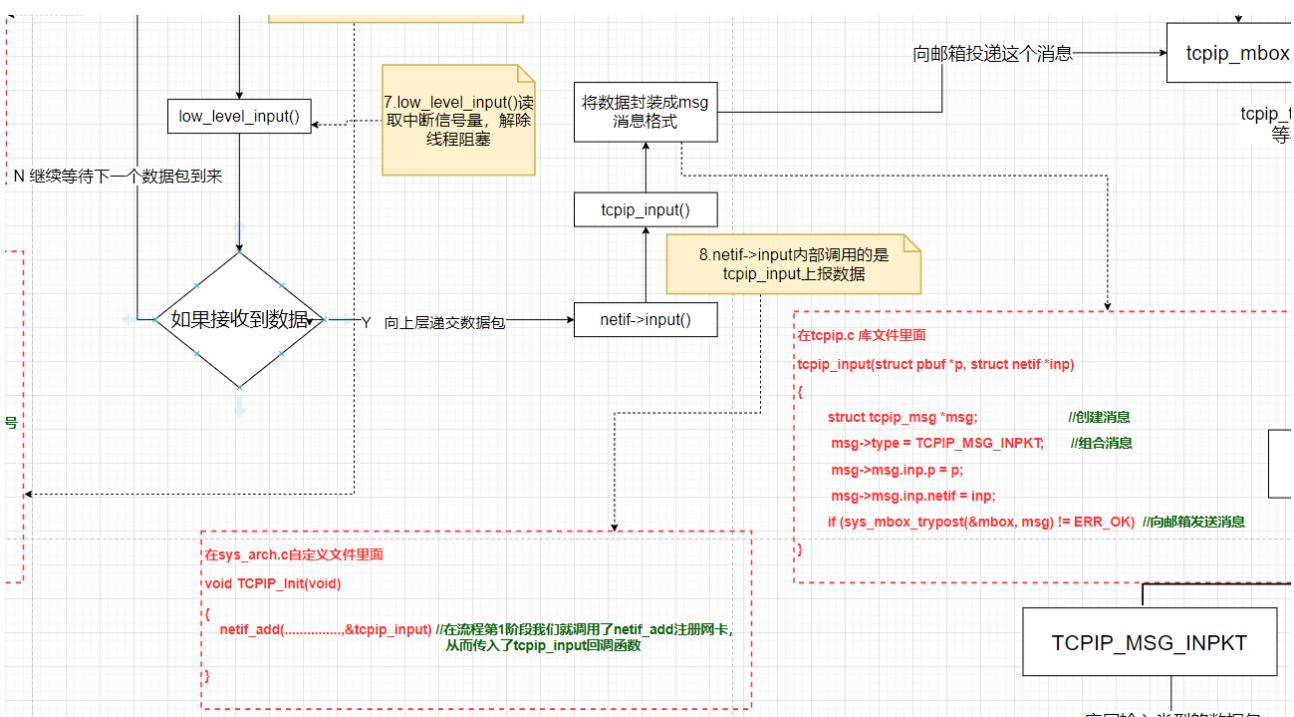
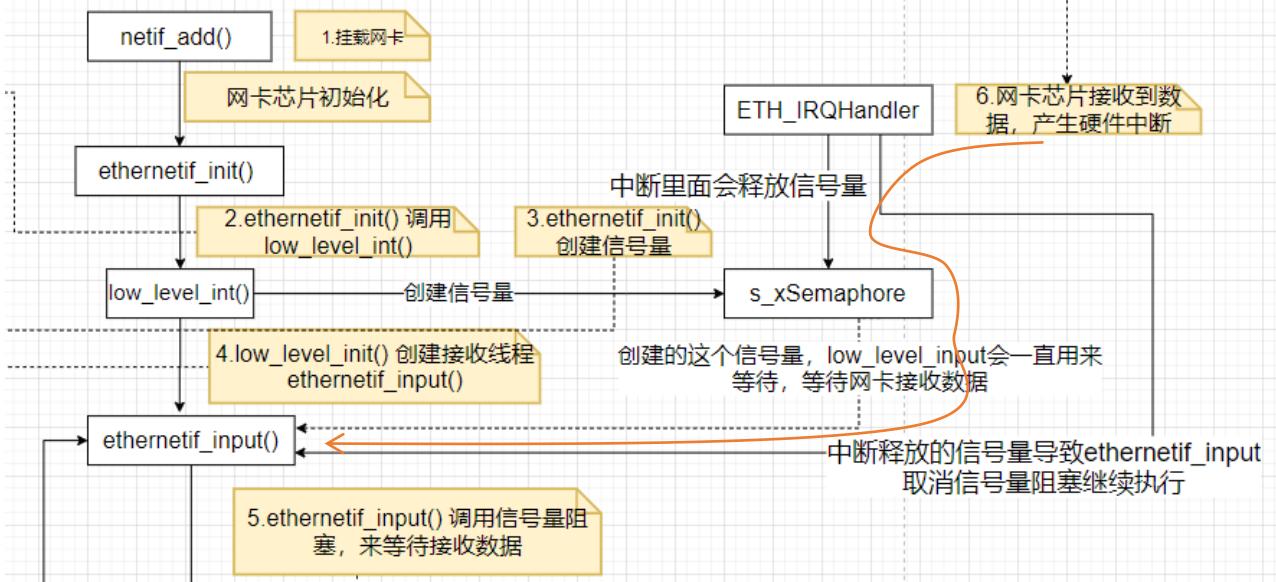


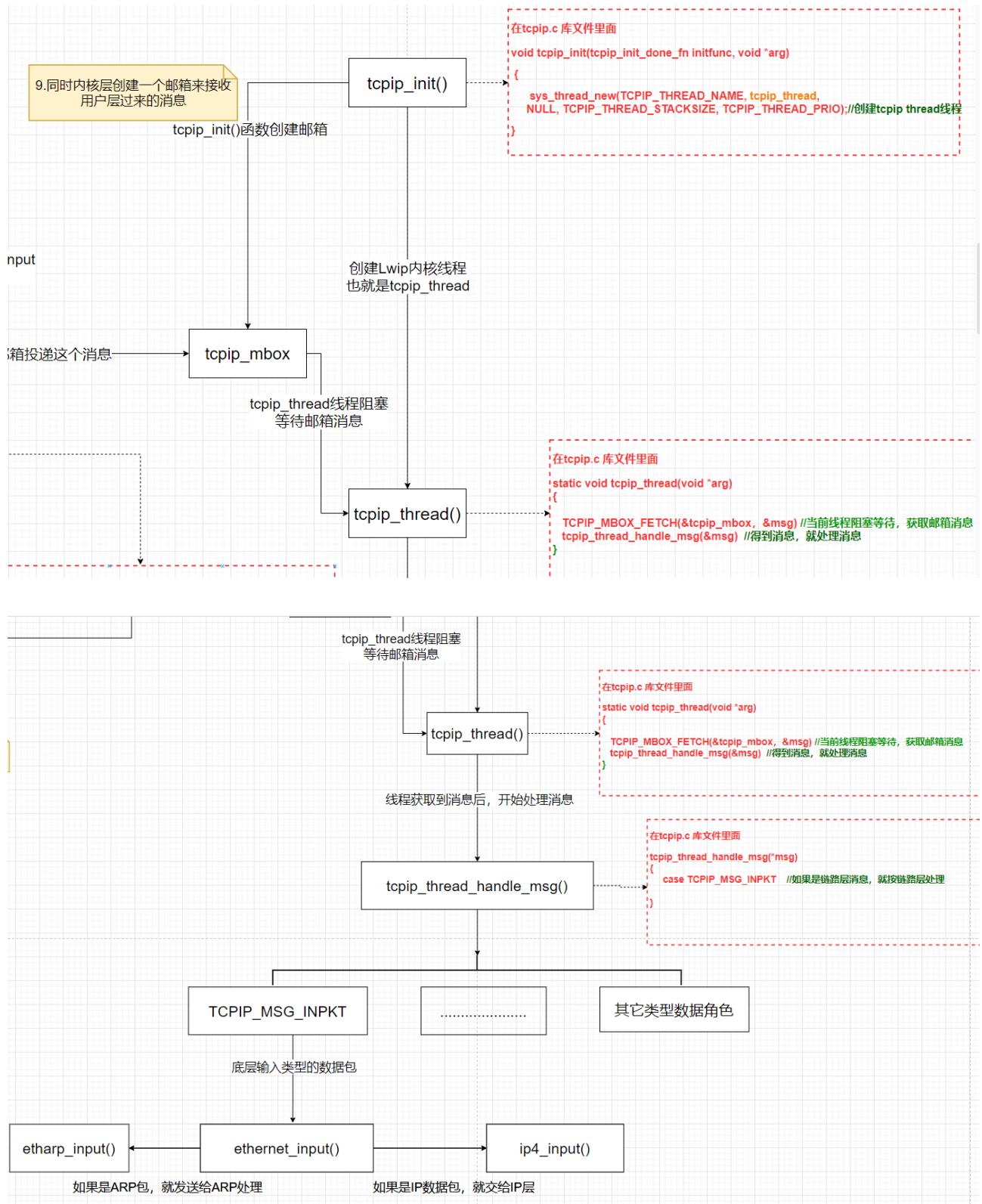
```

    在bsp_eth.C文件官方网卡驱动里面
    void ETH_IRQHandler(void)
    {
        taskENTER_CRITICAL_FROM_ISR(); //进入临界区
        xSemaphoreGiveFromISR ( s_xSemaphore,..... ) //释放信号量

        taskEXIT_CRITICAL_FROM_ISR(); //退出临界区
    }

```





这就是整个 lwip 数据转发流程

ARP 协议

MAC 地址，就是硬件物理地址。MAC 地址被固化在网卡的 ROM 中。

MAC 地址长度为 6 字节（48 比特），其前 3 个字节表示组织唯一标志符由 IEEE 的注册管理机构给不同厂家分配的代码，以区分不同的厂家。

后 3 个字节由厂家自行分配，称为扩展标识符。同一个厂家生产的网卡中 MAC 地址后 24 位是不同的

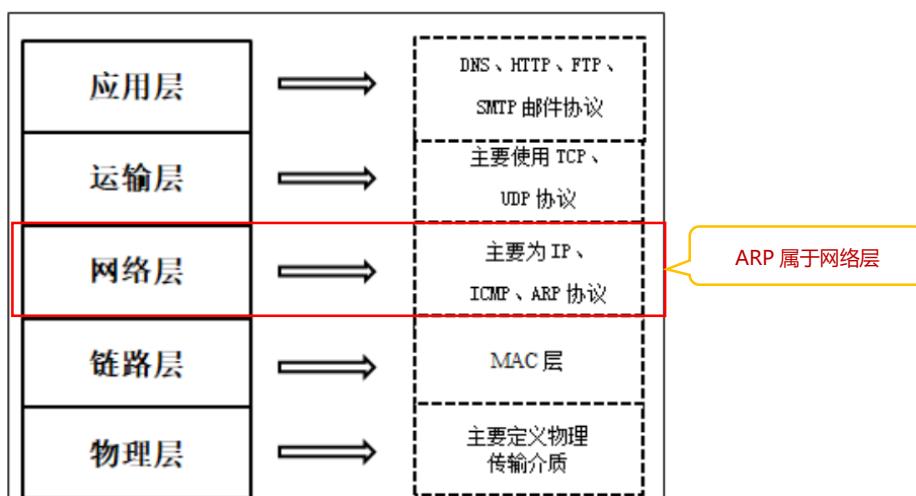
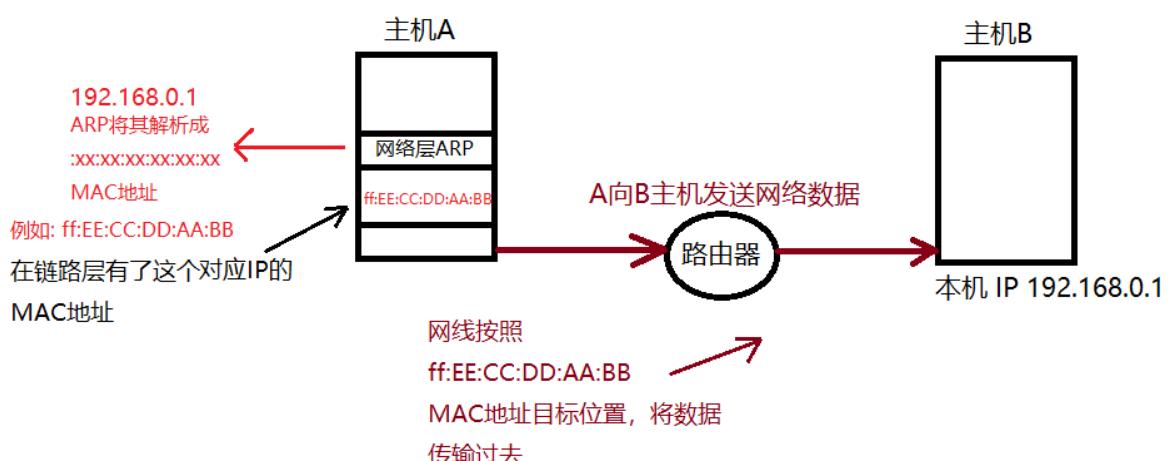


图 1-1 TCP/IP 协议的分层



这才是网线，路由器网络认可的地址，MAC 地址。之所以做成 IP 地址，只是为了让我们阅读方便。
以太网帧结构

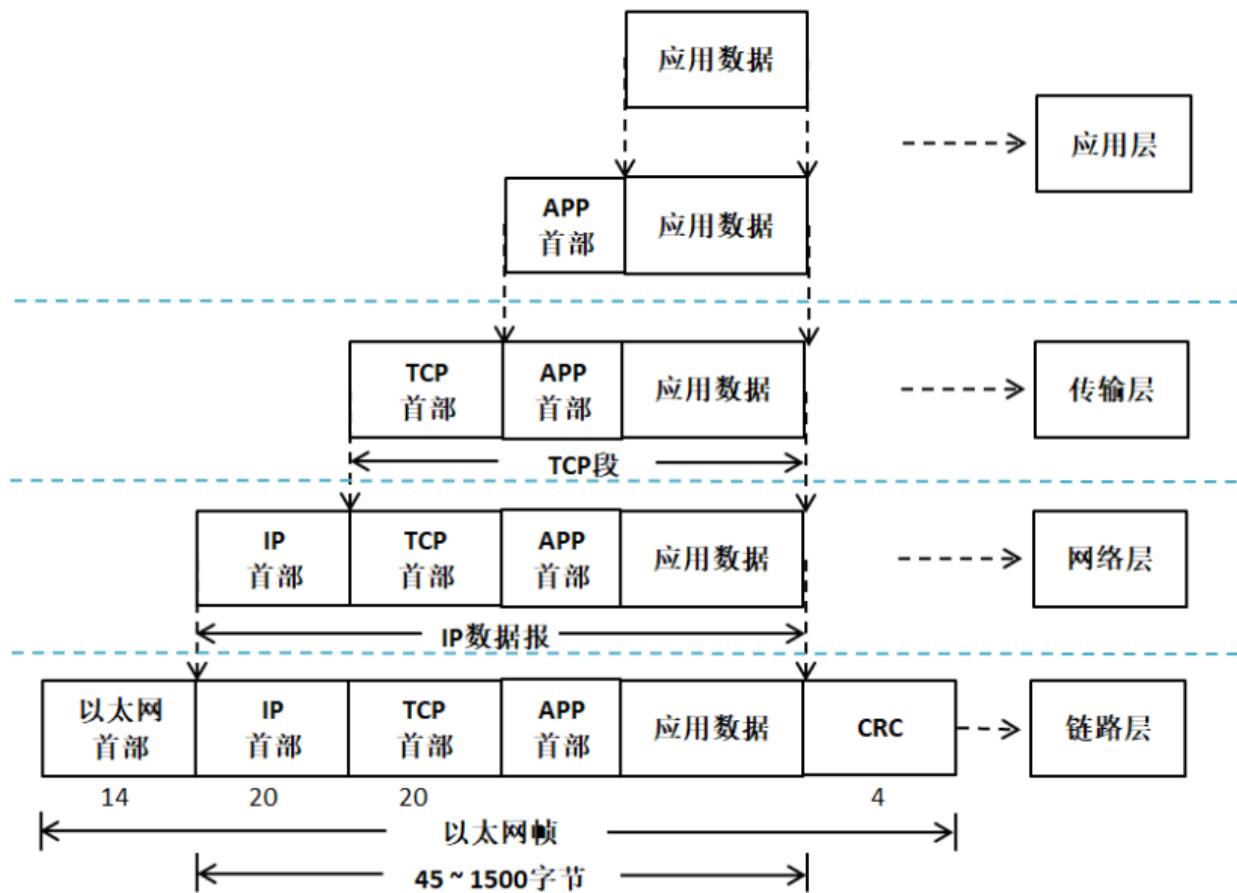


图 1-2 TCP/IP 协议栈各层的报文封装与拆封

前同步码	帧开始符	目标MAC地址	源MAC地址	类型	数据	CRC
7字节	1字节	6字节	6字节	2字节	46~1500字节	4字节

ARP 协议就是将 IP 协议转换成 MAC 地址的代码。

LWIP 内存管理

Lwip 使用的是固定内存块大小分配策略，相当于每次分配的内存大小有个最小单位。



内核在初始化内存池的时候，是根据用户配置的宏定义进行初始化的，比如，用户定义了 LWIP_UDP 这个宏定义，在编译的时候，编译器就会将与 UDP 协议控制块相关的数据构编译编译进去，这样子就将 LWIP_MEMPOOL(UDP_PCB, MEMP_NUM_UDP_PCB, sizeof(struct udp_pcb).” UDP_PCB”) 包含进去，在初始化的时候， UDP 协议控制块需要的 POOL 资源就会被初始化，其数量由 MEMP_NUM_UDP_PCB 宏定义决定。所以主程序执行 memp_get_memorysize() 函数，就得到 UDP 的大小，然后用类似 malloc 的函数对 memp_get_memorysize() 算出的大小进行分配内存。

LWIP 无操作系统初始化流程

- 1.确保 LWIP 底层移植已经成功。
- 2.确保使用原子的 LWIP 例程获取 DHCP 成功。

结构体: struct netif ,是通过结构体 netif 来描述一个硬件网络接口的，在单网卡中，这个结构体只有一个。多网卡中 netif 可以有多个。将多个 netif 结构链接成一个数据链，就是多网卡。

1.在代码初始化的时候，先定义网络结构体，如下 4 个，网卡 netif，IP 地址 ip_addr，子网掩码和网关

```
struct netif *Netif_Init_Flag;      //调用 netif_add()函数时的返回值,用于判断网络初始化是否成功
struct ip_addr ipaddr;             //ip 地址
struct ip_addr netmask;            //子网掩码
struct ip_addr gw;                 //默认网关
```

2.给以太网卡 DMA 分配内存，STM32 独有的以太网 phy 内存分配，其它芯片不一样

```
DMARxDscrTab=mymalloc(SRAMIN,ETH_RXBUFN* sizeof(ETH_DMADESCTypeDef));//申请内存,以太网 DMA 接收
描述符
DMATxDscrTab=mymalloc(SRAMIN,ETH_TXBUFN* sizeof(ETH_DMADESCTypeDef));//申请内存,以太网 DMA 发送
描述符
```

```
Rx_Buff=mymalloc(SRAMIN,ETH_RX_BUF_SIZE*ETH_RXBUFN); //申请内存,以太网底层驱动接收 buffers 指针
```

```

Tx_Buff=mymalloc(SRAMIN,ETH_TX_BUF_SIZE*ETH_TXBUFN); //申请内存,以太网底层驱动发送 buffers 指针
if(!DMARxDscrTab||!DMATxDscrTab||!Rx_Buff||!Tx_Buff)
{
    ETH_Mem_Free();
    return 1; //申请失败
}

```

3.获得 POOL 需要分配的内存大小, 这里需要 LWIP 内存管理的知识

```

u32 mempsize;
mempsize = memp_get_memorysize(); //得到 memp_memory 数组大小, LWIP 内存管理章节讲过
memp_memory = mymalloc(SRAMIN,mempsize); //为 memp_memory 申请内存

```

在 memp.c 文件中

u8_t *memp_memory;

//将其自定义为全局指针, 名字千万不能变

在 MEMP_SEPARATE_POOLS 宏中使用

ramheapsize=LWIP_MEM_ALIGN_SIZE(MEM_SIZE)+2*LWIP_MEM_ALIGN_SIZE(4*3)+MEM_ALIGNMENT;// 得到 ram heap 大小

```
ram_heap=mymalloc(SRAMIN,ramheapsize); //为 ram_heap 申请内存
```

```
if(!memp_memory||!ram_heap)//有申请失败的
```

```
{
```

lwip_comm_mem_free();//就是类似 free 函数

return 1;

```
}
```

4.硬件和协议栈初始化

```
LAN8720_Init(); //初始化 LAN8720 GPIO 和外设
```

```
lwip_init(); //初始化 LWIP 内核
```

5.因为使用的动态 IP, 先将地址, 掩码, 网关参数置 0

```
ipaddr.addr = 0;
```

```
netmask.addr = 0;
```

```
gw.addr = 0;
```

6.定义一个全局网络接口, 使用 netif_add 初始化

```
struct netif lwip_netif; //定义一个全局的网络接口
```

```
Netif_Init_Flag = netif_add(&lwip_netif,&ipaddr,&netmask,&gw,NULL,&ethernetif_init,&ethernet_input);
```

//向网卡列表中添加一个网口

```
if(Netif_Init_Flag==NULL) return 3; //网卡添加失败
```

else//网口添加成功后,设置 netif 为默认值,并且打开 netif 网口

```
{
```

netif_set_default(&lwip_netif); //设置 netif 为默认网口

netif_set_up(&lwip_netif); //打开 netif 网口

```
}
```

```
dhcp_start(&lwip_netif); //开启 DHCP 服务
```

err_t dhcp_start(struct netif *netif)
该接口用于设备启动 dhcp 模块，**主要是客户端的功能**。该模块生成 dhcp
描述符结构，并将 dhcp 的端口绑定到 udp 协议中，以及将本 dhcp 模块
跟远端服务器端口进行绑定。最后启动 dhcp 申请。
static void dhcp_recv(...)
该接口为一个注册接口，用于 dhcp 报文接收

```
while(1) //循环处理网络事件
```

```
{  
//每 250ms 调用一次 tcp_tmr()函数  
if (lwip_localtime - TCPTimer >= TCP_TMR_INTERVAL)  
{  
    TCPTimer = lwip_localtime;  
    tcp_tmr();  
}  
}  
//ARP 每 5s 周期性调用一次
```

```
if ((lwip_localtime - ARPTimer) >= ARP_TMR_INTERVAL)  
{  
    ARPTimer = lwip_localtime;  
    etharp_tmr();  
}
```

```
//每 500ms 调用一次 dhcp_fine_tmr()
```

```
if (lwip_localtime - DHCPfineTimer >= DHCP_FINE_TIMER_MSECS)  
{  
    DHCPfineTimer = lwip_localtime;  
    dhcp_fine_tmr();  
    if ((lwipdev.dhcpstatus != 2)&&(lwipdev.dhcpstatus != 0xFF)) //如果未获取到 DHCP，重复执行  
    {  
        lwip_dhcp_process_handle(); //DHCP 处理自定义函数，需要自己定义  
    }  
}
```

dhcp_fine_tmr() //用于请求应答超时处理
dhcp_coarse_tmr() //用于地址租用情况的到期处理

```
//每 60s 执行一次 DHCP 粗糙处理
```

```
if (lwip_localtime - DHCPcoarseTimer >= DHCP_COARSE_TIMER_MSECS)  
{  
    DHCPcoarseTimer = lwip_localtime;  
    dhcp_coarse_tmr();  
}
```

```
}
```

