

计算机网络-编程作业一实验报告

2012039 冯朝芑

概要

本项目基于Socket实现了简单的群聊程序。项目基于TCP协议，包含服务器端和客户端，客户端将消息发送给服务器，再由服务器将消息转发给其余客户端。

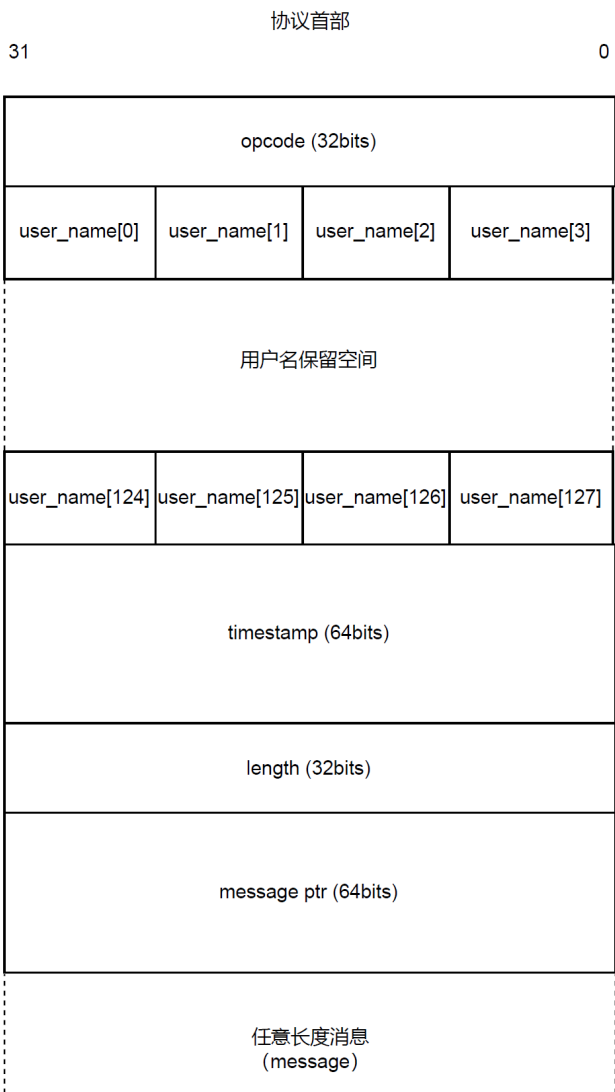
实现功能包括：多个聊天室群聊、私聊、用户名设置、退群、获取已有群组信息、退出程序等。多线程主要基于pthread实现。

协议设计

协议语法

本项目设计的协议由首部和消息本体构成。首部用于指明该条消息的含义、来源等，具体包括控制号、用户名、时间戳、消息长度和消息指针等部分。消息本体为具体需要传输的内容。

本项目定义的协议首部如下图所示。



在进行传输时，可以将首部与消息本体一同或分别发送。在接收程序中通过重定向message ptr来定位接收到的消息本体，以实现更加灵活的传输和较为便利的使用。但是message ptr和时间戳的存在，导致了此首部具有了一定的平台相关性，相关内容将在“遇到的问题”部分展开。

opcode现支持的类型有：

```
plan_text = 0,          //一般消息
join = 1,              //加入群组
user_registration = 2,   //用户注册
user_group_info = 3,     //请求群组列表
quit_group = 4,         //退出群组
program_exit = 254,      //退出程序
error = 255            //错误
```

user_name字段空间较大，一方面为了支持更长的用户名，另一方面也为协议的补充完善保留了空间。

时间戳字段将反映用户发送消息的时间。

长度字段指明了消息体的字节长度，接收设备可以据此分配相应的内存空间。

消息类型

plan_text类型消息，消息本体部分为用户发送的文字内容。

join类型消息，消息本体部分为用户要加入的组号（ASCII码）。

user_registration类型消息，消息本体部分为新用户用户名。

其余消息类型，消息本体部分不做要求。

协议时序