这就是整个网卡初始化启动流程。

LWIP 无操作系统 UDP 通信

RAW_UDP 发送数据实验

```
#include "lwip/pbuf.h"  
#include "lwip/udp.h" //udp 发送数据需要包含 pbuf 和 udp 头文件
```

```
struct udp_pcb * udp_new(void) //建立一个 UDP 控制块, UDP 可以建立很多个控制块  
IP4_ADDR(ipaddr, a,b,c,d) //将点分 10 进制 IP 地址转为 4 字节变量, 然后赋值给 ipaddr  
ipaddr: 使用 struct ip_addr 创建的变量, 填入 ipaddr。 struct ip_addr 创建的变量可以保存 ip 地址, 掩码, 网关。  
a,b,c,d 使用如下:
```

IP 地址如: 192.168.0.101

a: 填入 192, b: 填入 168, c: 填入 0, d: 填入 101

```
err_t udp_connect(struct udp_pcb *pcb, ip_addr_t *ipaddr, u16_t port) //连接远程主机, 由于 UDP  
通信是面向无连接的, 所以这不会产生任何的网络流量, 相当于没有真实的取连接服务器, 只是指定服务  
器 IP 和端口, 到时候发送的时候直接向服务器发送就是。
```

*pcb: 填入 udp_new 创建的 udp 控制块, 证明该连接是新控制块连接的

*ipaddr: 传入 ipaddr, 也就是 IP4_ADDR 转换的远端服务器 IP 地址变量。

port: 传入远程服务器端口号, UDP 一般默认 8089

```
err_t udp_bind(struct udp_pcb *pcb, ip_addr_t *ipaddr, u16_t port) //绑定本机 IP 地址和端口
```

*pcb: 填入 udp_new 创建的 udp 控制块, 证明该连接是新控制块连接的

*ipaddr: 填写本机 IP 地址, 一般是填 IP_ADDR_ANY。也可以填广播 IP_ADDR_ANY

```
void udp_recv(struct udp_pcb *pcb, udp_recv_fn recv, void *recv_arg) //接收服务器发来的数据, 回  
调函数注册, 无操作系统的 RAW_UDP 只能用事件回调函数来接收服务器发来的数据, 而不能用阻塞方  
式, 这就是 RAW 模式做网络通信的麻烦之处。
```

*pcb: 填入 udp_new 创建的 udp 控制块, 证明该连接是新控制块连接的

recv: 注册事件回调函数, 接收到服务器数据, 就执行该函数。回调函数的函数形参如下;

```
void udp_RecvData(void *arg, struct udp_pcb *upcb, struct pbuf *p, struct ip_addr *addr, u16_t port)
```

recv_arg: 外部传递给回调函数的参数, 执行 udp_recv 的时候传参, 默认填 NULL

```
err_t udp_send(struct udp_pcb *pcb, struct pbuf *p) //发送数据到服务器, 但是不能直接写入发送的  
参数给*p, 而是要使用 pbuf_alloc 分配内存, 然后赋值什么的, 才能把数据给*p, 麻烦得很下面有介绍。
```

*pcb: 填入 udp_new 创建的 udp 控制块, 证明该连接是新控制块连接的

*p: 传入 pbuf_alloc 分配发送数据内存后, 返回的指针。这就是麻烦的地方。

`struct pbuf * pbuf_alloc(pbuf_layer layer, u16_t length, pbuf_type type)` //在协议层分配内存给 pbuf
layer: 指定 pbuf 分配到哪一层, 这就是关键, 这个 layer 并不是每一层都使用同一个宏

PBUF_TRANSPORT: 如果为传输层(如 TCP, UDP)申请 pbuf, 那么就选 PBUF_TRANSPORT

PBUF_IP: 如果是 IP 层申请 pbuf, 那么选择 PBUF_IP

PBUF_LINK: 如果是链路层申请 pbuf, 那么选择 PBUF_LINK

length: 用 `strlen` 计算发送数据包的长度给 length, 如果是字节流就用 `sizeof` 计算

type: PBUF_RAM 一次性分配 size 大小的连续内存

RBUF_ROM 只需要分配小小的管理 pbuf 的控制管理内存

PBUF_ROOL 分配一个链表, 链表上每个元素所管理的内存最大不超过 PBUF_POOL_BUFSIZE, 它更像 linux 中的 `kmem_alloc` 内存高速缓存机制, 所以它也更适合在网卡驱动 irq 中断中为刚刚到来的网络数据包申请存储空间
一般选择 PBUF_ROOL

`err_t pbuf_take(struct pbuf *buf, const void *dataptr, u16_t len)` //用于向 pbuf_alloc 分配的 pbuf 拷贝数据, `udp_send` 就得用 `pbuf_take` 拷贝之后的数据才能转发。 (如果用 pbuf 的 payload, 线程就会卡死在 `udp_send`, 所以放弃 payload 方案, 用 `pbuf_take` 代替)

*buf: 传入 `pbuf_alloc` 分配的 pbuf

*dataptr: 放入要发送的数据包地址, 如数组地址, 字符串首地址

len: 计算发送数据包的长度, `strlen` 或者是 `sizeof` 计算

下面代码测试:

```
1 #include "stm32f4xx.h" //包含F407整个头文件
2 #include "stm32f4xx_it.h"
3 #include "delay.h"
4 #include <stdio.h>
5
6 #include "usart1.h"
7 #include "tim3.h"
8
9 #include "malloc.h" //这个malloc.h是自己实现的
10
11 #include "lwip/netif.h"
12 #include "lwip_comm.h"
13 #include "lwipopts.h"
14 #include "lwip/pbuf.h" //UDP加入头文件
15 #include "lwip/udp.h" //UDP加入头文件
16
17 #include <string.h>
#define UDP_PORT 8089 //UDP默认端口号8089
/*接收UDP服务器数据的事件回调函数, 现在不用, 先写起*/
void udp_RecvData(void *arg, struct udp_pcb *upcb, struct pbuf *p, struct ip_addr *addr, u16_t port)
{
}

int main(void)
{
    const uint8_t udp_sendData[] = "stm32f407 udp data"; //这就是UDP要发送的数据
    struct udp_pcb *udppcb; //必须定义一个UDP控制块
    struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
    err_t err; //udp函数执行后返回状态

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为115200
    TIM3_Int_Init(999, 839);
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    printf("xxxxxxxx\r\n");
}
```

初始化 LWIP 协议，获取 DHCP 数据，如下：

```
while(lwip_comm_init()) //lwip初始化
{
    printf("lwip_comm_init failed...\r\n");
    delay_ms(1000);
}

printf("lwip_success_init...\r\n");

while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF))//等待DHCP获取成功/超时溢出
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数
}
```

创建 UDP 连接，如下：

```
udppcb = udp_new(); //创建udp
if(udppcb) //udppcb不等于NULL，表示创建成功
{
    IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量， //因为IP控制块里面通过32位存储IP地址
    err = udp_connect(udppcb, &RemoteAddr, UDP_PORT); //设置远端IP和端口
    if(err == ERR_OK) //udp执行链接成功，但不是真正的链接哈，UDP没有链接这种功能
    {
        err = udp_bind(udppcb, IP_ADDR_ANY, UDP_PORT); //设置本地IP和端口，其中IP_ADDR_ANY表明使用DHCP网卡地址， udp_recv(udppcb, , udp_RecvData, NULL); //注册控制块回调函数，端口收到数据，回调处理函数将被调用。 //在发送函数里会判断这个变量，选择网卡地址
        printf("udp connect success\r\n");
    }
}
```

UDP 发送数据前需要分配内存，然后用 pbuf_take 赋值，麻烦得很

```
struct pbuf *ptr;
ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char *)udp_sendData), PBUF_POOL);

if(ptr) //如果内存分配成功，发送数据
{
    pbuf_take(ptr, (char *)udp_sendData, strlen((char *)udp_sendData));
    udp_send(udppcb, ptr); //udp发送处理之后的数据
    pbuf_free(ptr); //释放内存
}
```

```
if(ptr) //如果内存分配成功，发送数据
{
    ptr->payload = (void *)udp_sendData; //如果用payload方式发送UDP数据，线程会卡在udp_send
    udp_send(udppcb, ptr); //udp发送处理之后的数据
    pbuf_free(ptr); //释放内存
}
```

所以 payload 是原子的坑，也许和 pbuf_alloc PBUF_POOL 有关，没深入研究

现在改为 pbuf_take，问题得到解决

UDP 循环发送

```
while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数，所以无操作系统需要一个死循环来不停的维护LWIP内核
    //*****循环发送UDP，每次发送都要分配内存->发送数据->释放内存，必须这样，不然会出现bug*****
    ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char *)udp_sendData), PBUF_POOL);
    pbuf_take(ptr, (char *)udp_sendData, strlen((char *)udp_sendData));
    udp_send(udppcb, ptr); //udp发送处理之后的数据
    pbuf_free(ptr); //释放内存
    delay_ms(10);
}
```

UDP 服务器 python 在 ubuntu 虚拟机上实现

```
import socket

udpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
udpserver.bind(("192.168.0.112",8089))

while True:
    data = udpserver.recv(1024)
    print(data.decode("utf-8"))
```

```
stm32f407 udp data
xxxxxx
lwip success init.
正在查找DHCP服务器,请稍等......
网卡的MAC地址为.....2.0.0.52.0.62
通过DHCP获取到IP地址.....192.168.0.101
通过DHCP获取到子网掩码.....255.255.255.0
通过DHCP获取到的默认网关.....192.168.0.1
udp connect success
```

服务器接收 F407 的 UDP 数据成功。

UDP 接收数据实验

主要就是修改接收数据事件回调函数

```
#define UDP_RX_LEN 2000 //定义UDP接收数据最大长度
#define UDP_PORT 8089 //UDP默认端口号8089
uint8_t udp_recvbuf[UDP_RX_LEN]; //定义UDP接收的数据缓存区 2000字节

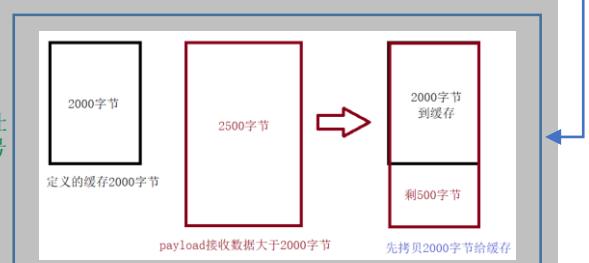
/*接收UDP服务器数据的事件回调函数, 现在实现接收数据的事件函数*/
void udp_RecvData(void *arg, struct udp_pcb *upcb, struct pbuf *p, struct ip_addr *addr, u16_t port)
{
    u32 data_len = 0;
    struct pbuf *q;

    if(p!=NULL) //接收到不为空的数据时, 也就是UDP服务器发送数据过来了
    {
        memset(udp_recvbuf, 0, UDP_RX_LEN); //数据接收缓冲区清零, 将以前或者上一次的数据清0
        for(q=p;q!=NULL;q=q->next) //遍历完整个pbuf链表
        {
            //判断要拷贝到q->payload中的数据长度是否大于UDP_RX_LEN的剩余空间, 如果大于
            //的话就只拷贝UDP_RX_LEN(2000)中剩余长度的数据, 否则的话就拷贝所有的数据
            if(q->len > (UDP_RX_LEN-data_len))
            {
                memcpy(udp_recvbuf+data_len,q->payload,(UDP_RX_LEN-data_len)); //拷贝数据UDP_RX_LEN(2000)字节
            }
            else
                memcpy(udp_recvbuf+data_len,q->payload,q->len); //如果接收数据不大于2000,
                //就拷贝< 2000字节以q->len为准
            data_len += q->len; //将本次for循环拷贝的数据长度给data_len,
            //下次循环地址接着累加data_len之后的地址开始拷贝
            if(data_len > UDP_RX_LEN) //超出UDP客户端接收数组的大小, 跳出
                break;
        }

        printf("Recv = %s\r\n", udp_recvbuf);

        upcb->remote_ip= *addr;          //记录远程主机的IP地址
        upcb->remote_port= port;         //记录远程主机的端口号

        uint8_t IADDR4,IADDR3,IADDR2,IADDR1;
        IADDR1 = upcb->remote_ip.addr&0xff; //得到192.
        IADDR2 = (upcb->remote_ip.addr>>8)&0xff; //得到168.
        IADDR3 = (upcb->remote_ip.addr>>16)&0xff; //得到0.
        IADDR4 = (upcb->remote_ip.addr>>24)&0xff; //得到112
        printf("Recv IP %d . %d . %d . %d port = %d\r\n", IADDR1,IADDR2,IADDR3,IADDR4,upcb->remote_port);
        pbuf_free(p); //每次接受完一定要释放内存, 不然会出现pbuf_take: invalid buf" failed
    }
}
```



主函数实现方式和 UDP 发送数据实验一样

```
int main(void)
{
    const uint8_t udp_sendData[] = "stm32f407 udp data"; //这就是UDP要发送的数据
    struct udp_pcb *udppcb; //必须定义一个UDP控制块
    struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
    err_t err; //udp函数执行后返回状态

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为115200
    TIM3_Int_Init(999, 839);
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    printf("xxxxxxxx\r\n");

    while(lwip_comm_init()) //lwip初始化
    {
        printf("lwip_comm_init failed...\r\n");
        delay_ms(1000);
    }

    printf("lwip_success_init...\r\n");

    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待DHCP获取成功/超时溢出
    {
        lwip_periodic_handle(); //LWIP内核需要定时处理的函数
    }

    udppcb = udp_new(); //创建udp
    if(udppcb) //udppcb不等于NULL, 表示创建成功
    {
        IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
        // //因为IP控制块里面通过32位存储IP地址)
        err = udp_connect(udppcb, &RemoteAddr, UDP_PORT); //设置远端IP和端口
        if(err == ERR_OK) //udp执行链接成功, 但不是真正的链接哈, UDP没有链接这种功能
        {
            err = udp_bind(udppcb, IP_ADDR_ANY, UDP_PORT); //设置本地IP和端口, 其中IP_ADDR_ANY表明使用DHCP网卡地址
            udp_recv(udppcb, udp_RecvData, NULL); //注册控制块回调函数, 端口收到数据, 回调处理函数将被调用。
            // //在发送函数里会判断这个变量, 选择网卡地址
            printf("udp connect success\r\n");
        }
    }
}
```

主要是要注意注册 UDP 接收函数这里, 一定要注册接收函数

```
while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数, 所以无操作系统需要一个死循环来不停的维护LWI
    //*****循环发送UDP, 每次发送都要分配内存->发送数据->释放内存, 必须这样, 不然会出现bug*****
    ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char *)udp_sendData), PBUF_POOL);
    pbuf_take(ptr, (char *)udp_sendData, strlen((char *)udp_sendData));
    udp_send(udppcb, ptr); //udp发送处理之后的数据
    pbuf_free(ptr); //释放内存
    delay_ms(100);
}
```

lwip_periodic_handle() //LWIP 内核处理函数一定要频繁调用, 不管是 UDP 发送数据, 还是 UDP 接收数据, 都要执行该函数来处理。如果程序进入了一个很长的 for 循环, 也必须在很长的 for 循环里面频繁调用 lwip_periodic_handle()

Python 实现 UDP 接收和回发功能

```
import socket

udpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
udpserver.bind(("192.168.0.112",8089))

while True:
    data,addr = udpserver.recvfrom(1024)
    print(addr)
    print(data.decode("utf-8"))

    udpserver.sendto("callback Data to B".encode('utf-8'),addr)
```

```
('192.168.0.101', 8089)
stm32f407 udp data
('192.168.0.101', 8089)
```

UDP 接收数据成功

```
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
Recv = callback Data to B
Recv IP 192 . 168 . 0 . 112 port = 8089
```

f407 以太网收到服务器 UDP 数据

LWIP 无操作系统 TCP 通信

RAW_TCP 发送数据实验

```
struct tcp_pcb * tcp_new(void) //创建 TCP 控制块
```

IP4_ADDR(ipaddr, a,b,c,d) //将点分 10 进制 IP 地址转为 4 字节变量，然后赋值给 ipaddr，和 UDP 一样

```
err_t tcp_connect(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port, tcp_connected_fn connected)
```

*pcb: 传入 TCP 控制块地址

*ipaddr: 传入 ipaddr，也就是 IP4_ADDR 转换的远端服务器 IP 地址变量

port: 传入 TCP 服务器端口号

connected: 写入 TCP 连接成功后执行的回调函数

```
err_t tcp_write(struct tcp_pcb *pcb, const void *arg, u16_t len, u8_t apiflags) //发送 TCP 数据
```

注意: 执行 tcp_write 之后，并不会立即发送，实际上这个函数仅是将数据放入发送队列中，等待 lwip 认为发送缓冲队列数据够多了，再自动发送。所以数据发送次数给人感觉很慢。但是大批量传输效率很高。

*pcb: 传入 TCP 控制块地址

*arg: 传入发送的 TCP 数据地址，如数组地址或者字符串地址

len: 要发送的数据长度，strlen 或 sizeof 计算

apiflags: 发送数据方式选择：

0x01 数据将复制到堆栈中发送(一般都选这个)

0x02 对于 TCP 连接，在数据发送的最后一段加入 PSH

```
err_t tcp_output(struct tcp_pcb *pcb) //解决 tcp_write 无法立即发送数据问题
```

*pcb: 传入 TCP 控制块地址

执行 tcp_write 之后，马上执行 tcp_output，数据就能马上发出去

```
int main(void)
{
    const uint8_t tcp_sendData[] = "stm32f407 tcp data"; //TCP要发送的数据
    struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
    err_t err; //tcp函数执行后返回状态

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为115200
    TIM3_Int_Init(999, 839);
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    printf("xxxxxxxx\r\n");

    while(lwip_comm_init()) //lwip初始化
    {
        printf("lwip_comm_init failed...\r\n");
        delay_ms(1000);
    }

    printf("lwip_success_init...\r\n");
}
```

初始化和 UDP 没什么区别

```

while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待DHCP获取成功/超时溢出
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数
}

tcppcb = tcp_new(); //创建TCP控制块
if(tcppcb) //tcppcb不等于NULL, 表示tcp创建成功
{
    IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
    tcp_connect(tcppcb, &RemoteAddr, TCP_PORT, tcp_client_connected); //TCP主动连接
    printf("tcp pcb create success\r\n");
}

```

获取 dhcp 之后创建 TCP

```
tcp_err(tcppcb, client_err); //注册异常处理回调函数, 如果网络中断了, 或者其它网络故障, 调用该事件
```

TCP 连接失败, 或者 TCP 连接断开这种事件我们也需要知道, 这样才好做 TCP 重连机制, 加入异常处理回调

我们要实现这些回调函数, 比如 TCP 连接成功回调函数

```
//LWIP TCP连接建立后调用回调函数
```

```

err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    printf("connect ok\r\n");
}

```

```

static void client_err(void *arg, err_t err)
{
    printf("connect error\r\n");
    tcp_close(tpcb); //如果发生连接TCP失败, 先释放tcp控制块

    //进行重连函数执行
}

```

```

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数, 所以无操作系统需要一个死循环来不停的维护LWIP
    tcp_write(tpcb, tcp_sendData, sizeof(tcp_sendData), 1); //实际上这个函数仅是将数据放入发送队列中,
    //但并不会立即发送, 等待lwip认为发生缓冲队列数据够多了, 再自动发送

    delay_ms(100);
}

```

100ms 循环发送一帧数据

```

import socket
from socket import *

tcpserver = socket(AF_INET, SOCK_STREAM)
tcpserver.bind(("192.168.0.112", 8888))
tcpserver.listen(5)

while True:
    newsocket, addr = tcpserver.accept()
    while True:
        data = newsocket.recv(1024)
        print("receive : %s : %s" % (str(addr), data))

newsocket.close()
tcpserver.close()

```

python TCP 服务器

```

receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00'
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00'
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00'
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00'
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00stm32f407 tcp data\x00'
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00'

```

测试发现 TCP 服务器接收数据并不是 100ms 接收一条，换行，100ms 后再接收一行。

而是服务器等待几秒，然后一次性接收很多次发送的数据，然后服务器再等几秒，再一次性接收很多数据
这就是我在 `tcp_write` API 函数中写到的：执行 `tcp_write` 之后，并不会立即发送，实际上这个函数仅是将数据放入发送队列中，等待 lwip 认为发生缓冲队列数据够多了，再自动发送

```

while (1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数，所以无操作系统需要一个死循环来不停的维护LWIP

    tcp_write(tcpcb, tcp_sendData, sizeof(tcp_sendData), 1); //实际上这个函数仅是将数据放入发送队列中，但并不会立即发送，等待lwip认为发生缓冲队列数据够多了，再自动发送
    tcp_output(tcpcb); //将发送缓冲队列中的数据立即发送出去
    delay_ms(100);
}

```

加入 `tcp_output` 机制

这个 0x00 是字符串结束符

```

xxxxzzzz
lwip_success_init...
正在查找DHCP服务器,请稍等.....
网卡的MAC地址为:.....2.0.0.52.0.62
通过DHCP获取到IP地址.....192.168.0.102
通过DHCP获取到子网掩码.....255.255.255.0
通过DHCP获取到的默认网关.....192.168.0.1
tcp connect ok
pcb create success

```

F407 连接 TCP 服务器成功

```

root@ubuntu:/home/xzz/pyserver# python3 tcpserver.py
receive : ('192.168.0.102', 49153) : b'stm32f407 tcp data\x00'

```

TCP 服务器 100ms 收到一次数据

TCP 发送数据基本功能测试完成，下面要加入 TCP 断线检测处理，和断线重连，TCP 接收数据处理。

`void tcp_poll(struct tcp_pcb *pcb, tcp_poll_fn poll, u8_t interval)` //设置间隔时间执行 poll

poll 主要用于发一些周期性数据，如心跳包，检查 TCP 连接状态等.....

*pcb：传入 TCP 控制块地址

poll：回调函数，根据设置的间隔时间周期执行，回调原型 `err_t (*tcp_poll_fn)(void *arg, struct tcp_pcb *tpcb)`

interval：间隔执行周期为 `interval*0.5s`，因为 0.5s 是内核定时器的处理周期，如果我定义 1 秒执行一次 poll 回调，那么就填 2，因为 $2 * 0.5s = 1s$ ，把周期算进去了的。

```

tcp pcb = tcp_new(); //创建TCP控制块

if(tcp pcb) //tcp pcb不等于NULL, 表示tcp创建成功
{
    IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
    tcp_connect(tcp pcb, &RemoteAddr, TCP_PORT, tcp_client_connected); //TCP主动连接
    printf("tcp pcb create success\r\n");
}

tcp_err(tcp pcb, client_err); //注册异常处理回调函数, 如果网络中间断了, 或者其它网络故障, 调用该事件

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数, 所以无操作系统需要一个死循环来不停的维护LWIP内核
}

/*
 * tcp 发送数据函数, RAW_TCP的发送函数是事件回调方式发送
 */
static err_t client_send(void *arg, struct tcp_pcb *tpcb) ←
{
    uint8_t send_buf[] = "tcp data send to server"; //TCP要发送的数据

    tcp_write(tpcb, send_buf, sizeof(send_buf), 1); //实际上这个函数仅是将数据放入发送队列中, 不会理解发送
    tcp_output(tpcb); //将发送缓冲队列中的数据立即发送出去

    return ERR_OK;
}

//lwIP TCP连接建立后调用回调函数
err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    printf("connect ok\r\n");
    tcp_poll(tpcb, client_send, 2); //默认0.5秒调用一次TCP发送函数, 这儿是2, 就是2*0.5每1秒发送一次 ←
}

```

lwip_success_init..
正在查找DHCP服务器, 请稍等.....
网卡en0的MAC地址为: 2.0.0.52.0.62
通过DHCP获取到IP地址: 192.168.0.108
通过DHCP获取到子网掩码: 255.255.255.0
通过DHCP获取到的默认网关: 192.168.0.1
tcp connect ok
pcb create success

```

root@ubuntu:/home/xzz/pyserver# python3 tcpserver.py
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'
receive : ('192.168.0.108', 49153) : b'tcp data send to server\x00'

```

获取 DHCP 地址, 开发板 IP 为 192.168.0.108
和我前面 `tcp_poll` 设置的时间 $2 \times 0.5 = 1$ 秒不符。

我发现 `poll` 发送的数据是 2 秒左右执行一次

```

//lwIP TCP连接建立后调用回调函数
err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    printf("connect ok\r\n");
    tcp_poll(tpcb, client_send, 1); //默认0.5秒调用一次TCP发送函数, 这儿是2, 就是2*0.5每1秒发送一次
}

```

`tcp_poll` 改成 1, 执行速度明显变快了。改成 0.5 秒更快。

TCP 客户端和服务器断开的几种检测方式

1. 服务器只是关闭了 TCP 服务端软件, 但是客户端并没有拔断网线, 外网连接还是正常的
`err_t tcp_close(struct tcp_pcb *pcb)` //关闭当前创建的 tcp 控制块

```

while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待DHCP获取成功/超时溢出
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数
}

tcp pcb = tcp_new(); //创建TCP控制块

if(tcp pcb) //tcp pcb不等于NULL, 表示tcp创建成功
{
    IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节
    tcp_connect(tcp pcb, &RemoteAddr, 8888, tcp_client_connected);
    printf("TCP client connect server\r\n");
}

tcp_err(tcp pcb, client_err); //注册异常处理回调函数, 如果网络中断了, 或者其它网络故障, 调用该事件

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数, 所以无操作系统需要一个死循环来不停的维护LWIP内核
}

```

```

//LWIP TCP连接建立后调用回调函数
err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    printf("connect ok\r\n");
    tcp_poll(tpcb, client_send, 1); //默认0.5秒调用一次TCP发送函数, 这儿是2, 就是1*0.5每1秒发送一次
    tcp_recv(tpcb, client_recv); //注册接收回调函数, 用于接收TCP数据
}

```

```

/*接收到TCP的数据, 执行接收事件函数*/
static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p != NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb, p->tot_len); //更新接收窗口, 让LWIP内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf, p->payload, p->len); //将接收数据拷贝进缓冲区, 采样p->len长度才正确
        printf("Recv = %s\r\n", tcp_recvbuf); //打印接收的数据
        memset(p->payload, 0, p->len); //清空接收的数据缓冲区, 不是获取p->tot_len的长度, 而是获取p->len长度
        memset(tcp_recvbuf, 0, p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务端连接断开, 但是网线连接的TCP服务端没有断开, 而是服务端关闭了TCP软件
    {
        //所以TCP服务端断开连接只有在tcp_recv建立的client_recv回调函数中使用err来判断
        //注意:就算我拔掉网线, client_recv回调都不会进入err == ERR_OK分支, 而且我重新接上网线了, 会继续传输
        //只有TCP服务器关闭, 服务端会自动回发最后一个断开TCP的请求. 该分支才执行
        printf("server disconnected\r\n");
        tcp_close(tpcb); //当服务器关闭TCP连接的时候, 客户端自身也要清除TCP控制块
        //执行重新连接TCP函数
    }
    return ERR_OK;
}

```

服务器断开 TCP 连接之后, 不是我们最先以为的在 tcp_err 注册的回调函数执行。而是在接收函数中用判断的方式去探测服务器断开

因为断开连接是服务器软件层面的断开, 所以我们也要关闭掉这次连接服务器的 TCP 控制块

网线拔断检测属于以太网硬件检测, 不属于 lwip 协议栈范畴, 所以要用以太网底层函数实现

在 Lan8720.c 中有一个网线速度检测函数

```

//得到8720的速度模式
//返回值:
//001:10M半双工
//101:10M全双工
//010:100M半双工
//110:100M全双工
//其他:错误.
u8 LAN8720_Get_Speed(void)
{
    u8 speed;
    speed=((ETH_ReadPHYRegister(0x00, 31)&0x1C)>>2); //从LAN8720的31号寄存器中读取网络速度和双工模式
    return speed;
}

```

```

u8 ETH_Status;

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数，所以无操作系统需要一个死循环来不停的维护LWIP内核
    ETH_Status = LAN8720_Get_Speed(); //找个定时器实时检测，或者主循环实时检测网线
    printf("ETH_Status = %x\r\n",ETH_Status);
    delay_ms(100);
}

```

```

ETH_Status = 6

```

网线正常连接，因为我
是100M以太网，所以
显示0x06(110)

```

ETH_Status = 0
ETH_Status = 0
ETH_Status = 0
ETH_Status = 0
ETH_Status = 6

```

将正在运行的网卡，网
线直接拔掉，就会出现
在0和6之间来回跳动

所以网线的断开检测，必须使用 STM32F407 内置的以太网函数来读取 phy 芯片的寄存器判断。

至于路由器是否连接上外网，只有用 ICMP 写一个 ping 百度的程序来测试，这里可以使用 lwip 协议栈来实现

TCP 接收服务器数据

`void tcp_recv(struct tcp_pcb *pcb, tcp_recv_fn recv)` //注册接收 TCP 服务器数据的回调函数

*pcb: 传入 TCP 控制块地址

recv: 注册接收服务器的回调函数，服务器数据到来，回调函数会执行

`err_t (*tcp_recv_fn)(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)` //回调函数原型

`void tcp_recved(struct tcp_pcb *pcb, u16_t len)` //更新时间窗口

用户接收到数据之后，应该调用一下这个函数来更新接收窗口，因为内核不知道应用层是

否真正接收到数据，如果不调用这个函数，内核就没法进行确认，内核就会不停的重复发事件，导致 `tcp_recv_fn` 回调函数不停的重复执行。

*pcb: 传入 TCP 控制块地址

len: 传入 p->tot_len 获取的接收数据长度

```

/*接收到TCP的数据，执行接收事件函数*/
static err_t client_recv(void *arg, struct tcp_pcb *tcpb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tcpb,p->tot_len); //更新接收窗口,让LWIP内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我tcp_recv事件
        printf("Recv = %s\r\n",p->payload); //打印接收的数据
        memset(p->payload,0,p->tot_len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务器连接断开,但是网线连接的TCP服务器没有断开,而是服务器关闭了TCP软件
    {
        printf("server disconnected\r\n");
        tcp_close(tcpb);
    }
    //执行重新连接TCP函数
}
return ERR_OK;
}

//lwIP TCP连接建立后调用回调函数
err_t tcp_client_connected(void *arg, struct tcp_pcb *tcpb, err_t err)
{
    printf("connect ok\r\n");
    tcp_poll(tcpb,client_send,1); //默认0.5秒调用一次TCP发送函数,这儿是2,就是1*0.5每1秒发送一次
    tcp_recv(tcpb,client_recv); //注册接收回调函数,用于接收TCP数据
}

```

lwip 初始化和 TCP 发送代码不变,只需要添加 `tcp_recv` 和接收回调函数即可

```

import socket
from socket import *

tcpserver = socket(AF_INET,SOCK_STREAM)
tcpserver.bind(("192.168.0.112",8888))

tcpserver.listen(5)

while True:
    newsocket,addr = tcpserver.accept()
    while True:
        data = newsocket.recv(1024)
        print("receive : %s : %s" %(str(addr),data))
        newsocket.send("echo server tcp data".encode('utf-8'))

newsocket.close()
tcpserver.close()

```

```

lwip success init
正在查找DHCP服务器,请稍等..... 2.0.0.52.0.62
网卡en的MAC地址为:..... 192.168.0.101
通过DHCP获取到IP地址..... 192.168.0.101
通过DHCP获取到子网掩码..... 255.255.255.0
通过DHCP获取到的默认网关..... 192.168.0.1
tcp connect ok
pcb create success
Recv = echo server top data
Recv = echo server top data.0
Recv = echo server top data.0

```

问题出在 `p->payload`, `payload` 出来接收的有效数据,后面还跟着其它数据,所以不能直接使用 `payload` 获取数据,要使用 `p->len` 将 `payload` 接收的有效数据拷贝到缓冲区,然后打印。

```

uint8_t tcp_recvbuf[UDP_RX_LEN]; //定义UDP接收的数据缓存区 2000字节

/*接收到TCP的数据，执行接收事件函数*/
static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p != NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb, p->tot_len); //更新接收窗口，让LWIP内核指定我在回调函数中接收到数据了，不然内核会重复发送收到的数据包给我tcp_recv事件
        memcpy(tcp_recvbuf, p->payload, p->len); //将接收数据拷贝进缓冲区，采样p->len长度才正确
        printf("Recv = %s\r\n", tcp_recvbuf); //打印接收的数据
        memset(p->payload, 0, p->len); //清空接收的数据缓冲区，不是获取p->tot_len的长度，而是获取p->len长度
        memset(tcp_recvbuf, 0, p->len); //清空接收的数据缓冲区，p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务器连接断开，但是网线连接的TCP服务器没有断开，而是服务器关闭了TCP软件
    {
        printf("server disconnected\r\n");
        tcp_close(tpcb);
    }
    //执行重新连接TCP函数
}
return ERR_OK;
}

```

```

Recv = echo server top data

```

数据接收正常了。

将 payload 数据拷贝到缓存中，使用 p->len 这个长度变量

```

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数，所以无操作系统需要一个死循环来不停的维护LWIP内核
    tcp_write(tpcb, tcp_sendData, sizeof(tcp_sendData), 1); //实际上这个函数仅是将数据放入发送队列中，不会理解发送
    tcp_output(tpcb); //将发送缓冲队列中的数据立即发送出去
    delay_ms(5);
}

```

向 TCP 服务器 5ms 发送一次数据，看看本机 F407 TCP 接收函数会不会有缓冲区溢出情况

```

Recv = echo server top data

```

缓冲区数据接收正常

RAW_TCP 做服务器实验

err_t tcp_bind(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port) //调用 tcp_bind() 函数将本地的 IP 地址、端口号与一个控制块进行绑定

*pcb: 传入 TCP 控制块地址

*ipaddr: 传入 IP_ADDR_ANY, 表示使用 DHCP 给本机分配的 IP

port: 传入本机服务器端口号

struct tcp_pcb * tcp_listen(pcb) //本机进入监听模式

Pcb: 传入 tcp_new()出来的控制块地址

tcp_pcb: tcp_listen 执行后将返回一个新的 TCP 控制块, 所以需要在全局区用 struct tcp_pcb * 创建一个新的 tcp 控制块。

void tcp_accept(struct tcp_pcb *pcb, tcp_accept_fn accept) //监听连接上本机的客户端

*pcb: 传入 tcp_listen()返回的新 tcp 控制块地址

accept: 传入当客户端连接上之后, 执行的回调函数地址

回调函数原型: err_t (*tcp_accept_fn)(void *arg, struct tcp_pcb *newpcb, err_t err)

这个回调函数必须要有返回值, 不然客户端连接不上本服务端。记住....

```
struct tcp_pcb *tcppcb; //必须定义一个TCP控制块,控制块必须是全局,因为很多TCP相关的回调函数需要使用控制块
struct tcp_pcb *tcppcbconn; //另外定义一个TCP服务器控制块

int main(void)
{
    err_t err; //tcp函数执行后返回状态

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为115200
    TIM3_Int_Init(999, 839);
    my_mem_init(SRAMIN); //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池
    printf("xxxxxxxx\r\n");

    while(lwip_comm_init()) //lwip初始化
    {
        printf("lwip_comm_init failed...\r\n");
        delay_ms(1000);
    }

    printf("lwip_success_init...\r\n");

    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待DHCP获取成功/超时溢出
    {
        lwip_periodic_handle(); //LWIP内核需要定时处理的函数
    }
}
```

注意, 如果是 TCP 服务端,
一定是两个 TCP 控制块

初始化和 TCP 客户端 UDP
之类没什么区别

```

tcppcb = tcp_new(); //创建TCP控制块
if(tcppcb) //tcppcb不等于NULL, 表示tcp创建成功
{
    tcp_bind(tcppcb, IP_ADDR_ANY, 8888); //设置本机绑定为TCP服务端
    tcppcbconn = tcp_listen(tcppcb); //进入监听状态
    tcp_accept(tcppcbconn, tcpecho_accept); //客户端连接上本服务端, 执行回调函数
    printf("TCP client connect server\r\n");
}

tcp_err(tcppcb, client_err); //注册异常处理回调函数, 如果网络中断了, 或者其它网络故障, 调用该事件

while(1)
{
    lwip_periodic_handle(); //LWIP内核需要定时处理的函数, 所以无操作系统需要一个死循环来不停的维护LWIP内核
}

/*当有设备连接上本机之后, 会执行该accept事件回调函数*/
static err_t tcpecho_accept(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    tcp_recv(newpcb, server_recv); //注册接收回调函数, 用于接收TCP数据
    return ERR_OK; //在accept回调中, 必须加入ERR_OK, 不然程序无法接受服务端数据
}

```

注意, 这里一定要返回 ERR_OK, 不然会出现 BUG

```

/*接收到TCP的数据, 执行接收事件函数*/
static err_t server_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb, p->tot_len); //更新接收窗口, 让LWIP内核指定我在回调函数中接收到数据了,
                                         //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf, p->payload, p->len); //将接收数据拷贝进缓冲区, 采样p->len长度才正确
        printf("Recv = %s\r\n", tcp_recvbuf); //打印接收的数据
        memset(p->payload, 0, p->len); //清空接收的数据缓冲区, 不是获取p->tot_len的长度, 而是获取p->len长度
        memset(tcp_recvbuf, 0, p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP客户端连接断开, 但是网线连接的TCP客户端没有断开, 而是客户端关闭了TCP软件
    {
        printf("server disconnected\r\n");
        tcp_close(tpcb);

        //执行重新连接TCP函数
    }
    return ERR_OK;
}

```

服务端接收数据代码和前面 TCP 客户端章节没有区别。

```

from socket import *
import time

tcpsocket = socket(AF_INET, SOCK_STREAM)
tcpsocket.connect(("192.168.0.101", 8888))

while True:
    tcpsocket.send("adcd1234".encode("utf-8"))
    print("send data")
    time.sleep(1)

tcpsocket.close()

```

这是客户端发送数据 python 代码

```

Recv = adcd1234

```

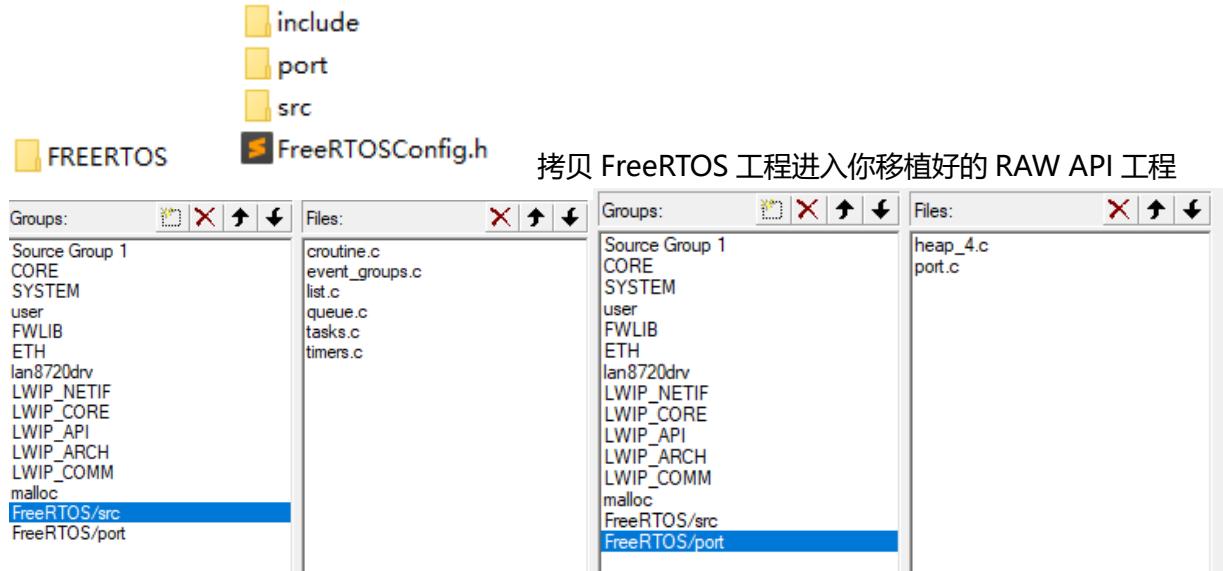
F407 服务端接收正常

LWIP 带 FreeRTOS RAW API 网络通信

这是因为 LWIP 带 FreeRTOS 的 socket 版本没有移植成功, 所以使用 RAW API 来替代的方案

FreeRTOS RAW API 实现

在 STM32F407 使用手册我已经讲过如何在 FreeRTOS 下移植 RAW API 了，这里再回顾下



加载 Src 和 Port 目录的文件

```
.\FREERTOS\include  
\FREERTOS  
\FREERTOS\port\ARM_CM4F
```

加入 FreeRTOS 头文件

```
#include "FreeRTOS.h" //使用 FreeRTOS 里面的函数一定要包含这个头文件  
#include "task.h" //使用 FreeRTOS 里面的函数一定要包含这个头文件  
#include "semphr.h"  
#include <stdarg.h>  
  
SemaphoreHandle_t semDebug = NULL;  
void debug_printf(const char *fmt, ...) //加入 FreeRTOS 模式下的打印方式，需要信号量  
{  
    va_list args;  
    if(xSemaphoreTake(semDebug,5)==pdFALSE) return;  
    va_start(args,fmt);  
    vprintf(fmt,args);  
    va_end(args);  
    xSemaphoreGive(semDebug);  
}  
main()  
{  
    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量  
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1，因为 debug_printf 函数最后一行才会给 semDebug 写 1  
    .....  
}  
  
/* 定时器任务堆栈 */  
static StackType_t Timer_Task_Stack[128];  
/* 定时器任务控制块 */  
static StaticTask_t Timer_Task_TCB;  
void vApplicationGetTimerTaskMemory(StaticTask_t **ppxTimerTaskTCBBuffer,  
                                    StackType_t **ppxTimerTaskStackBuffer,  
                                    uint32_t *pulTimerTaskStackSize)  
{  
    *ppxTimerTaskTCBBuffer=&Timer_Task_TCB; /* 任务控制块内存 */  
    *ppxTimerTaskStackBuffer=Timer_Task_Stack; /* 任务堆栈内存 */  
    *pulTimerTaskStackSize=configTIMER_TASK_STACK_DEPTH; /* 任务堆栈大小 */  
}
```

```

/* 空闲任务堆栈 */
static StackType_t Idle_Task_Stack[128];
/* 空闲任务控制块 */
static StaticTask_t Idle_Task_TCB;

void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer,StackType_t **ppxIdleTaskStackBuffer,uint32_t *pulIdleTaskStackSize )
{
    *ppxIdleTaskTCBBuffer=&Idle_Task_TCB; /* 任务控制块内存 */
    *ppxIdleTaskStackBuffer=Idle_Task_Stack; /* 任务堆栈内存 */
    *pulIdleTaskStackSize=configMINIMAL_STACK_SIZE; /* 任务堆栈大小 */
}

```

在 stm32f4xx_it.c 文件,增加 FreeRTOS 中断函数

```

#include "FreeRTOS.h"
#include "task.h"

extern void xPortSysTickHandler( void ); //加入 FreeRTOS 的中断, 不加入会死机

static __IO uint32_t TimeDelay; //这是之前以太网初始化的延时, 不要动
void SysTick_Handler(void)
{
    if(TimeDelay != 0x00)
    {
        TimeDelay--;
    }

    #if(INCLUDE_xTaskGetSchedulerState == 1) //当调度器启动后 我们才能执行这个函数
    {
        #endif
        xPortSysTickHandler(); //调用 FreeRTOS 中断函数
        #if(INCLUDE_xTaskGetSchedulerState == 1)
    }
    #endif
}

```

修改 lan8720.c 文件的以太网中断配置部分

```

//以太网中断分组配置
void ETHERNET_NVICConfiguration(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    NVIC_InitStructure.NVIC_IRQChannel = ETH_IRQn; //以太网中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0X01; //中断寄存器组 2 最高优先级, 将 STM32 以太网中断优先级降低, 最高优先级留给 FreeRTOS
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0X02;
    NVIC_InitStructure.NVIC_IROChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

主程序使用 LWIP 方式

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为 115200
    TIM3_Int_Init(999,839);
    my_mem_init(SRAMIN); //初始化内部内存池

    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1, 因为 debug_printf 函数最后一行才会给 semDebug 写 1
    printf("xxxxxxxx\r\n");
    while(lwip_comm_init()) //lwip 初始化
    {
        printf("lwip_comm_init failed...\r\n");
        delay_ms(1000);
    }
    printf("lwip_success_init..\r\n");
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF))//等待 DHCP 获取成功/超时溢出
    {
        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数
    }
}

```

```

//任务启动之前，可以使用裸机方式初始化 LWIP 和 dhcp，如上所写
//创建开始任务
xTaskCreate((TaskFunction_t )NetWork_task,           //任务函数 网络通信 LWIP 内核中断维护
            (const char* )"NetWork_task",      //任务名称
            (uint16_t )512,                 //任务堆栈大小
            (void* )NULL,                  //传递给任务函数的参数
            (UBaseType_t )2,                //任务优先级
            (TaskHandle_t* )&NetWorkTask_Handler); //任务句柄

//创建任务
xTaskCreate((TaskFunction_t )Task2,           //任务函数
            (const char* )"Task2",      //任务名称
            (uint16_t )512,                 //任务堆栈大小
            (void* )NULL,                  //传递给任务函数的参数
            (UBaseType_t )3,                //任务优先级
            (TaskHandle_t* )&Task2_Handler); //任务句柄

vTaskStartScheduler();           //开启任务调度

while(1)
{
}

}

/******任务执行的具体方式如下******/
TaskHandle_t NetWorkTask_Handler;

void NetWork_task(void *pvParameters)
{
    debug_printf("NetWork_task Init...\r\n");
    while(1)
    {
        debug_printf("NetWork_task run...\r\n");
        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数，现在将无操作系统死循环维护的 LWIP 内核，放入有操作系统的线程里面死循环(移植主要就是这一步)
        vTaskDelay(10); //延时 10ms 因为 lwip_periodic_handle 里面的 ARP, TCP 和 DHCP 都是几百毫秒处理一次，所以用 10ms 间隔没有问题
    }
}
//任务句柄
TaskHandle_t Task2_Handler;

void Task2(void *pvParameters)
{
    debug_printf("Task2 Init\r\n");
    while(1)
    {
        debug_printf("task22222 run...\r\n");
        vTaskDelay(100); //延时 10ms
    }
}

```

这就是 RAW API 在 FreeRTOS 移植过程，我服务器测试了一下是可以 ping 通的，证明 RAW API 在 FreeRTOS 下可以使用。

FreeRTOS 下 RAW API 的 UDP 测试

```

/*接收 UDP 服务器数据的事件回调函数，现在实现接收数据的事件函数*/
void udp_RecvData(void *arg,struct udp_pcb *upcb,struct pbuf *p,struct ip_addr *addr,u16_t port)
{
    u32 data_len = 0;
    struct pbuf *q;

    if(p!=NULL) //接收到不为空的数据时，也就是 UDP 服务器发送数据过来了
    {
        memset(udp_recvbuf,0,UDP_RX_LEN); //数据接收缓冲区清零，将以前或者上一次的数据清0
        for(q=p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
        {
            //判断要拷贝到 q->payload 中的数据长度是否大于 UDP_RX_LEN 的剩余空间，如果大于
            //的话就只拷贝 UDP_RX_LEN(2000)中剩余长度的数据，否则的话就拷贝所有的数据
            if(q->len > (UDP_RX_LEN-data_len))
            {

```

```

        memcpy(udp_recvbuf+data_len,q->payload,(UDP_RX_LEN-data_len)); //拷贝数据 UDP_RX_LEN(2000)字节
    }
    else
        memcpy(udp_recvbuf+data_len,q->payload,q->len); //如果接收数据不大于 2000,
        //就拷贝< 2000 字节以 q->len 为准
    data_len += q->len; //将本次 for 循环拷贝的数据长度给 data_len,
    //下次循环地址接着累加 data_len 之后的地址开始拷贝

    if(data_len > UDP_RX_LEN) //超出 UDP 客户端接收数组的大小,跳出
        break;
}
debug_printf("Recv = %s\r\n",udp_recvbuf);

upcb->remote_ip= *addr; //记录远程主机的 IP 地址
upcb->remote_port= port; //记录远程主机的端口号

uint8_t IADDR4,IADDR3,IADDR2,IADDR1;
IADDR1 = upcb->remote_ip.addr&0xff; //得到 192.
IADDR2 = (upcb->remote_ip.addr>>8)&0xff; //得到 168.
IADDR3 = (upcb->remote_ip.addr>>16)&0xff; //得到 0.
IADDR4 = (upcb->remote_ip.addr>>24)&0xff; //得到 112
debug_printf("Recv IP %d . %d . %d . %d  port = %d\r\n",IADDR1,IADDR2,IADDR3,IADDR4,upcb->remote_port);
pbuf_free(p); //每次接受完一定要释放内存,不然会出现 pbuf_take: invalid buf" failed
}

}

//任务句柄
TaskHandle_t Task2_Handler;

void Task2(void *pvParameters)
{
    const uint8_t udp_sendData[] = "stm32f407 udp data"; //这就是 UDP 要发送的数据
    struct udp_pcb *udppcb; //必须定义一个 UDP 控制块
    struct ip_addr RemoteAddr; //保存远端服务器 IP 地址的变量
    err_t err; //udp 函数执行后返回状态
    debug_printf("Task2 Init\r\n");

    udppcb = udp_new(); //创建 udp
    if(udppcb) //udppcb 不等于 NULL, 表示创建成功
    {

        IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分 10 进制 IP 转为 4 字节变量然后将 IP 赋值给 ip_addr 定义的变量,
        //因为 IP 控制块里面通过 32 位存储 IP 地址
        err = udp_connect(udppcb,&RemoteAddr,UDP_PORT); //设置远端 IP 和端口
        if(err == ERR_OK) //udp 执行链接成功, 但不是真正的链接哈, UDP 没有链接这种功能
        {
            err = udp_bind(udppcb,IP_ADDR_ANY,UDP_PORT); //设置本地 IP 和端口, 其中 IP_ADDR_ANY 表明使用 DHCP 网卡地址,
            udp_recv(udppcb , udp_RecvData , NULL); //注册控制块回调函数, 端口收到数据, 回调处理函数将被调用。
            //在发送函数里会判断这个变量, 选择网卡地址
            debug_printf("udp connect success\r\n");
        }
    }

    struct pbuf *ptr; //在裸机 RAW API 是死循环前先发送一次 UDP。
    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData),PBUF_POOL);

    if(ptr) //如果内存分配成功, 发送数据
    {
        pbuf_take(ptr,(char*)udp_sendData,strlen((char*)udp_sendData));
        udp_send(udppcb,ptr); //udp 发送处理之后的数据
        pbuf_free(ptr); //释放内存
    }
    while(1) //死循环中再次申请 UDP, 再次发送
    {
        debug_printf("udp while send\r\n");
        //*****循环发送 UDP, 每次发送都要分配内存->发送数据->释放内存, 必须这样, 不然会出现 bug*****
        ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData),PBUF_POOL);
        pbuf_take(ptr,(char*)udp_sendData,strlen((char*)udp_sendData));
        debug_printf("%s\r\n",udp_sendData);
        udp_send(udppcb,ptr); //udp 发送处理之后的数据
        pbuf_free(ptr); //释放内存
    }
    vTaskDelay(10); //延时 10ms
}
}

```



后台接收一次 UDP 数据之后，就再也没有数据了，STM32 这个也是卡住了

问题就出在 while 循环之前分配了一次 pbuf，发送了一次数据。然后 while 再循环发送就不行了，好奇怪。这可能是 STM32F4 内部 SRAM 内存太小造成的，其它 MCU/CPU 平台应该不会出现这种情况。

```

void Task2(void *pvParameters)
{
    const uint8_t udp_sendData[] = "stm32f407 udp data"; //这就是 UDP 要发送的数据
    struct udp_pcb *udppcb; //必须定义一个 UDP 控制块
    struct ip_addr RemoteAddr; //保存远端服务器 IP 地址的变量
    err_t err; //udp 函数执行后返回状态

    debug_printf("Task2 Init\r\n");

    udppcb = udp_new(); //创建 udp
    if(udppcb) //udppcb 不等于 NULL, 表示创建成功
    {

        IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分 10 进制 IP 转为 4 字节变量然后将 IP 赋值给 ip_addr 定义的变量,
                                                       //因为 IP 控制块里面通过 32 位存储 IP 地址
        err = udp_connect(udppcb, &RemoteAddr, UDP_PORT); //设置远端 IP 和端口
        if(err == ERR_OK) //udp 执行链接成功, 但不是真正的链接哈, UDP 没有链接这种功能
        {
            err = udp_bind(udppcb, IP_ADDR_ANY, UDP_PORT); //设置本地 IP 和端口, 其中 IP_ADDR_ANY 表明使用 DHCP 网卡地址,
            udp_recv(udppcb, udp_RecvData, NULL); //注册控制块回调函数, 端口收到数据, 回调处理函数将被调用。
                                                       //在发送函数里会判断这个变量, 选择网卡地址
            debug_printf("udp connect success\r\n");
        }
    }

    //*****前面连接 UDP 方式还是不变*****
    struct pbuf *ptr; //在裸机 RAW API 是死循环前先发送一次 UDP
    ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char *)udp_sendData), PBUF_POOL); //主要是屏蔽了在死循环之前分配内存发送 UDP 的功能
    if(ptr) //如果内存分配成功, 发送数据
    {
        pbuf_take(ptr, (char *)udp_sendData, strlen((char *)udp_sendData));
        udp_send(udppcb, ptr); //udp 发送处理之后的数据
        pbuf_free(ptr); //释放内存
    }
    while(1) //死循环中再次申请 UDP, 再次发送
    {
        debug_printf("udp while send\r\n");
        //*****循环发送 UDP, 每次发送都要分配内存->发送数据->释放内存, 必须这样, 不然会出现 bug***** //不知道是不是和这个流程有关
        ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char *)udp_sendData), PBUF_POOL);
        pbuf_take(ptr, (char *)udp_sendData, strlen((char *)udp_sendData));
        debug_printf("%s\r\n", udp_sendData);
        udp_send(udppcb, ptr); //udp 发送处理之后的数据
        pbuf_free(ptr); //释放内存
        vTaskDelay(10); //延时 10ms
    }
}

```

```

stm32f407 udp data
('192.168.0.100', 8888)

```

你看 UDP 收发正常了。

下面测试下 UDP 变化数据的发送性能

```

//任务句柄
TaskHandle_t Task2_Handler;

void Task2(void *pvParameters)
{
    const uint8_t udp_sendData1[] = "11111111111111111111111111111111"; //这就是 UDP 要发送的数据
    const uint8_t udp_sendData2[] = "22222222222222222222222222222222"; //这就是 UDP 要发送的数据
    const uint8_t udp_sendData3[] = "33333333333333333333333333333333"; //这就是 UDP 要发送的数据
    const uint8_t udp_sendData4[] = "44444444444444444444444444444444"; //这就是 UDP 要发送的数据
    const uint8_t udp_sendData5[] = "55555555555555555555555555555555"; //这就是 UDP 要发送的数据
    ......



连接上 UDP 服务器
while(1) //死循环中再次申请 UDP, 再次发送
{
    //*****循环发送 UDP, 每次发送都要分配内存->发送数据->释放内存,必须这样, 不然会出现 bug*****
    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData1),PBUF_POOL);
    pbuf_take(ptr,(char*)udp_sendData1,strlen((char*)udp_sendData1));
    debug_printf("%s\r\n",udp_sendData1);
    udp_send(udppcb,ptr); //udp 发送处理之后的数据
    pbuf_free(ptr); //释放内存
    vTaskDelay(100); //延时 10ms

    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData2),PBUF_POOL);
    pbuf_take(ptr,(char*)udp_sendData2,strlen((char*)udp_sendData2));
    debug_printf("%s\r\n",udp_sendData2);
    udp_send(udppcb,ptr); //udp 发送处理之后的数据
    pbuf_free(ptr); //释放内存

    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData3),PBUF_POOL);
    pbuf_take(ptr,(char*)udp_sendData3,strlen((char*)udp_sendData3));
    debug_printf("%s\r\n",udp_sendData3);
    udp_send(udppcb,ptr); //udp 发送处理之后的数据
    pbuf_free(ptr); //释放内存

    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData4),PBUF_POOL);
    pbuf_take(ptr,(char*)udp_sendData4,strlen((char*)udp_sendData4));
    debug_printf("%s\r\n",udp_sendData4);
    udp_send(udppcb,ptr); //udp 发送处理之后的数据
    pbuf_free(ptr); //释放内存

    ptr = pbuf_alloc(PBUF_TRANSPORT,strlen((char *)udp_sendData5),PBUF_POOL);
    pbuf_take(ptr,(char*)udp_sendData5,strlen((char*)udp_sendData5));
    debug_printf("%s\r\n",udp_sendData5);
    udp_send(udppcb,ptr); //udp 发送处理之后的数据
    pbuf_free(ptr); //释放内存
    vTaskDelay(10); //延时 10ms
}

```

```

('192.168.0.100', 8888)
1111111111111111111111111111111111111111111111111111111111
('192.168.0.100', 8888)
222222222222222222222222222222222222222222222222222222222222222222

```

连续发送两次，系统就卡死了，多半是 pbuf_alloc，多次内存连续分配造成的。

所以我们最好把数据分配到一块内存区域上，在这块内存区域上不断的修改数据。STM32F4 内存太小了。

```

//任务句柄
TaskHandle_t Task2_Handler;
void Task2(void *pvParameters)
{
    uint8_t udp_sendData[1] = {0,1,2,3,4,5,6,7,8,9}; //这就是UDP要发送的数据
    struct udp_pcb *udppcb; //必须定义一个UDP控制块
    struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
    err_t err; //udp函数执行后返回状态
    debug_printf("Task2 Init\r\n");
    udppcb = udp_new(); //创建udp
    if(udppcb) //如果udppcb不等于NULL, 表示创建成功
    {
        IP4_ADDR(&RemoteAddr, 192,168,0,112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量
        //（因为IP控制块里面通过32位存储IP地址）
        err = udp_connect(udppcb, &RemoteAddr, UDP_PORT); //设置远端IP和端口
        if(err == ERR_OK) //udp执行链接成功, 但不是真正的链接哈, UDP没有链接这种功能
        {
            err = udp_bind(udppcb, IP_ADDR_ANY, UDP_PORT); //设置本地IP和端口, 其中IP_ADDR_ANY表明使用DHCP网卡地址,
            udp_recv(udppcb, udp_RecvData, NULL); //注册回调块回调函数, 端口收到数据, 回调处理函数将被调用。
            debug_printf("udp connect success\r\n");
        }
    }
    struct pbuf *ptr; //在裸机RAW API 是死循环前先发送一次UDP
    // ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char*)udp_sendData), PBUF_POOL);
    //
    // if(ptr)//如果内存分配成功, 发送数据
    // {
    //     pbuf_take(ptr,(char*)udp_sendData,strlen((char*)udp_sendData));
    //     udp_send(udppcb,ptr); //udp发送处理之后的数据
    //     pbuf_free(ptr); //释放内存
    // }

    while(1) //死循环中再次申请UDP, 再次发送
    {
        //*****每次发送都要分配内存->发送数据->释放内存, 必须这样, 不然会出现bug*****
        for(int i = 0; i < 100; i++)
        {
            ptr = pbuf_alloc(PBUF_TRANSPORT, strlen((char*)udp_sendData), PBUF_POOL);
            pbuf_take(ptr,(char*)udp_sendData,strlen((char*)udp_sendData));
            debug_printf("%d\r\n",udp_sendData);
            udp_send(udppcb,ptr); //udp发送处理之后的数据
            pbuf_free(ptr); //释放内存

            udp_sendData[0] = + i;
            udp_sendData[1] = + i;
            udp_sendData[2] = + i;
            udp_sendData[3] = + i;
            udp_sendData[4] = + i;
            udp_sendData[5] = + i;
            udp_sendData[6] = + i;
            udp_sendData[7] = + i;
            udp_sendData[8] = + i;
            udp_sendData[9] = + i;

            vTaskDelay(100); //延时10ms
        }
    }
}

```

我就定义一块固定长度的内存

每次分配内存，都分配
udp_sendData1 这块固
定地址，固定长度的内存

每次要发送不一样的新
数据，都放在这块固定
长度的内存上

```

SSCOM V5.13.1 串口/网络数据流调试器,作者:大
通讯端口 串口设置 显示 发送 多字符串 小
('192.168.0.100', 8888)
$$$$$$$$$$
('192.168.0.100', 8888)
%%%%%%%%%
('192.168.0.100', 8888)
&&&&&&&&&
('192.168.0.100', 8888)
::::::::::
('192.168.0.100', 8888)
(((((((((((
('192.168.0.100', 8888)
))))))))))
('192.168.0.100', 8888)
*****

```

固定长度内存，测试很稳定。

所以 LWIP RAW API 的 UDP 开发，只有和后端工程师讨论好固定长度的协议，再开发。

FreeRTOS 下 RAW API 的 TCP 测试

```
#include "lwip/netif.h"
#include "lwip_comm.h"
#include "lwipopts.h"
#include "lwip/pbuf.h" //TCP 加入头文件
#include "lwip/tcp.h" //TCP 加入头文件
#include <string.h>
#include "FreeRTOS.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件
#include "task.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件
最基本的这些头文件要有
```

```
struct tcp_pcb *tcppcb = NULL; //必须定义一个 TCP 控制块
TCP 线程如下:
err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    debug_printf("connect OK\r\n");
}
static void client_err(void *arg, err_t err)
{
    tcp_close(tpcb);
}
//任务句柄
TaskHandle_t Task2_Handler;

void Task2(void *pvParameters)
{
    uint8_t TCP_sendData[] = "12345678"; //这就是 TCP 要发送的数据
    struct ip_addr RemoteAddr; //保存远端服务器 IP 地址的变量
    err_t err; //tcp 函数执行后返回状态

    debug_printf("Task2 Init\r\n");
    tcppcb = tcp_new();

    if(tpcb) //udppcb 不等于 NULL, 表示创建成功
    {
        IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分 10 进制 IP 转为 4 字节变量然后将 IP 赋值给 ip_addr 定义的变量,
        //因为 IP 控制块里面通过 32 位存储 IP 地址
        err = tcp_connect(tpcb, &RemoteAddr, 8888, tcp_client_connected); //设置远端 IP 和端口 8888
        vTaskDelay(1); //连接客户端 TCP 之后必须延时, 不然直接死机。神奇, 可能是加入 FreeRTOS 造成的, 一定要记住。
        debug_printf("tcp create success\r\n");
    }
    tcp_err(tpcb, client_err);

    while(1) //死循环中再次申请 UDP, 再次发送
    {
        tcp_write(tpcb, TCP_sendData, sizeof(TCP_sendData), 1);
        tcp_output(tpcb);
        vTaskDelay(100); //延时 10ms
    }
}
receive : ('192.168.0.100', 49153) : b'12345678\x00'

xxxxxxxxx
lwip success init...
正在查找DHCP服务器, 请稍等..... 2.0.0.52.0.62
网卡en0 MAC地址为: 2.0.0.52.0.62
通过DHCP获得到IP地址..... 192.168.0.100
通过DHCP获得到子网掩码..... 255.255.255.0
通过DHCP获得到的默认网关..... 192.168.0.1
NetWork_task Init...
Task2 Init
top connect success
connect OK
```

测试，TCP 发送数据到服务器成功。

下面测试 TCP 发送接收功能

```
/*********************************************TCP 任务处理测试******/
uint8_t tcp_recvbuf[1000]; //定义 TCP 接收的数据缓存区 1000 字节
struct tcp_pcb *tcppcb = NULL; //必须定义一个 TCP 控制块
static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示 pbuf 接收到服务器的数据
    {
        tcp_recved(tpcb, p->tot_len); //更新接收窗口, 让 LWIP 内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我 tcp_recv 事件
```

```

memcpy(tcp_recvbuf,p->payload,p->len);//将接收数据拷贝进缓冲区，采样 p->len 长度才正确
printf("Recv = %s\n",tcp_recvbuf);//打印接收的数据
memset(p->payload,0,p->len);//清空接收的数据缓冲区，不是获取 p->tot_len 的长度，而是获取 p->len 长度
memset(tcp_recvbuf,0,p->len);//清空接收的数据缓冲区，p->tot_len 就是接收的数据长度
pbuf_free(p);
}
else if(err == ERR_OK) //表示 TCP 服务端连接断开，但是网线连接的 TCP 服务端没有断开，而是服务端关闭了 TCP 软件
{
    //所以 TCP 服务端断开连接只有在 tcp_recv 建立的 client_recv 回调函数中使用 err 来判断
    //注意：就算我拔掉网线，client_recv 回调都不会进入 err == ERR_OK 分支，而且我重新接上网线了，会继续传输
    // 只有 TCP 服务器关闭，服务端会自动回发最后一个断开 TCP 的请求。该分支才执行
    printf("server disconnected\n");
    tcp_close(tpcb); //当服务器关闭 TCP 连接的时候，客户端自身也要清除 TCP 控制块
    //执行重新连接 TCP 函数.....
}
return ERR_OK;
}

err_t tcp_client_connected(void*arg, struct tcp_pcb *tpcb, err_t err)
{
    debug_printf("connect OK\n");
    tcp_recv(tpcpcb,client_recv); //注册接收回调函数，用于接收 TCP 数据
}

static void client_err(void *arg, err_t err)
{
    tcp_close(tpcpcb);
}

//任务句柄
TaskHandle_t Task2_Handler;
void Task2(void *pvParameters)
{
    uint8_t TCP_sendData[] = "12345678"; //这就是 TCP 要发送的数据
    struct ip_addr RemoteAddr; //保存远端服务器 IP 地址的变量
    err_t err; //tcp 函数执行后返回状态
    debug_printf("Task2 Init\n");
    tpcpcb = tcp_new();
    if(tpcpcb) //udppcb 不等于 NULL，表示创建成功
    {
        IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分 10 进制 IP 转为 4 字节变量然后将 IP 赋值给 ip_addr 定义的变量,
        //因为 IP 控制块里面通过 32 位存储 IP 地址
        err = tcp_connect(tpcpcb,&RemoteAddr,8888,tcp_client_connected); //设置远端 IP 和端口 8888
        vTaskDelay(1); //连接客户端 TCP 之后必须延时，不然直接死机。神奇
        debug_printf("tcp create success\n");
    }
    tcp_err(tpcpcb , client_err);
    while(1) //死循环中再次申请 UDP，再次发送
    {
        tcp_write(tpcpcb,TCP_sendData,sizeof(TCP_sendData),1);
        tcp_output(tpcpcb);
        vTaskDelay(100); //延时 10ms
    }
}

```

客户端程序已写完

```

2 from socket import *
3
4 tcpserver = socket(AF_INET,SOCK_STREAM)
5 tcpserver.bind(("192.168.0.112",8888))
6
7 tcpserver.listen(5)
8
9 while True:
10     newsocket,addr = tcpserver.accept()
11     while True:
12         data = newsocket.recv(1024)
13         if len(data) >0 :
14             print("receive : %s : %s" %(str(addr),data))
15             newsocket.send("echo tcp server data".encode('utf-8'))
16         else:
17             print("client disconnet");
18
19     newsocket.close()
20
21 tcpserver.close()

```

服务器 python 程序

数据发送到服务器，接收服务器数据成功

LwIP SOCKET 网络接口使用

LwIP Socket API 是基于 NETCONN API 之上实现的。

使用 Socket API 之前，需要在 lwipopts.h 文件中定义
#define LWIP_NETCOMM 1
#define LWIP_SOCKET 1

int lwip_socket(int domain, int type, int protocol) //创建 socket

domain: 常用 AF_INET 参数(IPV4)，如果是 IPV6 就要使用 AF_INET6

type: SOCK_STREAM: 主要用于 TCP 传输

SOCK_DGRAM: 主要用于 UDP 传输

SOCK_RAW: 原始套接字，用于应用程序访问 7 层网络协议栈的网络层原始数据包

protocol: 对于 TCP 和 UDP 协议，protocol 都写入 0

返回值: 就是创建的 socket 描述符，后面都要操作这个描述符。貌似 lwip 只能同时创建 4 个 socket

int lwip_bind(int s, const struct sockaddr *name, socklen_t namelen) //服务端才会用的端口绑定

s: 传入前面 lwip_socket 创建之后返回的文件描述符

*name: 虽然数据类型是 sockaddr，但是实际当中我们都是使用 sockaddr_in 类型来定义

int lwip_connect(int s, const struct sockaddr *name, socklen_t namelen) //用于客户端连接服务器

s: 传入前面 lwip_socket 创建之后返回的文件描述符

int lwip_listen(int s, int backlog) //用于服务端监听程序，主要用于 TCP 服务器中

s: 传入前面 lwip_socket 创建之后返回的文件描述符

backlog:

int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen) //等待客户端连接服务器，该函数必须配合 listen 函数使用

int lwip_recvfrom(int s, void *mem, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)

//用于接收数据，不管是 TCP 还是 UDP 应用中，都可以使用 recvfrom 来接收数据。

s: 传入前面 lwip_socket 创建之后返回的文件描述符

*mem: 传入接收数据的缓存起始地址

len: 指定接收数据的最大长度，如果接收成功，返回接收数据的实际长度给 int

int lwip_recv(int s, void *mem, size_t len, int flags)

s: 传入前面 lwip_socket 创建之后返回的文件描述符

*mem: 传入接收数据的缓存起始地址

len: 指定接收数据的最大长度，如果接收成功，返回接收数据的实际长度给 int

flags: 拷贝或者不拷贝的标志位

```
int lwip_sendto(int s, const void *data, size_t size, int flags, const struct sockaddr *to, socklen_t tolen) //用于 UDP 协议中的发送数据函数
```

```
int lwip_send(int s, const void *data, size_t size, int flags)
```

```
int lwip_write(int s, const void *data, size_t size) //针对 TCP 协议的发送数据函数
```

s: 传入前面 lwip_socket 创建之后返回的文件描述符

```
int lwip_close(int s) //关闭一个套接字
```

TCP 连接实验(ESP32 版本), 因为 ESP32 已经固化了 LWIP 协议栈

```
void app_main()
{
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );

    tcpip_adapter_init(); //初始化 TCP/IP 适配层 不然启动 wifi 模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) );

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = "TP-LINK_6-1", //要连接设备 SSID
            .password = "65526235", //要连接设备密码
        },
    };
    MDF_LOGI("Setting WiFi configuration SSID %s...", wifi_config.sta.ssid);
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi 启动为 STA 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) ); //给 wifi 接口设置参数, STA 和 AP
    //SSID 和密码

    ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi
```

前面蓝色框里面, 是标准连接 wifi 路由器的方法

```
while(wificonnectFlag == 0) //wifi没有连接成功, 不允许使用socket, wificonnectFlag = 1 表示wifi连接成功
{
    sleep(1);
}
```

后面 socket 建立之前, 为什么要死循环判断标志位, 还要加延时,
请查阅我的<ESP32 system peripheral userguide>手册

```
sleep(5); //这里需要延时5秒, 因为连接上wifi后, 路由器还没有来得及分配IP地址, 就建立SOCKET, 也会连接TCP失败
/*TCP Socket实验*/
int sockfd = 0;
sockfd = lwip_socket(AF_INET, SOCK_STREAM, IPPROTO_IP); //创建socket TCP连接
if(sockfd < 0) //socket创建失败
    MDF_LOGI("socket creat error");
```

```

struct sockaddr_in client_addr; //IP, 端口号配置结构体

client_addr.sin_family = AF_INET;
client_addr.sin_port = htons(8888); //TCP服务器端口号
client_addr.sin_addr.s_addr = inet_addr("192.168.0.104");

memset(&(client_addr.sin_zero), 0, sizeof(client_addr.sin_zero));

if(lwip_connect(sockfd,(struct sockaddr *)&client_addr , sizeof(struct sockaddr)) == -1)
    MDF_LOGI("socket connect TCP failed"); //TCP连接失败, 可以用while循环建立重复connect
else
    MDF_LOGI("socket connect TCP success");

char send[20] = "esp32....123456"; //要发送的数据
while(1){

    if(lwip_write(sockfd,send,sizeof(send))<0)
        break; //如果发送失败, 跳出死循环, 关闭socket
    MDF_LOGI(" TCP send");
    sleep(1);
}

```

TCP 客户端建立完毕

使用 python3 实现的 TCP 服务端

```

import socket
from socket import *
import time
import _thread

def multClient(newsocket,addr):
    while True:
        recvdata = newsocket.recv(1024)
        if len(recvdata) > 0:
            print("addr = %s data = %s" %(addr,recvdata))
        else:
            print("client close")
            break
    newsocket.close()

def main():
    tcpserver = socket(AF_INET,SOCK_STREAM)
    tcpserver.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
    tcpserver.bind(('',8888))
    tcpserver.listen(5)

    try:
        while True:
            newsocket,addr = tcpserver.accept()
            sleepboxClient = _thread.start_new_thread(multClient,(newsocket,addr))
    finally:
        tcpserver.close()

if __name__ == "__main__":
    main()

```

```

I (551) wifi.state: assoc -> run (10)
I (781) wifi:connected with TP-LINK_6-1, aid = 2, channel 11, BW20, bssid = d0:76:e7:9f:14:4a
I (781) wifi:security type: 4, phy: bgn, rssi: -65
I (791) wifi:pm start, type: 1

I (801) [==SleepBox==, 20]: connect AP success
I (841) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1691) event: sta ip: 192.168.0.102, mask: 255.255.255.0, gw: 192.168.0.1
I (6381) [==SleepBox==, 90]: socket connect TCP success
I (6381) [==SleepBox==, 98]: TCP send
I (7391) [==SleepBox==, 98]: TCP send
I (8391) [==SleepBox==, 98]: TCP send
I (9391) [==SleepBox==, 98]: TCP send
I (10391) [==SleepBox==, 98]: TCP send
I (11391) [==SleepBox==, 98]: TCP send
I (12391) [==SleepBox==, 98]: TCP send
I (13391) [==SleepBox==, 98]: TCP send
I (14391) [==SleepBox==, 98]: TCP send
I (15391) [==SleepBox==, 98]: TCP send
I (16391) [==SleepBox==, 98]: TCP send
I (17391) [==SleepBox==, 98]: TCP send

```

客户端发送数据

```

addr = ('192.168.0.102', 51254) data = b'esp32....123456\x00\x00\x00\x00\x00\x00'

```

服务端接收成功

LWIP 实现 http 协议

URL语法

如: `http://www.baidu.com/search/test.txt`

解析: `<scheme>:<user>:<password>@<host>:<port>/<path>; <params>?<query>#<frag>`

`<scheme>`: 指定你访问服务器用哪种协议访问, 有`http`, `https`, `ftp`几种协议。根据以上URL 应该是`http`方式访问

`<user>:<password>` 用户名和密码很少使用

`<host>`: 用`http`协议访问哪一台服务器, 所以`host`是填入服务器域名, 或者服务器IP都可以, 让本机知道去连接哪一台服务器

`<path>`: 访问路径, 就是你要下载到本机浏览器页面的文本, 图片, 视频, 在服务器硬盘什么路径

`<params>?<query>#<frag>` 这3个参数使用得少



我们用 postman 软件测试下请求 http 流程

The screenshot shows the Postman interface with the following steps:

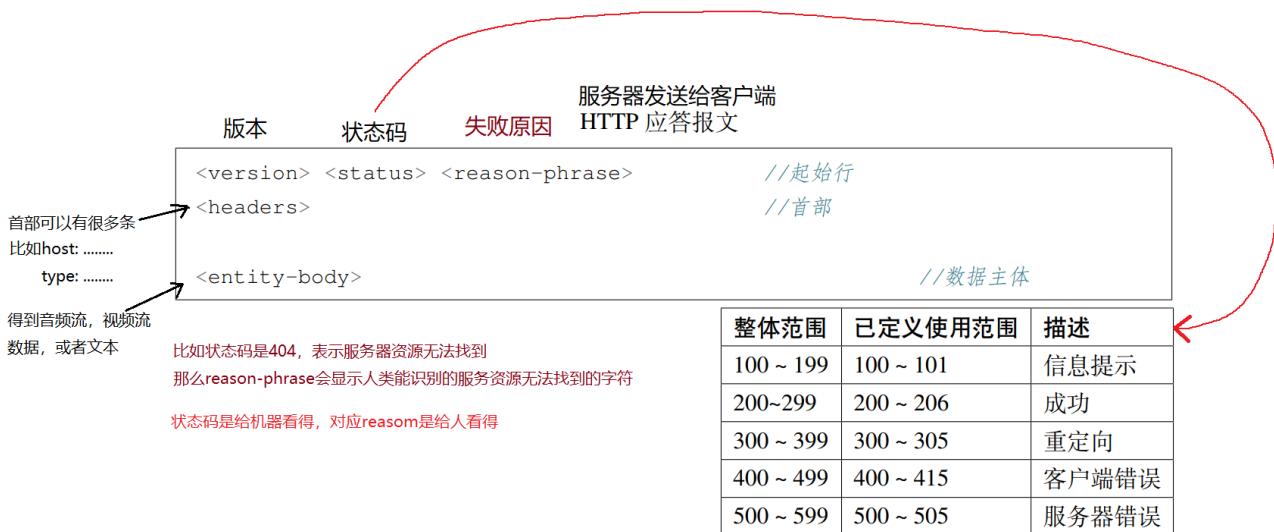
- 1 我写入百度地址, 点击 Send 发送
- 2.点击 Code 就能看到发送的什么字段

The request details are as follows:

- Method: GET
- URL: www.baidu.com
- Headers (21):
 - Host: www.baidu.com
 - cache-control: no-cache
 - Postman-Token: 4addbbfd-c203-48b6-b33e-b8776692aeecc
- Body (Pretty):
 - 1 GET HTTP/1.1
 - 2 Host: www.baidu.com
 - 3 cache-control: no-cache
 - 4 Postman-Token: 4addbbfd-c203-48b6-b33e-b8776692aeecc
 - 5

The response status is 200 OK. The message structure is annotated with arrows pointing to specific fields in the request body:

- HTTP 请求报文
- //起始行
- //首部
- //数据主体
- 方法: <method>
- 请求网址: <request-URL>
- 版本: <version>
- 首部一般—> <headers>
- 是网址
- 数据随便自定义
- 很多时候这里为空
把网址/路径交给
<headers>来填
- 版本, 就是1.1



baike.baidu.com/item/人工智能/9180?fromtitle=AI&fromid=25417&fr=aladdin
比如我拷贝
这一段地址过去

你看，这次发送就很符合我们 http 请求格式，这个 URL 就是我们获取资源的路径

Copy to Clipboard

HTTP

```

1 GET /item/%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD/9180?fromtitle=AI&amp; fromid=25417&amp; fr=aladdin HTTP/1.1
2 Host: baike.baidu.com
3 cache-control: no-cache
4 Postman-Token: f3a02881-9156-4600-b80c-8d9e5a8ced7b
5

```

填入GET或者POST, HEAD这些请求方法

填入请求的网址/路径
很多时候这里为空
把网址/路径交给
<headers>来填

版本, 就是1.1

HTTP 请求报文

HTTP 应答报文

首部一般
是网址

<method> <request-URL> <version>

//起始行
//首部

<headers>

<entity-body>

//数据主体

数据随便自定义

更多的 http 原理及实现，请查阅我的《pythonMQTT_HTTP_userguide》手册，一看就知道怎么用了。

RAW API http 实现

《pythonMQTT_HTTP_userguide》手册中讲解了 http 服务器如何搭建，下面就用单片机写 http 客户端，在 TCP 这一层实现 http 传输。

GET 请求实现，也就是单片机只获取 http 服务器数据

python 服务器手动实现的 http 程序

```
import socket

if __name__ == '__main__':
    tcp_server = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #创建TCP服务器SOCKET套接字
    tcp_server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,True) #绑定端口号
    tcp_server.bind(("","8888")) # 默认使用本机IP, http协议端口官方要求80但是我可以自定义为8888, 只要浏览器指定8888就行
    tcp_server.listen(128) #设置监听

    while True:
        new_socket, ip_port = tcp_server.accept() #等待接收客户端的连接请求
        recv_data = new_socket.recv(4096) #如果网页采用的GET请求, 那么网页客户端发的数据最多不会超过4K
        print(recv_data) #接收的浏览器数据就不打印了, 免得混乱

        with open("xzz/index.html","r") as file: #with open 的好处就是打开指定文件, 不需要手动close关闭文件
            file_data = file.read() #读取指定文件内容

            response_line = "HTTP/1.1 200 OK\r\n" #响应行
            response_header = "Server: PWS/1.0\r\n" #响应头
            response_null = "\r\n" #空行
            response_body = file_data #响应体(响应体就是发的html文件, 也可以是txt, 或者其它数据)

            response = response_line + response_header + response_null + response_body
            new_socket.send(response.encode("utf-8")) #将封装的数据发出去
            new_socket.close() #关闭本次为客户端创建的socket

<!DOCTYPE html>
<html>
<head>
<title> Welcome xzz </title>
<h1> welcome to xzz html </h1>
</head>
</html>
```

服务器回发这个 html 文件的内容给单片机

单片机程序如下：

```
struct tcp_pcb *tcppcb = NULL; //必须定义一个TCP控制块

static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb,p->tot_len); //更新接收窗口, 让LWIP内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf,p->payload,p->len); //将接收数据拷贝进缓冲区, 采样p->len长度才正确
        //printf("Recv = %s\r\n",tcp_recvbuf); //打印接收的数据
        MQTT_RECVFLAG = 1;

        debug_printf("Recv = ");
        // for(int i = 0; i< p->len; i++)
        // {
        //     debug_printf(" %x ",tcp_recvbuf[i]); //打印接收的数据
        // }
        debug_printf(" %s ",tcp_recvbuf); //用字符方式打印出http服务器返回的数据
        debug_printf("\r\n");

        memset(p->payload,0,p->len); //清空接收的数据缓冲区, 不是获取p->tot_len的长度, 而是获取p->len长度
        memset(tcp_recvbuf,0,p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务端连接断开, 但是网线连接的TCP服务端没有断开, 而是服务端关闭了TCP软件
    {
        //所以TCP服务端断开连接只有在tcp_recv建立的client_recv回调函数中使用err来判断
        //注意: 就算我拔掉网线, client_recv回调都不会进入err == ERR_OK分支, 而且我重新接上网线了, 会继续传输
        //只有TCP服务端关闭, 服务端会自动回发最后一个断开TCP的请求. 该分支才执行
        debug_printf("server disconnected\r\n");
        tcp_close(tpcb); //当服务器关闭TCP连接的时候, 客户端自身也要清除TCP控制块

        //执行重新连接TCP函数
    }
    return ERR_OK;
}
```

以上是单片机接收 TCP 数据的程序

```

if(tcpcb) //tcpcb不等于NULL, 表示创建成功
{
    IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
    //因为IP控制块里面通过32位存储IP地址)
    err = tcp_connect(tcpcb,&RemoteAddr,8888,tcp_client_connected); //设置远端IP和http服务器端口是80, 我用8888代替
    vTaskDelay(1); //连接客户端TCP之后必须延时, 不然直接死机。神奇

    debug_printf("tcp create success\r\n");
}

tcp_err(tcpcb , client_err);

while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0

HTTP_GET(); //发送http报文给服务器
while(1) //死循环中再次申请UDP, 再次发送
{
    vTaskDelay(100); //延时100ms
}

```

以上是单片机发送 http GET 请求程序

```

/******************
* 单片机GET请求方式
*****************/
void HTTP_GET(void) <----- 目标函数
{
    char key[128] = {0};
    uint8_t TxBuffer[256] = {0};

    sprintf(TxBuffer,"GET /devices/123456/datapoints HTTP/1.1\r\n");
    sprintf(key,"api-key: 89654321\r\n");
    strcat(TxBuffer,key);
    strcat(TxBuffer,"Host:api.xzz.com\r\n\r\n");

    HttpSendData(TxBuffer,strlen(TxBuffer)); //TCP发送数据
}

```

```

#include <stdio.h>
#include <string.h>

#include "stm32f4xx.h" //包含F407整个头文件
#include "lwip/netif.h"
#include "lwip_comm.h"
#include "lwipopts.h"
#include "lwip/pbuf.h" //TCP加入头文件
#include "lwip/tcp.h" //TCP加入头文件

extern struct tcp_pcb *tcpcb;

//使用LWIP RAW API的TCP发送数据
void HttpSendData(uint8_t *pSendBuff,uint32_t len)
{
    tcp_write(tcpcb,pSendBuff,len,1); //TCP发送数据
    tcp_output(tcpcb); //TCP马上发送
}

```

lwip success init.
正在查找DHCP服务器, 请稍等
网卡endMAC地址为: 2 0 0 .78 .0 .48
通过DHCP获取到IP地址: 192.168.0.100
通过DHCP获取到子网掩码: 255.255.255.0
通过DHCP获取到的默认网关: 192.168.0.1
Network task Init...
Task2 Init
connect OK
tcp_create success
Recv = HTTP/1.1 200 OK
Server: PWS/1.0
<!DOCTYPE html>
<html>
<head>
<title> Welcome xzz </title>
<h1> welcome to xzz html </h1>
</head>
</html>
server disconnected

服务器返回的 http
响应和 html 数据

程序运行

```
b'GET /devices/123456/datapoints HTTP/1.1\r\napi-key: 89654321\r\nHost:api.xzz.com\r\n\r\n'
```

服务器收到了数据

详细解析GET请求的数据格式

```
char key[128] = {0};  
uint8_t TxBuffer[256] = {0};  
sprintf(TxBuffer,"GET /devices/123456/datapoints HTTP/1.1\r\n");  
sprintf(key,"api-key: 89654321\r\n");  
strcat(TxBuffer,key);  
strcat(TxBuffer,"Host:api.xzz.com\r\n\r\n");
```

解析如下：

"GET /devices/123456/datapoints HTTP/1.1\r\n"
<method> <URL> 版本
请求头 本来是服务器域名
但是物联网设备也可
以用设备ID来表示

"api-key: 89654321\r\n" 这是GET头后面跟的数据可有可无，根据平台服务器来确定

"Host:api.xzz.com\r\n\r\n" 最后的首部必须要有，首部一般是网址，最后一定是\r\n\r\n结尾
如果尾部只有一个\r\nn. http服务器绝对不会响应

我们使用标准的 http 服务器库再来测试下我单片机的 http 请求代码

```
from http.server import HTTPServer,BaseHTTPRequestHandler  
import json  
  
data = {'result': 'this is a test'}  
  
class Request(BaseHTTPRequestHandler):  
    def do_GET(self):  
        self.send_response(200)  
        self.send_header('Content-type', 'application/json')  
        self.end_headers()  
        self.wfile.write(json.dumps(data).encode())  
  
if __name__ == '__main__':  
    server = HTTPServer(('', 8888), Request)  
    print("Starting server")  
    server.serve_forever()
```

python http 服务器

单片机程序一点都没变

```
/* 单片机GET请求方式  
*****  
void HTTP_GET(void)  
{  
    char key[128] = {0};  
    uint8_t TxBuffer[256] = {0};  
  
    sprintf(TxBuffer,"GET /devices/123456/datapoints HTTP/1.1\r\n");  
    sprintf(key,"api-key: 89654321\r\n");  
    strcat(TxBuffer,key);  
    strcat(TxBuffer,"Host:api.xzz.com\r\n\r\n");  
  
    HttpSendData(TxBuffer,strlen(TxBuffer)); //TCP发送数据  
}  
if(tcppcb) //tcpccb不等于NULL, 表示创建成功  
{  
    IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,  
    //((因为IP控制块里面通过32位存储IP地址)  
    err = tcp_connect(tcpccb,&RemoteAddr,8888,tcp_client_connected); //设置远端IP和http服务器端口是80, 我用8888代替  
    vTaskDelay(1); //连接客户端TCP之后必须延时, 不然直接死机. 神奇  
    debug_printf("tcp create success\r\n");  
}  
  
tcp_err(tcpccb , client_err);  
while(TCP_ConnectFlag == 0); //判断TCP是否连接完成  
{  
    vTaskDelay(100);  
}  
TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0  
HTTP_GET(); //发送http报文给服务器  
  
while(1) //死循环中再次申请UDP, 再次发送  
{  
    vTaskDelay(100); //延时100ms  
}
```

```
root@ubuntu:/home/xzz/pyserver# python3 httpserver.py  
Starting server
```

192.168.0.100 - - [24/Sep/2022 20:26:43] "GET /devices/123456/datapoints HTTP/1.1" 200 - 服务器收到了 GET 请求

```
tcp create success  
Recv = HTTP/1.0 200 OK  
Server: BaseHTTP/0.6 Python/3.5.2  
Date: Sun, 25 Sep 2022 03:26:43 GMT  
Content-type: application/json
```

```
Recv = {"result": "this is a test"}  
server disconnected
```

服务器返回响应数据,第一次返回 http 头信息,第二次返回 JSON

数据。

```
*****  
* 单片机GET请求方式  
*****  
void HTTP_GET(void)  
{  
    char key[128] = {0};  
    uint8_t TxBuffer[256] = {0};  
  
    sprintf(TxBuffer, "ET /devices/123456/datapoints HTTP/1.1\r\n");  
    sprintf(key, "api-key: 89654321\r\n");  
    strcat(TxBuffer, key);  
    strcat(TxBuffer, "Host:api.xzz.com\r\n\r\n");  
  
    HttpSendData(TxBuffer, strlen(TxBuffer)); //TCP发送数据  
}
```

如果我取消掉 GET, 直接 ET 发送会怎么样?

```
192.168.0.100 - - [24/Sep/2022 20:31:27] code 501, message Unsupported method ('ET')  
192.168.0.100 - - [24/Sep/2022 20:31:27] "ET /devices/123456/datapoints HTTP/1.1" 501 -
```

服务器也能收到' ET '发送的数据,但是会认为这是错误的信息,服务器返回 501 错误码,表示服务器错误。

```
tcp create success  
Recv = HTTP/1.0 501 Unsupported method ('ET')  
Server: BaseHTTP/0.6 Python/3.5.2  
Date: Sun, 25 Sep 2022 03:31:27 GMT  
Connection: close  
Content-Type: text/html; charset=utf-8  
Content-Length: 495  
  
Recv = "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
       "http://www.w3.org/TR/html4/strict.dtd">  
<html>  
  <head>  
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
    <title>Error response</title>  
  </head>  
  <body>  
    <h1>Error response</h1>  
    <p>Error code: 501</p>  
    <p>Message: Unsupported method ('ET').</p>
```

单片机也会收到错误乱码

POST 推送, 单片机主动发数据给服务器

```
*****  
* 单片机POST推送方式  
*****  
void HTTP_POST(void)  
{  
    char key[128] = {0};  
    uint8_t TxBuffer[256] = {0};  
  
    sprintf(TxBuffer, "POST /devices/123456/datapoints HTTP/1.1\r\n");  
    sprintf(key, "api-key: 89654321\r\n");  
    strcat(TxBuffer, key);  
    strcat(TxBuffer, "Host:api.xzz.com\r\n\r\n");  
  
    HttpSendData(TxBuffer, strlen(TxBuffer)); //TCP发送数据  
}
```

只需要将 GET 位置改成 POST 字符就可以了, 其余都不动

```

while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0

HTTP_POST(); //推送数据到服务器

```

主程序直接推送数
据到服务器

```

from http.server import HTTPServer,BaseHTTPRequestHandler
import json

data = {'result': 'this is a test'}


class Resquest(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(data).encode())

if __name__ == '__main__':
    server = HTTPServer(('', 8888), Resquest)
    print("Starting server")
    server.serve_forever()

```

python 服务器我们用 GET 来接收 POST 数据

下面启动服务器

```

192.168.0.100 - - [24/Sep/2022 21:53:35] code 501, message Unsupported method ('POST')
192.168.0.100 - - [24/Sep/2022 21:53:35] "POST /devices/123456/datapoints HTTP/1.1" 501 -

```

服务器发现接收的是 POST 数据，直接返回 501，数据格式 错误

```

Recv = HTTP/1.0 501 Unsupported method ('POST')
Server: BaseHTTP/0.6 Python/3.5.2
Date: Sun, 25 Sep 2022 04:53:35 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 497

```

```

Recv = <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    ...

```

单片机也收到了 501，错误信息

这是怎么回事呢？这是因为你 http 服务器除了 GET，必须单独实现 POST 函数才行。

```

class Resquest(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(data).encode())

    def do_POST(self):
        path = self.path
        print(path) # 获取单片机POST来的路径

if __name__ == '__main__':
    server = HTTPServer(('', 8888), Resquest)
    print("Starting server")
    server.serve_forever()

```

python 实现 do POST

```

root@ubuntu:/home/xzz/pyserver# python3 httpserver.py
Starting server

```

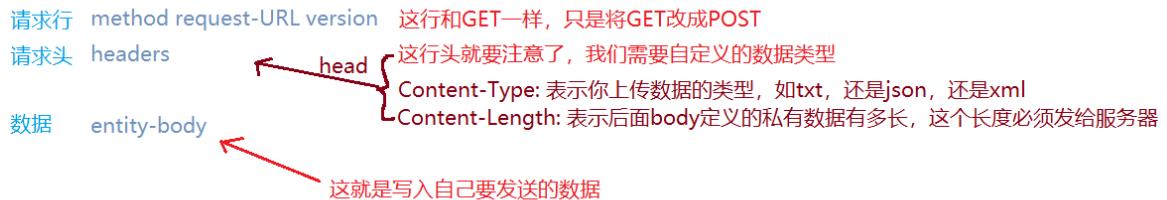
接收单片机 POST
数据成功

connect OK
tcp create success
server disconnected

单片机也是 POST 发送成
功，然后主动断开 http 服
务器，没有 501 返回

但是单片机的实时数据如何 POST 到服务端呢？这里就要注意数据格式了，**其实 POST 比 GET 要多一个 Content-Length 字段，这是必须的，前面我没有讲。**

POST数据格式的定义



例子如下：

```
# 请求行
POST /devices/123456/datapoints HTTP/1.1
# 下面都是请求头
Host:api.xzz.com
Content-Length: 8
Content-Type: application/x-www-form-urlencoded; charset=UTF-8 数据类型也可以不要
# 下面是消息主体内容
8个数据
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08
```

```
*****
* 单片机POST推送方式
*****
void HTTP_POST(void)
{
    char key[128] = {0};
    uint8_t TxBuffer[256] = {0};
    uint8_t dataBuff[128] = {0};

    sprintf(TxBuffer, "POST /devices/123456/datapoints HTTP/1.1\r\n"); //报文首行
    sprintf(key, "api-key: 89654321\r\n"); //可以有,也可以没有
    strcat(TxBuffer, key);
    strcat(TxBuffer, "Host:api.xzz.com\r\n"); //追加报文

    sprintf(dataBuff, "(\"temp\":%d,\"humi\":%d)", 10, 20); //将自己私有的数据放入
    sprintf(key, "Content-Length:%d\r\n\r\n", strlen(dataBuff)); //计算私有数据长度, 构建长度报文头

    strcat(TxBuffer, key); //先将http文本头放入
    memset(key, 0, sizeof(key)); //清0缓存
    sprintf(key, "%s\r\n\r\n", dataBuff);

    strcat(TxBuffer, key); //再放入数据
    HttpSendData(TxBuffer, strlen(TxBuffer)); //TCP发送数据
}
```

"POST /devices/123456/datapoints HTTP/1.1\r\n" 首行
"api-key: 89654321\r\n"
"Host:api.xzz.com\r\n"
"Content-Length:%d\r\n\r\n" } 这些都是请求头，主要要加入Content-Length:%d数据长度
"(\\"temp\\":%d,\\"humi\\":%d)" 这是要发送的数据

```
while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0

HTTP_POST(); //推送数据到服务器

while(1) //死循环中再次申请UDP, 再次发送
{
    vTaskDelay(300); //延时100ms
}
```

主程序 POST 一次数据到服务器

```

class Resquest(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(data).encode())

    def do_POST(self):
        path = self.path
        print(path) # 获取单片机POST来的路径
        content_length = int(self.headers.get('Content-Length')) #提取Content-Length里面的数据长度,
                                                                #这个长度是单片机发过来的
        print(content_length)
        postData = self.rfile.read(content_length) ##读取entity-body里面自己写的私有数据，这个数据长度和Content-Length>
计算的必须一致
        print(postData)

if __name__ == '__main__':
    server = HTTPServer(('', 8888), Resquest)
    print("Starting server")
    server.serve_forever()

```

Python 服务器 POST 添加内容

```

root@ubuntu:/home/xzz/pyserver# python3 httpserver.py
Starting server

/devices/123456/datapoints
21
b'{"temp":10,"humi":20}'

```

单片机发送一帧 POST 到服务器

这里有个问题，就是 POST 无法死循环的发送，因为执行一次 POST 之后，服务端就会断开 TCP 连接。

RAW API MQTT 实现

MQTT 基础查阅《pythonMQTT 开发手册》，这里只讲数据格式和协议实现。

报文	描述	流向	值	固定报头	可变报头	负载
CONNECT	客户端请求与服务器连接	客户端 -> 服务器	1	有	有	有
CONNACK	MQTT 服务端确认连接建立	服务端 -> 客户端	2	有	有	有
PUBLISH	发布消息	客户端 <-> 服务器	3	有	有	有
PUBACK	收到发布消息确认(QOS1)	客户端 <-> 服务器	4	有	有	无
PUBREC	发布消息收到(QOS2)	客户端 <-> 服务器	5	有	有	无
PUBREL	发布消息释放(QOS2)	客户端 <-> 服务器	6	有	有	无
PUBCOMP	发布消息完成(QOS2)	客户端 <-> 服务器	7	有	有	无
SUBSCRIBE	订阅请求	客户端 -> 服务端	8	有	有	有
SUBACK	订阅确认	服务端 -> 客户端	9	有	有	有
UNSUBSCRIBE	取消订阅	客户端 -> 服务端	10	有	有	有
UNSUBACK	取消订阅确认	服务端 -> 客户端	11	有	有	无
PING	客户端发送心跳包给服务端(长连接)	C->S	12	有	无	无
PINGRSP	PING 命令回复	服务端 -> 客户端	13	有	无	无
DISCONNECT	断开连接	客户端 -> 服务端	14	有	无	无

MQTT就是实现以上这14个报文数据帧

客户端主动发数据到服务器
要求断开连接

意思就是我的客户端设备发送请求连接MQTT服务器
意思就是服务器收到客户端连接请求，客户端连接服务器成功，服务器返回连接成功信息给客户端（比如客户端发送的密码，服务器查询看是否正确，如果正确，服务器返回正确信息给客户端）

比如客户端后面1小时不发数据给MQTT服务端，如果是阿里云的MQTT服务端，发现一直收不到数据，就会断开与客户端的MQTT连接。但是我客户端不想和服务器断开连接，那么久只有法固定长度的心跳包，让服务器知道我客户端还在持续的发数据。这样服务端就不会主动断开连接。

服务端收到客户端ping数据之后，服务端并没有收到有价值的数据，但是服务端又不能主动断开客户端，因为客户端已经发了长连接ping了，所以服务端只有回复一个ping到客户端

connect 报文实现

报文	描述	流向	值	固定报头	可变报头	负载
CONNECT	客户端请求与服务器连接	客户端 -> 服务器	1	有	有	有

图例 3.1 –CONNECT 报文的固定报头 版本3.1.1

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT 报文类型 (1)				Reserved 保留位			
这就是发送的第1字节0x10	0	0	0	1	0	0	0	0
byte 2...	剩余长度值							

紧接着 byte2 字节的剩余长度就是可变数据帧+负载数据帧

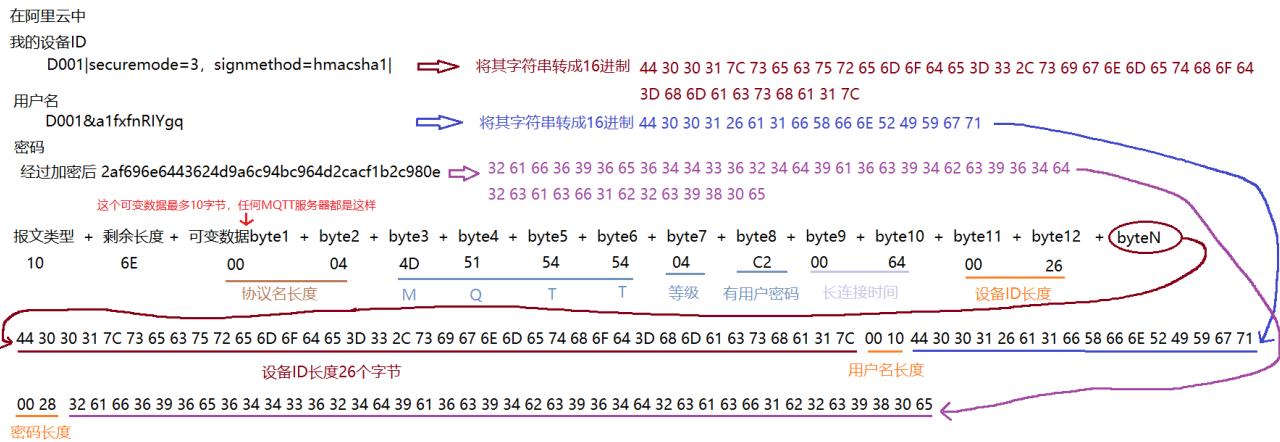
例如： 报文类型 + 可变数据帧 + 负载

0x10 10个字节 10个字节

那么剩余长度byte2，除去报文类型0x10，就剩下20字节的剩余长度

所以结果就是 0x10 + 0x14 + 可变数据帧， + 负载

byte1 byte2



最简单的 MQTT 连接服务器 CONNECT 程序

```
//使用LWIP RAW API的TCP发送数据
void MQTTSendData(uint8_t *pSendBuff,uint32_t len)
{
    tcp_write(tcpcb,pSendBuff,len,1); //TCP发送数据
    tcp_output(tcpcb); //TCP马上发送
}

/*
 *函数名: 连接服务器报文
 *参数: 无
 *返回值: 无
 */
void MQTT_ConectPack(void)
{
    uint8_t ConnectMsg[] = {0X10,0X12,0X00,0X04,0X4D,0X51,0X54,0X54,0X04,0X02,0X00,0X3C,0X00,0X06,0X31,0X32,0X33,0X34,0X35,0X36};
    MQTTSendData(ConnectMsg,sizeof(ConnectMsg));
}
```

```
struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
err_t err; //tcp函数执行后返回状态

debug_printf("Task2 Init\r\n");

tcpcb = tcp_new();

if(tcpcb) //udppcb不等于NULL, 表示创建成功
{

    IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
                                            //((因为IP控制块里面通过32位存储IP地址)
    err = tcp_connect(tcpcb,&RemoteAddr,1883,tcp_client_connected); //设置远端IP和端口8888
    vTaskDelay(1); //连接客户端TCP之后必须延时, 不然直接死机。神奇

    debug_printf("tcp create success\r\n");
}

tcp_err(tcpcb , client_err);

MQTT_ConectPack();
vTaskDelay(1);
```

等到服务器启动后运行该程序。这个连接程序单独测试没有问题, 但是和 publish 一起使用就会有问题, 还是和 TCP 连接之后的判定有关, 在 publish 章节我会讲解。

```
/etc/mosquitto# mosquitto -c mosquitto.conf
mosquitto version 1.6.8 starting
Config loaded from mosquitto.conf.
Opening ipv4 listen socket on port 1883.
Opening ipv6 listen socket on port 1883.
New connection from 192.168.0.103 on port 1883.
Client <unknown> disconnected due to protocol error.ubuntu服务器启动 MQTT
```

```
1662950430: New connection from 192.168.0.100 on port 1883.
1662950430: New client connected from 192.168.0.100 as 123456 (p2, c1, k60).
```

客户端连接成功。

client_recv 接收服务器发来的 TCP 数据

```
static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p != NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recv(p, p->tot_len); //更新接收窗口,让LWIP内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf, p->payload, p->len); //将接收数据拷贝进缓冲区, 采样p->len长度才正确
        //printf("Recv = %s\r\n", tcp_recvbuf); //打印接收的数据
        for(int i = 0; i < p->len; i++)
        {
            printf("Recv = %x\r\n", tcp_recvbuf[i]); //打印接收的数据
        }

        memset(p->payload, 0, p->len); //清空接收的数据缓冲区, 不是获取p->tot_len的长度, 而是获取p->len长度
        memset(tcp_recvbuf, 0, p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务端连接断开, 但是网线连接的TCP服务端没有断开, 而是服务端关闭了TCP软件
    {
        //所以TCP服务端断开连接只有在tcp_recv建立的client_recv回调函数中使用err来判断
        //注意: 就算我拔掉网线, client_recv回调都不会进入err == ERR_OK分支, 而且我重新接上网线了, 会继续传输
        // 只有TCP服务端关闭, 服务端会自动回发最后一个断开TCP的请求. 该分支才执行
        printf("server disconnected\r\n");
        tcp_close(tpcb); //当服务器关闭TCP连接的时候, 客户端自身也要清除TCP控制块

        //执行重新连接TCP函数
    }
    return ERR_OK;
}
Recv = 20
Recv = 2
Recv = 0
Recv = 0      客户端连接 MQTT 服务器成功, 会返回 0x20 02 00 00
```

如果在客户端的 MQTT 没有断开的情况下, 在执行一次连接, 就会出现如下情况:

```
1662950983: New connection from 192.168.0.100 on port 1883.
1662950983: New client connected from 192.168.0.100 as 123456 (p2, c1, k60).
1662951008: Socket error on client 123456, disconnecting.
```

这就是表示客户端连接设备 ID 123456 失败, 因为上一次 MQTT 连接没有断开。

遇到这种情况, 第 1 个方式是等待时间超时, 服务器主动断开

```
1662951107: Client <unknown> has exceeded timeout, disconnecting.
server disconnected      这时候客户端也会打印断开
```

第 2 个方式就是主动断开。

```

*****
*MQTT 断开报文
*****
void MQTT_Disconnect(void)
{
    uint8_t DISconnectArray[] = {0xE0,0x00};
    MQTTSendData(DISconnectArray,sizeof(DISconnectArray));
}

```

断开报文下面内容有协议讲解, 这里只是记到 0xE 00 就是。

主函数, 发送 5 次 publish, 就主动断开与服务器连接

```

while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0
MQTT_ConnectPack(); //连接MQTT服务器

uint8_t Count = 0;
while(1) //死循环中再次申请UDP, 再次发送
{
    if(MQTT_RECVFLAG == 1)
    {
        Count++;
        MQTT_Publish();
    }
    if(Count == 5)
    {
        MQTT_Disconnect();
    }
    vTaskDelay(100); //延时100ms
}

```

```

xxxxxx
lwip success init.
正在查找DHCP服务器, 请稍等.....
网卡en的MAC地址为..... 2.0.0.78.0.48
通过DHCP获取到IP地址..... 192.168.0.103
通过DHCP获取到子网掩码..... 255.255.255.0
通过DHCP获取到的默认网关..... 192.168.0.1
Network_task Init...
Task2 Init
connect OK
tcp create success
Recv = 20 2 0 0
Recv = 40 2 0 1
server disconnected

```

连接 MQTT 服务器

发 5 次 publish
数据到服务器

设备主动断开
MQTT 连接

```

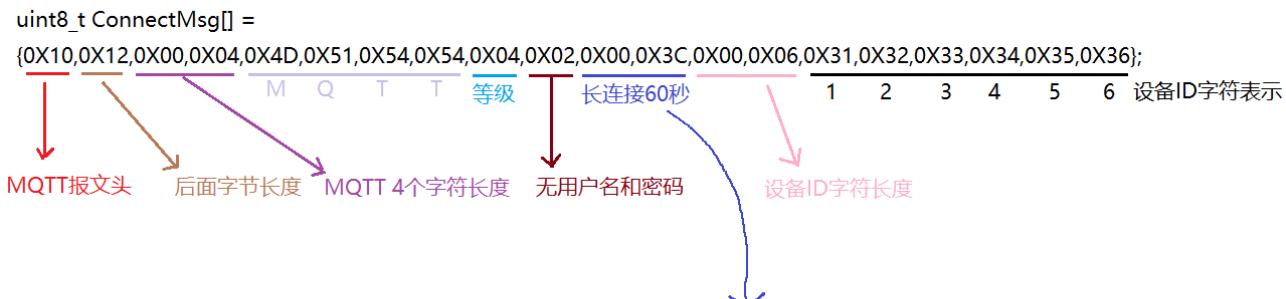
1663475047: New connection from 192.168.0.103 on port 1883.
1663475047: New client connected from 192.168.0.103 as 123456 (p2, c1, k60).
1663475048: Client 123456 disconnected.

```

对本次发送的连接数据进行解析

根据测试发现, 只发送报文类型 + 协议名 + 无用户名和密码 + 长连接是不行的
必须是报文类型 + 协议名 + 无用户名和密码 + 设备ID, 这样才能成功连接MQTT服务器

这就证明了我写的python MQTT客户端是系统默认加入了设备ID的, 不然怎么这么容易成功



1662951107: Client <unknown> has exceeded timeout, disconnecting.

60秒之类, 客户端没有数据发到服务器, 服务器主动断开指定设备ID的连接

这就是我之前说的, 等待60秒服务器断开之后, 同一个设备ID可以再次连接MQTT服务器

CONNACK 客户端再次确认是否和服务器一直存在 MQTT 连接

报文	描述	流向	值	固定报头	可变报头	负载
CONNACK	MQTT服务端确认连接建立	服务端->客户端	2	有	有	有

这个 CONNACK 报文就是我客户端连接 MQTT 服务器之后, 服务器返回的 0x20 02 00 00 。

PUBLISH 发送报文

报文	描述	流向	值	固定报头	可变报头	负载
PUBLISH	发布消息	客户端 <-> 服务器	3	有	有	有

PUBLISH 发布消息(先看看收到服务器发布的消息数据)

图例 3.10 – PUBLISH 报文固定报头

DUP为1表示支持重发模式，我们现在不需要

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT 控制报文类型 (3)				DUP	QoS 等级		RETAIN
	0	0	1	1	X	X	X	X
byte 2	0	0	1	1	剩余长度 0	0	0	0

如果没有重发模式，
QOS就是0

= 0x30



PUBLISH 程序测试

我这里先讲解下 TCP 的连接延时不准确，造成 tcp_write 发数据时好时不好

```
static uint8_t MQTT_RECVFLAG = 0; //判断MQTT是否连接完成
static uint8_t TCP_ConnectFlag = 0; //判断TCP是否连接完成
err_t tcp_client_connected(void*arg, struct tcp_pcb *tpcb, err_t err)
{
    debug_printf("connect OK\r\n");
    tcp_recv(tpcpcb,client_recv); //注册接收回调函数，用于接收TCP数据
    TCP_ConnectFlag = 1; //TCP是否链接完成，判断标志位
}
tcppcb = tcp_new();

if(tpcpcb) //udppcb不等于NULL，表示创建成功
{
    IP4_ADDR(&RemoteAddr,192,168,0,112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量，  
//((因为IP控制块里面通过32位存储IP地址)
    err = tcp_connect(tpcpcb,&RemoteAddr,1883,tcp_client_connected); //设置远端IP和端口8888
    vTaskDelay(1); //连接客户端TCP之后必须延时，不然直接死机。神奇
    debug_printf("tcp create success\r\n");
}

tcp_err(tpcpcb , client_err);

while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0
MQTT_ConnectPack(); //连接MQTT服务器
```

然后给全局变量置1，来证明TCP
已经连接成功，下面才能用
tcp_write 发数据

判断 TCP 连接成功之后，退出线程
死循环，进入 MQTT 服务器连接，
也就是执行 tcp_write

```

while(1) //死循环中再次申请UDP，再次发送
{
    if(MQTT_RECVFLAG == 1)
    {
        MQTT_Publish();
    }
    vTaskDelay(100); //延时10ms
}

```

我们知道连接 MQTT 服务器之后，服务器会返回 0x20 02 00 00
我在 TCP 接收函数中获取服务器回复状态，确认无误，再执行 publish 里
面的 tcp_write

```

//使用LWIP RAW API的TCP发送数据
void MQTTSendData(uint8_t *pSendBuff,uint32_t len)
{
    tcp_write(tcpcb,pSendBuff,len,1); //TCP发送数据
    tcp_output(tcpcb); //TCP马上发送
}

/*****************
* 发送主题和数据
********************/
void MQTT_Publish(void)
{
    uint8_t PublishMsg[] = {0X30,0X0C,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X31,0X32,0X33};
    MQTTSendData(PublishMsg,sizeof(PublishMsg));
}

```

下面对 publish 的数据包进行解释

uint8_t PublishMsg[] = {0X30,0X0C,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X31,0X32,0X33};

这是QOS0 (等级0)的publis报文发送格式

```

def on_connect(client,userdata,flags,rc):
    if rc == 0:
        print("connect")
        print("conneccted with result code" + str(rc))

def on_message(client,userdata,msg):
    print(msg.topic + " " + str(msg.payload))
    client.publish("serverrecv","server123456",1)

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host = "192.168.0.112",port = 1883, keepalive = 60)

client.subscribe('xzzmqtt',qos = 1)
client.loop_start()

while True:
    pass

```

这个就是服务端订阅的主
题，是不是和你前面的 lwip
实现的 publish 很像

MQTT 服务器程序

```

1662975394: New connection from 192.168.0.100 on port 1883.
1662975394: New client connected from 192.168.0.100 as 123456 (p2, c1, k60).

```

客户端连接上 MQTT 服务器

```

xzzmqtt b'123'

```

服务端数据接收成功。

Publish QOS1 等级 1 测试

QOS1等级的publis和QOS0不一样，需要在主题后面加入报文ID

```
*****
* 发送主题和数据
*****
void MQTT_Publish(void)
{
    // uint8_t PublishMsg[] = {0X30,0X0C,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X31,0X32,0X33}; //QOS0
    uint8_t PublishMsg[] = {0X32,0X0E,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X00,0X01,0X31,0X32,0X33}; //QOS1
    MQTTSendData(PublishMsg,sizeof(PublishMsg));
}
```

下面进行代码分析

uint8_t PublishMsg[] = {0X32,0X0E,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X00,0X01,0X31,0X32,0X33}; //QOS1

增加了2字节报文ID
主题长度

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT 控制报文类型 (3)				DUP	QoS 等级		RETAIN
	0	0	1	1	X	X	X	
byte 2	0	0	1	1	剩余长度	0	1	0

=0x32

报文ID 数据

这报文ID就是我们 QOS1需要定义的，必须2字节
ID序号自己随便定义就是了

服务器返回: 0x40 02 00 01 02表示后面跟的字节长度

服务器返回0x40表示服务器收到数据了 00 01 就是你设备发的报文ID，服务器原封不动返回

下面贴出代码示例

```
static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb,p->tot_len); //更新接收窗口,让LWIP内核指定我在回调函数中接收到数据了,
                                         //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf,p->payload,p->len); //将接收数据拷贝进缓冲区, 采样p->len长度才正确
        //printf("Recv = %s\r\n",tcp_recvbuf); //打印接收的数据
        MQTT_RECVFLAG = 1;
        for(int i = 0; i< p->len; i++)
        {
            printf("Recv = %x\r\n",tcp_recvbuf[i]); //打印接收的数据
        }

        memset(p->payload,0,p->len); //清空接收的数据缓冲区, 不是获取p->tot_len的长度, 而是获取p->len长度
        memset(tcp_recvbuf,0,p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务端连接断开, 但是网线连接的TCP服务端没有断开, 而是服务端关闭了TCP软件
    {
        //所以TCP服务端断开连接只有在tcp_recv建立的client_recv回调函数中使用err来判断
        //注意: 就算我拔掉网线, client_recv回调都不会进入err == ERR_OK分支, 而且我重新接上网线了, 会继续传输
        // 只有TCP服务器关闭, 服务端会自动回发最后一个断开TCP的请求. 该分支才执行
        printf("server disconnected\r\n");
        tcp_close(tpcb); //当服务器关闭TCP连接的时候, 客户端自身也要清除TCP控制块

        //执行重新连接TCP函数
    }
    return ERR_OK;
}
```

接收服务器返回数据
就是上面的 TCP 接收函数

```

err_t tcp_client_connected(void *arg, struct tcp_pcb *tcp_pcb, err_t err)
{
    debug_printf("connect OK\r\n");
    tcp_recv(tcp_pcb, client_recv); //注册接收回调函数，用于接收TCP数据

    TCP_ConnectFlag = 1; //TCP是否链接完成，判断标志位
}

static void client_err(void *arg, err_t err)
{
    tcp_close(tcp_pcb);
}

void Task2(void *pvParameters)
{
    uint8_t TCP_sendData[] = "12345678"; //这就是TCP要发送的数据

    struct ip_addr RemoteAddr; //保存远端服务器IP地址的变量
    err_t err; //tcp函数执行后返回状态

    debug_printf("Task2 Init\r\n");

    tcppcb = tcp_new();

    if(tcppcb) //udppcb不等于NULL，表示创建成功
    {
        IP4_ADDR(&RemoteAddr, 192, 168, 0, 112); //将点分10进制IP转为4字节变量然后将IP赋值给ip_addr定义的变量,
                                                //((因为IP控制块里面通过32位存储IP地址)
        err = tcp_connect(tcppcb, &RemoteAddr, 1883, tcp_client_connected); //设置远端IP和端口8888
        vTaskDelay(1); //连接客户端TCP之后必须延时，不然直接死机。神奇

        debug_printf("tcp create success\r\n");
    }

    tcp_err(tcppcb, client_err);

    while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
    {
        vTaskDelay(100);
    }

    TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0
    MQTT_ConectPack(); //连接MQTT服务器

    while(1) //死循环中再次申请UDP，再次发送
    {
        if(MQTT_RECVFLAG == 1)
        {
            MQTT_Publish();
        }
        vTaskDelay(100); //延时100ms
    }
}

```

确保 TCP 连接成功

```

*****发送主题和数据*****
void MQTT_Publish(void)
{
    // uint8_t PublishMsg[] = {0X30,0X0C,0X00,0X07,0X78,0X7A,0X71,0X74,0X74,0X31,0X32,0X33}; //QOS0
    uint8_t PublishMsg[] = {0X32,0X0E,0X00,0X07,0X78,0X7A,0X71,0X74,0X74,0X00,0X01,0X31,0X32,0X33}; //QOS1
    MQTTSendData(PublishMsg, sizeof(PublishMsg));
}

```

Recv = 40
Recv = 2
Recv = 0
Recv = 1

服务器返回 0x40 02 00 01 正确。看下面章节如何使用订阅接收服务器的数据。

报文	描述	流向	值	固定报头	可变报头	负载
PUBACK	收到发布消息确认(QOS1)	客户端 <-> 服务器	4	有	有	无
PUBREC	发布消息收到(QOS2)	客户端 <-> 服务器	5	有	有	无
PUBREL	发布消息释放(QOS2)	客户端 <-> 服务器	6	有	有	无
PUBCOMP	发布消息完成(QOS2)	客户端 <-> 服务器	7	有	有	无
SUBSCRIBE	订阅请求	客户端 -> 服务端	8	有	有	有

SUBSCRIBE使用

有些云服务器可以在网页端直接订阅主题，比如阿里云，所以就不需要客户端订阅了
有些服务器，比如自己私有服务器，就需要客户端发送SUBSCRIBE订阅，来订阅主题

图例 3.20 – SUBSCRIBE 报文固定报头

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT 控制报文类型 (8)						保留位	
	1	0	0	0	0	0	1	0
byte 2	剩余长度							

报文头 + byte1 + byte2 + byte3 + byte4 + byte5 + byte6 + byte7
发送订阅 0x82 00 0A 后面主题字符的长度 78 7A 7A
报文标识符：报文头 + byte0 后面主题字符的长度 78 7A 7A
后面字符串额16进制，这个xzz就是我要订阅的主题，千万不要有空格哦，不然空格也算字符。

报文标识符：就是这一个报文的ID，这个ID就是为了让你知道你报文是否发送成功。如果报文发送失败，
服务器会返回报文ID，让你知道是哪条报文失败了。(00 0A 表示订阅报文)

报文头 + byte0 byte1 + byte2 + byte3 + byte4 + byte5 + byte6 + byte7
发送订阅 0x82 07 00 0A 后面主题字符的长度 78 7A 7A 这个报文默认是QOS等级0
后面字符串长度

如果要想报文有等级1

报文头 + byte0 byte1 + byte2 + byte3 + byte4 + byte5 + byte6 + byte7 在报文最后面加入01
发送订阅 0x82 07 00 0A 后面主题字符的长度 78 7A 7A 01 就表示该报文为等级1
后面字符串长度

服务器返回 90 03 00 0A 01

表示服务器收到订阅报文

最后这个字节01，表示你发送的报文等级，如果你发送的报文QOS是01，那么服务器
也回复01。(注意 阿里云除外，阿里云不管你发送的QOS等级是0或者1，都统一回复01)

只有你客户端主动去向 MQTT 服务器中间件订阅了报文之后，MQTT 服务器的客户端才能向你设备发数据。

```
def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))
    client.publish("serverrecv", "server123456", 1)
```

这就是我 python 服务器，接收到客户端数据之后，又向主题“serverrecv”发数据。但是我客户端之前是收不到这个“serverrecv”主题的数据。这是因为客户端没有主动订阅“serverrecv”主题。

本地设备端订阅 MQTT 主题，接收服务器发来的数据

```
/*
 * 订阅主题
 */
void MQTT_SubSubscribe(void)
{
    uint8_t SubSubscribe[] = {0X82,0X0F,0X00,0X0A,0X00,0X0A, 0X73,0X65,0X72,0X76,0X65,0X72,0X72,0X65,0X63,0X76,0X00};
    MQTTSendData(SubSubscribe, sizeof(SubSubscribe));
}
```

uint8_t SubSubscribe[] = {0X82,0X0F,0X00,0X0A,0X00,0X0A, 0X73,0X65,0X72,0X76,0X65,0X72,0X72,0X65,0X63,0X76,0X00};
订阅报头 订阅报文ID 订阅的主题字符长度 主题字符 该主题QOS为0
后面数据长度

我们尝试将以上订阅报文发给服务器

```
/*
 * 函数名: 连接服务器报文
 * 参 数: 无
 * 返回值: 无
 */
void MQTT_ConnectPack(void)
{
    uint8_t ConnectMsg[] = {0X10,0X12,0X00,0X04,0X4D,0X51,0X54,0X54,0X04,0X02,0X00,0X3C,0X00,0X06,0X31,0X32,0X33,0X34,0X35,0X36};
    MQTTSendData(ConnectMsg,sizeof(ConnectMsg));
}

/*********************发送主题和数据********************/
void MQTT_Publish(void)
{
    uint8_t PublishMsg[] = {0X32,0X0E,0X00,0X07,0X78,0X7A,0X7A,0X6D,0X71,0X74,0X74,0X00,0X01,0X31,0X32,0X33}; //QOS1
    MQTTSendData(PublishMsg,sizeof(PublishMsg));
}

/*********************MQTT 断开报文********************/
void MQTT_Disconnect(void)
{
    uint8_t DISconnectArray[] = {0xE0,0x00};
    MQTTSendData(DISconnectArray,sizeof(DISconnectArray));
}

static err_t client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p!= NULL) //表示pbuf接收到服务器的数据
    {
        tcp_recved(tpcb,p->tot_len); //更新接收窗口,让LWIP内核指定我在回调函数中接收到数据了,
        //不然内核会重复发送收到的数据包给我tcp_recv事件

        memcpy(tcp_recvbuf,p->payload,p->len); //将接收数据拷贝进缓冲区,采样p->len长度才正确
        //printf("Recv = %s\r\n",tcp_recvbuf); //打印接收的数据
        MQTT_RECVFLAG = 1;

        debug_printf("Recv = ");
        for(int i = 0; i < p->len; i++)
        {
            debug_printf(" %x ",tcp_recvbuf[i]); //打印接收的数据
        }
        debug_printf("\r\n");

        memset(p->payload,0,p->len); //清空接收的数据缓冲区,不是获取p->tot_len的长度,而是获取p->len长度
        memset(tcp_recvbuf,0,p->len); //清空接收的数据缓冲区, p->tot_len就是接收的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK) //表示TCP服务端连接断开,但是网线连接的TCP服务端没有断开,而是服务端关闭了TCP软件
    {
        //所以TCP服务端断开连接只有在tcp_recv建立的client_recv回调函数中使用err来判断
        //注意:就算我拔掉网线,client_recv回调都不会进入err == ERR_OK分支,而且我重新接上网线了,会继续传输
        //只有TCP服务器关闭,服务端会自动回发最后一个断开TCP的请求.该分支才执行
        debug_printf("server disconnected\r\n");
        tcp_close(tpcb); //当服务器关闭TCP连接的时候,客户端自身也要清除TCP控制块

        //执行重新连接TCP函数
    }
    return ERR_OK;
}

while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}

TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0
MQTT_ConnectPack(); //连接MQTT服务器
MQTT_SubScribe(); //向MQTT中间服务器订阅主题

while(1) //死循环中再次申请UDP,再次发送
{
    if(MQTT_RECVFLAG == 1)
    {
        MQTT_Publish();
    }
    vTaskDelay(100); //延时100ms
}
```

向服务器发送数据,这样服务器才能发 publish 给我的设备,因为 IOT 服务端是阻塞返回的

```
def on_message(client,userdata,msg):
    print(msg.topic + " " + str(msg.payload))
    client.publish("serverrecv","server123456",1)
```

现在测试下,我的设备端 TCP 接收函数,能接收服务器 publish 来的数据

连接 MQTT 服务器之后,设备端一定要先订阅主题,这样服务器 publish 到设备端的数据才能接收到

服务端想主动给客户端发数据,那么服务端 publish 主题必须和设备端订阅的一样。

```

正在查找DHCP服务器,请稍等.....
网卡en的MAC地址为:.....2.0.0.78.0.48
通过DHCP获取到IP地址.....192.168.0.103
通过DHCP获取到子网掩码.....255.255.255.0
通过DHCP获取到的默认网关.....192.168.0.1
NetWork_task Init...
Task2 Init
tcp create success
connect OK
Recv = 20 2 0 0 ← 返回连接MQTT服务器成功
Recv = 90 3 0 a 0 ← 返回订阅成功
Recv = 40 2 0 1 ← PUBLISH QOS1等级的数据到服务端, 服务端返回
Recv = 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36
    ↓
    主题长度      服务器发的主题字符      服务器发来的数据
    后面数据长度

服务器向设备发的是QOS0的数据, 所以是30开头
Recv = 40 2 0 1
Recv = 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36
Recv = 40 2 0 1 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36
Recv = 40 2 0 1 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36
Recv = 40 2 0 1 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

```

设备端收到服务器主动 publish 的数据, 正常无误。

订阅确认

报文	描述	流向	值	固定报头	可变报头	负载
SUBACK	订阅确认	服务端->客户端	9	有	有	有

就是上一节讲的设备端订阅报文, 服务端返回的 0x90 03 00 0A 00 就是 SUBACK 报文

取消订阅

报文	描述	流向	值	固定报头	可变报头	负载
UNSUBSCRIBE	取消订阅	客户端->服务端	10	有	有	有

图例 3.28 – UNSUBSCRIBE 报文固定报头

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT 控制报文类型 (10)				保留位			
	1	0	1	0	0	0	1	0
byte 2	剩余长度							

报文头 + byte0 byte1 + byte2 + byte3 + byte4 + byte5 + byte6 + byte7

发送取消订阅 0xA2 07 00 0A 后面主题字符的长度 78 7A 7A 01
 取消订阅和订阅基本都是一样的字段, 只是报文头和报文尾不一样
 服务器返回 B0 02 00 0A 取消订阅没有QOS这个字节

下面进行取消订阅代码示例。

```
/*********************  
* 取消订阅确认  
* UNSUBSCRIBE  
*****  
void MQTT_UNSUBSCRIBE(void)  
{  
    uint8_t UNSubscribe[] = {0XA2,0X0E,0X00,0X0A,0X00,0X0A,0X73,0X65,0X72,0X76,0X65,0X72,0X65,0X63,0X76};  
    MQTTSendData(UNSubscribe,sizeof(UNSubscribe));  
}
```

取消订阅报文解析

```
uint8_t UNSubscribe[] = {0XA2,0X0E,0X00,0X0A,0X00,0X0A,0X73,0X65,0X72,0X76,0X65,0X72,0X65,0X63,0X76};
```

取消订阅报头 后面主题字符长度 主题字符 取消订阅不需要发QOS等级

后面字节长度 报文标识符，记住取消订阅报文标识符必须和订阅主题的报文标识符一样

所以要记住订阅主题的报文标识符

```
while(TCP_ConnectFlag == 0); //判断TCP是否连接完成  
{  
    vTaskDelay(100);  
}  
  
TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0  
MQTT_ConnectPack(); //连接MQTT服务器  
  
MQTT_SubSubscribe(); //向MQTT中间服务器订阅主题  
  
uint8_t Count = 0;  
while(1) //死循环中再次申请UDP，再次发送  
{  
  
    if(MQTT_RECVFLAG == 1)  
    {  
        Count++;  
        MQTT_Publish(); //发布数据  
    }  
    if(Count == 5)  
    {  
        MQTT_UNSUBSCRIBE(); //取消订阅  
    }  
    vTaskDelay(100); //延时100ms  
}
```

Recv = 20 2 0 0 ← 链接报文成功

Recv = 90 3 0 a 0 订阅主题成功

Recv = 40 2 0 1

Recv = [30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36] 设备发送5次publish

Recv = 40 2 0 1

Recv = [30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36]

Recv = 40 2 0 1 [30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36]

Recv = 40 2 0 1

Recv = [30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36]

Recv = 40 2 0 1

Recv = b0 2 0 a 取消订阅成功，返回服务器UNSUBACK报文

Recv = 40 2 0 1

我不知道主题取消之后怎么还可以发送该主题的报文

报文	描述	流向	值	固定报头	可变报头	负载
UNSUBACK	取消订阅确认	服务端->客户端	11	有	有	无

取消订阅就是服务器返回的 0xb0 02 00 0a，上面讲了的。

MQTT ping 包

报文	描述	流向	值	固定报头	可变报头	负载
PING	客户端发送心跳包给服务端(长连接)	C->S	12	有	无	无

C0 00 就是ping包

发送连接报文后 + C0 00

10 6E 00 04 4D 51 54 54 04 C2 00 64 00 26 44 发送完链接报文之后

再发送ping C0 00

服务器返回 D0 00

有时候可能网络环境不好，所以要多发几个ping包来确定

 发送 C0 00

 服务器返回D0 00

 发送 C0 00

 服务器返回D0 00

 发送 C0 00

 服务器返回D0 00

```
*****
* MQTT ping 服务端，确保MQTT还在连接中
*****
void MQTT_Ping(void)
{
    uint8_t ping[] = {0XC0, 0X00};
    MQTTSendData(ping, sizeof(ping));
}
```

```
while(TCP_ConnectFlag == 0); //判断TCP是否连接完成
{
    vTaskDelay(100);
}
```

```
TCP_ConnectFlag = 0; //TCP连接成功之后标志位清0
MQTT_ConnectPack(); //连接MQTT服务器
```

```
MQTT_SubSubscribe(); //向MQTT中间服务器订阅主题
```

```
while(1) //死循环中再次申请UDP，再次发送
{
```

```
    if(MQTT_RECVFLAG == 1)
    {
        Count++;
        MQTT_Publish(); //发布数据
        MQTT_Ping(); //发送ping包，确保和服务器连接正常
    }
}
```

```
vTaskDelay(100); //延时100ms
```

每次循环都要发送
ping 包

正在查找DHCP服务器,请稍等.....

网卡en的MAC地址为:.....2.0.0.78.0.48

通过DHCP获取到IP地址.....192.168.0.101

通过DHCP获取到子网掩码.....255.255.255.0

通过DHCP获取到的默认网关.....192.168.0.1

NetWork_task Init....

Task2 Init

connect OK

tcp create success

Recv = 20 2 0 0

Recv = 90 3 0 a 0

Recv = 40 2 0 1

Recv = d0 0 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

Recv = 40 2 0 1 这就是ping包发送给服务器，服务器返回的PINGRSP回复0xd0 00

Recv = d0 0 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

Recv = 40 2 0 1

Recv = d0 0 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

Recv = 40 2 0 1

Recv = d0 0

Recv = 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

Recv = 40 2 0 1

Recv = d0 0 30 18 0 a 73 65 72 76 65 72 72 65 63 76 73 65 72 76 65 72 31 32 33 34 35 36

Recv = 40 2 0 1

报文	描述	流向	值	固定报头	可变报头	负载
PINGRSP	PING命令回复	服务端->客户端	13	有	无	无

服务器接收 ping 成功，返回 0XD0 00.上面讲过。

设备端主动断开 MQTT 报文

报文	描述	流向	值	固定报头	可变报头	负载
DISCONNECT	断开连接	客户端->服务端	14	有	无	无

断开报文就发 E0 00 就可以了，很简单。因为 MQTT 断开，所以客户端也收不到什么东西。

MQTT.fx 1.7.1 版本下载使用

现在官网的 MQTT.fx 5.0 以上的版本都要注册或者收费，只有 1.7.1 版本是免费的，所以我用 1.7.1 版本。这是 1.7.1 版本软件的下载地址 https://file.bemfa.com/hw/zip/mqtt/mqttx_1.7.1_windows_64.exe

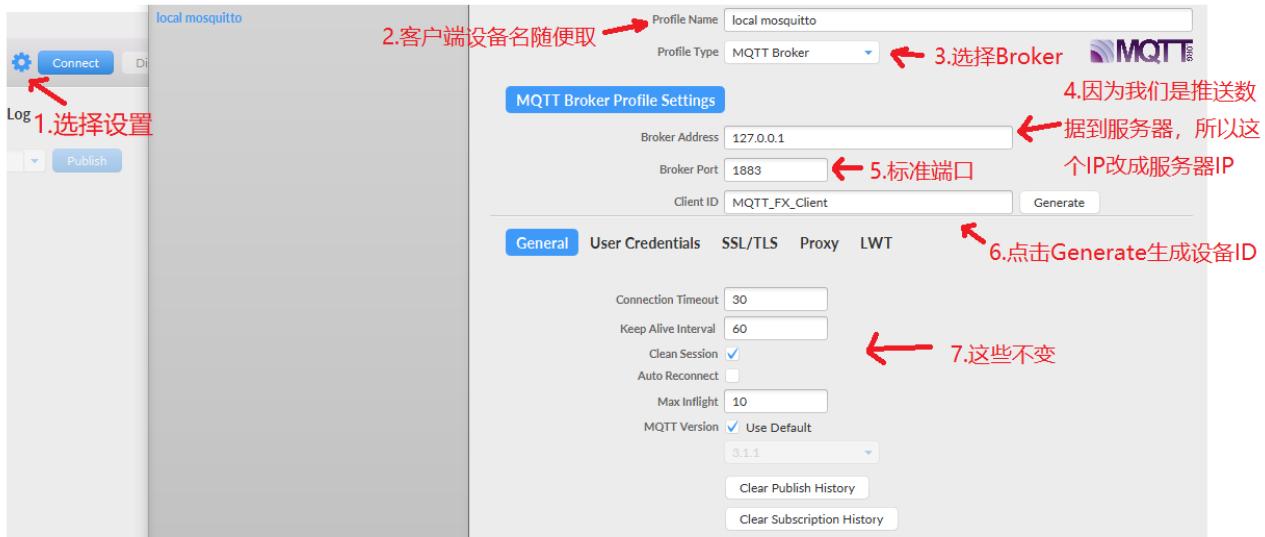


这是下载后的安装包，直接安装即可。



这是安装好的软件。

如果我现在是客户端设备，我实现推送



Profile Name: local mosquitto
Profile Type: MQTT Broker
Broker Address: 127.0.0.1
Broker Port: 1883
Client ID: MQTT_FX_Client
Connection Timeout: 30
Keep Alive Interval: 60
Clean Session: Auto Reconnect:
Max Inflight: 10
MQTT Version: Use Default: 3.1.1

改成了下面这样



Profile Name: local mosquitto
Profile Type: MQTT Broker
Broker Address: 192.168.0.112
Broker Port: 1883
Client ID: 3ce5cc706cb147699afe672cb0f34b73
Connection Timeout: 30
Keep Alive Interval: 60
Clean Session: Auto Reconnect:
Max Inflight: 10
MQTT Version: Use Default: 3.1.1

WireShark3.6.3 网络抓包工具使用



Wireshark-win64-3.6.3.exe

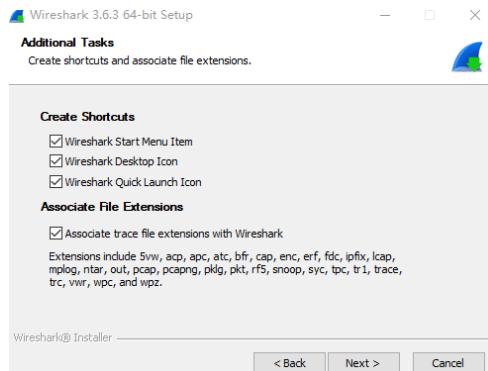
一直向下安装就是，遇到 802.11 的全部选择上继续安装

下载地址 <https://ftp.yz.yamagata-u.ac.jp/pub/network/security/wireshark/win64/all-versions/>



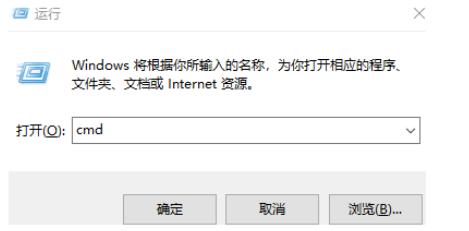
Wireshark-win64-3.6.3.exe

一定要下载 exe 的版本才能在 win10 上面安装哦



全部选择上一路向下安装完就是。中途跳出来的驱动也直接安装。

1. win+R使用cmd



C:\WINDOWS\system32\cmd.exe

Microsoft Windows [版本 10.0.19045.4291]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\xiang>ipconfig

Windows IP 配置

以太网适配器 以太网:

媒体状态 : 媒体已断开连接
连接特定的 DNS 后缀 :

以太网适配器 以太网 2:

连接特定的 DNS 后缀 :
本地链接 IPv6 地址 : fe80::555f:c1d:58a1:920a%16
IPv4 地址 : 192.168.3.2
子网掩码 : 255.255.255.0
默认网关. : 192.168.3.1

以太网适配器 VMware Network Adapter VMnet1:

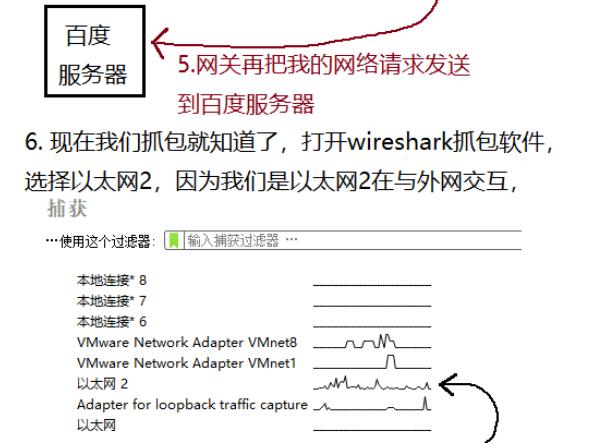
连接特定的 DNS 后缀 :
本地链接 IPv6 地址 : fe80::8d1a:a6da:78f7:3ce0%11
IPv4 地址 : 192.168.91.1
子网掩码 : 255.255.255.0
默认网关. :

2. 我使用的以太网2

连接的外网。也就是
网线插的以太网2网
卡，我是双网卡。



3. 本机IP



6. 现在我们抓包就知道了，打开wireshark抓包软件，
选择以太网2，因为我们是以太网2在与外网交互，
捕获

…使用这个过滤器: 输入捕获过滤器 ...

本地连接* 8

本地连接* 7

本地连接* 6

VMware Network Adapter VMnet8

VMware Network Adapter VMnet1

以太网 2

Adapter for loopback traffic capture

以太网

7. 你看以太网2有曲线在波动，证明我们

在频繁的连接外网

RAW API MODBUS TCP 移植

主要移植自己写的 MODBUS TCP 库,库的内容就只有一个 C 文件 modbus_tcp.c, 内容如下

```
#include "modbus_tcp.h"
#include "lwip/opt.h"
#include "lwip/arch.h"
#include "lwip/api.h"
#include "lwip/inet.h"
#include "lwip/tcp.h" //TCP 加入头文件
#include "stm32f4xx.h" //包含 F407 整个头文件

#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

//----- Define -----//

extern struct tcp_pcb *tcppcb;

//使用 LWIP RAW API 的 TCP 发送数据
void MODBUS_TCP_Send(uint8_t *pSendBuff,uint32_t len)
{
    tcp_write(tcppcb,pSendBuff,len,1); //TCP 发送数据
    tcp_output(tcppcb); //TCP 马上发送
}

//----- Function Prototype -----//
static char process(char *modbus_recvbuf);

static int mb_rsq_pdu(unsigned char *receive_buffer_temp,int cnt);
static int mb_excep_rsq_pdu(unsigned char *receive_buffer_temp,int error_code);
static int function_1(unsigned char *receive_buffer_temp);
static int function_2(unsigned char *receive_buffer_temp);
static int function_3(unsigned char *receive_buffer_temp);
static int function_4(unsigned char *receive_buffer_temp);
static int function_5(unsigned char *receive_buffer_temp,int cnt);
static int function_6(unsigned char *receive_buffer_temp,int cnt);
static int broadcast(unsigned char *receive_buffer_temp);

//----- Variable -----//
MODBUS_TCP_T modbus_tcp = {
    .process = process,
};

extern int modbus_conn;
char discrete_input[32] = {0x55,0x55,0x55};
char coil[32] = {0x55,0x55,0x55};
char input_reg[20] = {0,1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10}; //这就是要读取的输入寄存器, 以后要上传给服务器的数据, 就是修改这儿
char hold_reg[512] = {0x01,0x02,0x03}; //这就是要读取的输入寄存器, 以后要上传给服务器的数据, 就是修改这儿, 因为定义的是 8 位, 服务器上 16 位的排列方式就是 0x102,0x0300, 这种两个字节一组。
unsigned char receive_buffer_temp[100];

//这是固定模式的函数, 不需要修改
static char process(char *modbus_recvbuf)
{
    memcpy(receive_buffer_temp,modbus_recvbuf,12);
    memset(modbus_recvbuf,0,12);
    mb_rsq_pdu(receive_buffer_temp,12);

    return 0;
}
```

```

//得到网络数据, 选择功能码执行
static int mb_rsq_pdu(unsigned char *receive_buffer_temp,int counter_temp)
{
    if(receive_buffer_temp[0 + 6] == 0x01){
        switch(receive_buffer_temp[1 + 6]){
            case 1:
                function_1(receive_buffer_temp);
                break;
            case 2:
                function_2(receive_buffer_temp);
                break;
            case 3:
                function_3(receive_buffer_temp);
                break;
            case 4:
                function_4(receive_buffer_temp);
                break;
            case 5:
                function_5(receive_buffer_temp,counter_temp);
                break;
            case 6:
                function_6(receive_buffer_temp,counter_temp);
                break;
            default :
                mb_excep_rsq_pdu(receive_buffer_temp,1);
                break;
        }
    }else if(receive_buffer_temp[0 + 6] == 0){
        broadcast(receive_buffer_temp);
    }
    return 0;
}

//功能码 01, 接收处理
static int function_1(unsigned char *receive_buffer_temp)
{
    int i;
    unsigned short cnt;
    unsigned short coil_num;
    unsigned short start_address;
    int temp = 0;
    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];
    coil_num = receive_buffer_temp[4 + 6] << 8| receive_buffer_temp[5 + 6];

    if((start_address + coil_num) > 255){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }
    receive_buffer_temp[2 + 6] = ((coil_num % 8 )? (coil_num / 8 + 1) : (coil_num / 8));
    cnt = receive_buffer_temp[2 + 6] + 5 - 2;
    if(coil_num % 8){
        if(coil_num < 8){
            for(i = 0;i < coil_num;i ++)temp |= 1 << i;
            receive_buffer_temp[3 + 6] = ((coil[start_address / 8]) >> (start_address % 8) | (coil[start_address / 8 + 1]) << (8 - (start_address % 8))) & temp;
        }else {
            for(i = 0;i < receive_buffer_temp[2 + 6] - 1;i++)receive_buffer_temp[3 + i + 6] = (coil[i + start_address / 8]) >> (start_address % 8) | (coil[i + start_address / 8 + 1]) << (8 - (start_address % 8));
            receive_buffer_temp[3 + i + 6] = (coil[i + start_address / 8] << ((8 - (coil_num % 8 - start_address % 8) % 8)) & 0xff) >> (8 - (coil_num % 8));
        }
    }else {
        for(i = 0;i < receive_buffer_temp[2 + 6];i++)receive_buffer_temp[3 + i + 6] = (coil[i + start_address / 8]) >> (start_address % 8) | (coil[i + start_address / 8 + 1]) << (8 - (start_address % 8));
    }
    receive_buffer_temp[4] = (cnt & 0xff00) >> 8;
    receive_buffer_temp[5] = (cnt & 0x00ff);
    cnt = cnt + 6;
    MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,cnt); //采用 RAW API 的 TCP 数据发送函数
// write(modbus_conn, (const unsigned char*)receive_buffer_temp,cnt); //屏蔽掉 socket 的数据发送函数
    return 0;
}

```

```

}

//功能码 02, 接收处理
static int function_2(unsigned char *receive_buffer_temp)
{
    int i;
    unsigned short cnt;
    unsigned short discrete_num;
    unsigned short start_address;
    int temp = 0;

    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];

    discrete_num = receive_buffer_temp[4 + 6] << 8| receive_buffer_temp[5 + 6];

    if((start_address + discrete_num) > 255){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }

    receive_buffer_temp[2 + 6] = ((discrete_num % 8 )? (discrete_num / 8 + 1) : (discrete_num / 8));
    cnt = receive_buffer_temp[2 + 6] + 5 - 2;
    if(discrete_num % 8){
        if(discrete_num < 8){
            for(i = 0;i < discrete_num;i++)temp |= 1 << i;
            receive_buffer_temp[3 + 6] = ((discrete_input[start_address / 8]) >> (start_address % 8) | (discrete_input[start_address / 8 + 1]) << (8 - (start_address % 8))) & temp;
        }else {
            for(i = 0;i < receive_buffer_temp[2 + 6] - 1;i++)receive_buffer_temp[3 + i + 6] = (discrete_input[i + start_address / 8]) >> (start_address % 8) | (discrete_input[i + start_address / 8 + 1]) << (8 - (start_address % 8));
            receive_buffer_temp[3 + i + 6] = (discrete_input[i + start_address / 8] << ((8 - (discrete_num % 8 - start_address % 8) % 8)) & 0xff) >> (8 - (discrete_num % 8));
        }
    }else {
        for(i = 0;i < receive_buffer_temp[2 + 6];i++)receive_buffer_temp[3 + i + 6] = (discrete_input[i + start_address / 8]) >> (start_address % 8) | (discrete_input[i + start_address / 8 + 1]) << (8 - (start_address % 8));
    }

    receive_buffer_temp[4] = (cnt & 0xff00) >> 8;
    receive_buffer_temp[5] = (cnt & 0x00ff);
    cnt = cnt + 6;

    MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,cnt); //采用 RAW API 的 TCP 数据发送函数
    // write(modbus_conn, (const unsigned char*)receive_buffer_temp,cnt); //屏蔽掉 socket 的数据发送函数

    return 0;
}

//功能码 03, 接收处理
static int function_3(unsigned char *receive_buffer_temp)
{
    int i;
    int cnt;
    unsigned short int start_address;

    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];

    receive_buffer_temp[2 + 6] = receive_buffer_temp[5 + 6] * 2;

    if(receive_buffer_temp[2 + 6] > 48){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }

    if((start_address * 2 + receive_buffer_temp[2]) > 512){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
    }
}

```

```

    return 1;
}

cnt = receive_buffer_temp[2 + 6] + 5 - 2;
for(i = 0;i < receive_buffer_temp[2 + 6];i++){
    receive_buffer_temp[i + 3 + 6] = hold_reg[start_address * 2 + i];
}

receive_buffer_temp[4] = (cnt & 0xff00) >> 8;
receive_buffer_temp[5] = (cnt & 0x00ff);

cnt = cnt + 6;

MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,cnt); //采用 RAW API 的 TCP 数据发送函数
// write(modbus_conn, (const unsigned char*)receive_buffer_temp,cnt); //屏蔽掉 socket 的数据发送函数

return 0;
}

```

```

//功能码 04, 接收处理
static int function_4(unsigned char *receive_buffer_temp)
{
    int i;
    int cnt;
    unsigned short int start_address;

    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];

    receive_buffer_temp[2 + 6] = receive_buffer_temp[5 + 6] * 2;

    if((start_address * 2 + receive_buffer_temp[2 + 6]) > 20){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }

    cnt = receive_buffer_temp[2 + 6] + 5 - 2;
    for(i = 0;i < receive_buffer_temp[2 + 6];i++)receive_buffer_temp[i + 3 + 6] = input_reg[start_address * 2 + i];

    receive_buffer_temp[4] = (cnt & 0xff00) >> 8;
    receive_buffer_temp[5] = (cnt & 0x00ff);
    cnt = cnt + 6;

    MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,cnt);
// write(modbus_conn, (const unsigned char*)receive_buffer_temp,cnt);

    return 0;
}

```

```

//功能码 05, 接收处理
static int function_5(unsigned char *receive_buffer_temp,int counter_temp)
{
    unsigned short start_address;

    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];

    if(start_address > 255){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }

    if((receive_buffer_temp[4 + 6] == 0xff) && (receive_buffer_temp[5 + 6] == 0x00)){
        coil[(start_address / 8)] |= 1 << start_address % 8;
    }else if((receive_buffer_temp[4 + 6] == 0x00) && (receive_buffer_temp[5 + 6] == 0x00)){

```

```

coil[(start_address / 8)] &= ~(1 << start_address % 8);
}else {
    mb_excep_rsq_pdu(receive_buffer_temp,3);
}

receive_buffer_temp[4] = (6 & 0xff00) >> 8;
receive_buffer_temp[5] = (6 & 0x00ff);

MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,counter_temp);
// write(modbus_conn, (const unsigned char*)receive_buffer_temp,counter_temp);

return 0;
}

//功能码 06, 接收处理
static int function_6(unsigned char *receive_buffer_temp,int counter_temp)
{
    unsigned short start_address;

    start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];

    if(start_address > 255){
        mb_excep_rsq_pdu(receive_buffer_temp,2);
        return 1;
    }

    hold_reg[start_address * 2] = receive_buffer_temp[4 + 6];
    hold_reg[start_address * 2 + 1] = receive_buffer_temp[5 + 6];

    receive_buffer_temp[4] = (6 & 0xff00) >> 8;
    receive_buffer_temp[5] = (6 & 0x00ff);

    MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,counter_temp);
    // write(modbus_conn, (const unsigned char*)receive_buffer_temp,counter_temp);

    return 0;
}

static int mb_excep_rsq_pdu(unsigned char *receive_buffer_temp,int error_code)
{
    receive_buffer_temp[1 + 6] |= 0x80;
    switch(error_code) {
        case 1:
            receive_buffer_temp[2 + 6] = 1;
            break;
        case 2:
            receive_buffer_temp[2 + 6] = 2;
            break;
        case 3:
            receive_buffer_temp[2 + 6] = 3;
            break;
        case 4:
            receive_buffer_temp[2 + 6] = 4;
            break;
        default :
            break;
    }
    receive_buffer_temp[4] = (3 & 0xff00) >> 8;
    receive_buffer_temp[5] = (3 & 0x00ff);

    MODBUS_TCP_Send((uint8_t *)receive_buffer_temp,9);
    // write(modbus_conn, (const unsigned char*)receive_buffer_temp,9);

    return 0;
}

```

```

}

static int broadcast(unsigned char *receive_buffer_temp)
{
    int start_address;
    switch(receive_buffer_temp[1 + 6]){
        case 5:
            start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];
            if(start_address > 255){
                return 1;
            }
            if((receive_buffer_temp[4 + 6] == 0xff) && (receive_buffer_temp[5 + 6] == 0x00)){
                coil[(start_address / 8)] |= 1 << start_address % 8;
            }else if((receive_buffer_temp[4 + 6] == 0x00) && (receive_buffer_temp[5 + 6] == 0x00)){
                coil[(start_address / 8)] &= ~(1 << start_address % 8);
            }
            break;
        case 6:
            start_address = (receive_buffer_temp[2 + 6] << 8) | receive_buffer_temp[3 + 6];
            if(start_address > 255){
                return 1;
            }
            hold_reg[start_address * 2] = receive_buffer_temp[4 + 6];
            hold_reg[start_address * 2 + 1] = receive_buffer_temp[5 + 6];
            break;
    }
    return 0;
}

```

主函数实现

```

uint8_t tcp_recvbuf[1024]; //定义 TCP 接收的数据缓存区 1000 字节

struct tcp_pcb *tcppcb = NULL; //必须定义一个 TCP 控制块

static void client_err(void *arg, err_t err)
{
    tcp_close(tcppcb);
}

static uint32_t RxLen = 0; //接收数据长度

static err_t server_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
    if(p != NULL)
    {
        tcp_recved(tpcb, p->tot_len);

        memcpy(tcp_recvbuf, p->payload, p->len); //TCP 接收数据拷贝给 tcp_recvbuf
        RxLen = p->len; //接收到的数据长度
        pbuf_free(p);
    }
    else if(err == ERR_OK)
    {
        debug_printf("server disconnected\r\n"); //服务器断开，提示
        tcp_close(tpcb);
    }
}

return ERR_OK;
}

```

```

/*
* 本机是服务端，接收到 PC 端或者其它服务器做 MODBUS 主机发来的 TCP 数据
*/
static err_t tcpecho_accept(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    tcp_recv(newpcb,server_recv);
    return ERR_OK;
}

//任务句柄
TaskHandle_t Task2_Handler;

struct tcp_pcb *tcppcb;
struct tcp_pcb *tcppcbconn;

void Task2(void *pvParameters)
{
    tcppcb = tcp_new(); //创建 TCP 控制块
    if(tcppcb)
    {
        tcp_bind(tcppcb, IP_ADDR_ANY, 8888); //我现在测试，将 MODBUS TCP 端口设置成 8888，一般官方默认是 502，记住
        tcppcbconn = tcp_listen(tcppcb); //进入监听状态
        tcp_accept(tcppcbconn,tcpecho_accept); //客户端连接上服务端
    }

    tcp_err(tcppcb,client_err);

    while(1) //死循环中再次申请 UDP，再次发送
    {
        while(RxLen > 0) //接收到数据，进入
        {
            debug_printf("MODBUS TCP RxLen = %d hold = %x %x\r\n",RxLen,hold_reg[0],hold_reg[1]); //接收数据长度，03 功能地址 1 数据测试
            modbus_tcp.process((char *)tcp_recvbuf);
            RxLen = 0; //清除接收数据，退出循环
        }

        vTaskDelay(5); //延时 100ms
    }
}

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2
    delay_init(168); //延时初始化
    uart_init(115200); //串口初始化波特率为 115200
    TIM3_Int_Init(999,839);
    my_mem_init(SRAMIN); //初始化内部内存池

    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1，因为 debug_printf 函数最后一行才会给 semDebug 写 1

    printf("xxxxxxxxxx\r\n");

    while(lwip_comm_init()) //lwip 初始化
    {
        printf("lwip_comm_init failed...\r\n");
        delay_ms(1000);
    }

    printf("lwip_success_init...\r\n");

    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0xFF)) //等待 DHCP 获取成功/超时溢出
    {
}

```

```

        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数
    }

    //创建开始任务
    xTaskCreate((TaskFunction_t )NetWork_task,           //任务函数 网络通信 LWIP 内核中断维护
                (const char* )"NetWork_task",           //任务名称
                (uint16_t )512,                      //任务堆栈大小
                (void* )NULL,                       //传递给任务函数的参数
                (UBaseType_t )3,                   //任务优先级
                (TaskHandle_t* )&NetWorkTask_Handler); //任务句柄

    //创建任务
    xTaskCreate((TaskFunction_t )Task2,           //任务函数
                (const char* )"Task2",           //任务名称
                (uint16_t )2048,                 //任务堆栈大小
                (void* )NULL,                  //传递给任务函数的参数
                (UBaseType_t )2,               //任务优先级
                (TaskHandle_t* )&Task2_Handler); //任务句柄

    vTaskStartScheduler(); //开启任务调度

    while(1)
    {

    }
}

```

将 char 寄存器转换成 16 位寄存器，因为 16 位寄存器是 MODBUS 标准寄存器

```

extern int modbus_conn;
char discrete_input[32] = {0x55,0x55,0x55};
char coil[32] = {0x55,0x55,0x55};
char input_reg[20] = {0,1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10};
char hold_reg[512] = {0x01,0x02,0x03};
unsigned char receive_buffer_temp[100];

```

这就是MODBUS_TCP的寄存器，但是库采用的是char, 8位排列的
现在以hold_reg为例，转换MODBUS寄存器为16位



我将hold_reg寄存器转换成16位存储

```

typedef struct
{
    uint16_t reg0;
    uint16_t reg1;
    uint16_t reg2;
    uint16_t reg3;
    uint16_t reg4;
}MODBUS_Reg; //MODBUS寄存器，一般MODBUS寄存器都是16位

```

```

typedef struct
{
    uint16_t reg0;
    uint16_t reg1;
    uint16_t reg2;
    uint16_t reg3;
    uint16_t reg4;
}MODBUS_Reg; //MODBUS 寄存器，一般 MODBUS 寄存器都是 16 位

```

```

typedef union
{
    MODBUS_Reg reg;
    uint16_t array[256];
}

```

```

)MODBUS_Regun;

MODBUS_Regun Reg;

void MODBUS_RegOpt(void)
{
    uint32_t Count = 0;
    for(int i = 0; i < 256; i++)
    {

        Reg.array[i] = hold_reg[Count];
        Reg.array[i] = Reg.array[i] << 8;
        Count++;
        Reg.array[i] = Reg.array[i] | hold_reg[Count];
        Count++;
    }
}

/*************
* 8 位 MODBUS 寄存器到 16 位转换
*****/

void MODBUS_RegOpt(void)
{
    uint32_t Count = 0;
    for(int i = 0; i < 256; i++)
    {

        Reg.array[i] = hold_reg[Count];
        Reg.array[i] = Reg.array[i] << 8;
        Count++;
        Reg.array[i] = Reg.array[i] | hold_reg[Count];
        Count++;
    }
}

```

主函数在得到 MODBUS TCP 数据的时候直接转换

```

while(1) //死循环中再次申请 UDP, 再次发送
{
    while(RxLen > 0) //接收到数据, 进入
    {
        debug_printf("MODBUS TCP RxLen = %d \r\n",RxLen); //接收数据长度, 03 功能地址 1 数据测试
        modbus_tcp.process((char *)tcp_recvbuf);
        RxLen = 0; //清除接收数据, 退出循环

        MODBUS_RegOpt(); //直接将 8 位转换成 16 位, 存入 MODBUS 寄存器

        debug_printf("Reg0 = %d \r\n",Reg.reg.reg0); //打印确认是否转换成功
        debug_printf("Reg1 = %d \r\n",Reg.reg.reg1);
        debug_printf("Reg2 = %d \r\n",Reg.reg.reg2);
        debug_printf("Reg3 = %d \r\n",Reg.reg.reg3);
        debug_printf("Reg4 = %d \r\n",Reg.reg.reg4);

    }

    vTaskDelay(5); //延时 100ms
}

```

```

网卡en的MAC地址为:..... 2.0.0.78.0.48
通过DHCP获取到IP地址..... 192.168.0.103
通过DHCP获取到子网掩码..... 255.255.255.0
通过DHCP获取到的默认网关..... 192.168.0.1
Network_task Init...
MODBUS TCP RxLen = 12
Reg0 = 258
Reg1 = 768
Reg2 = 0
Reg3 = 0
Reg4 = 0
MODBUS TCP RxLen = 12
Reg0 = 258
Reg1 = 768
Reg2 = 0
Reg3 = 0
Reg4 = 0
MODBUS TCP RxLen = 12
Reg0 = 258
Reg1 = 768
Reg2 = 0
Reg3 = 0
Reg4 = 0

```

系统开机默认是 258,768。

```

extern int modbus_conn;
char discrete_input[32] = {0x55,0x55,0x55};
char coil[32] = {0x55,0x55,0x55};
char input_req[20] = {0,1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10};
char hold_reg[512] = {0x01,0x02,0x03};
unsigned char receive_buffer_temp[100];

```

这是因为在 modbus 库定义变量的时候固定了 hold_reg 的值。下面我直接用 poll 去修改。

	Alias	00000
0		1000
1		2000
2		3000
3		4000
4		5000
5		0

我 POLL 修改了数据

```

Alias - 00000
MODBUS TCP RxLen = 12
Reg0 = 1000
Reg1 = 2000
Reg2 = 3000
Reg3 = 4000
Reg4 = 5000
MODBUS TCP RxLen = 12
Reg0 = 1000
Reg1 = 2000
Reg2 = 3000
Reg3 = 4000
Reg4 = 5000

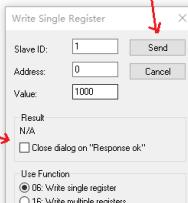
```

板子上寄存器也被需要改了。

但是还有一个问题，我 POLL 去修改数据，数据虽然修改成功了，修改窗口没有关闭

	Alias	00000
0		1000
1		2000
2		3000
3		4000
4		5000
5		0
6		0
7		0
8		0
9		0

点击Send之后，修改成功，但是该窗口不得关闭



这是因为板子没有回复OK

这个问题可以解决，也可以不解决。我下面还是解决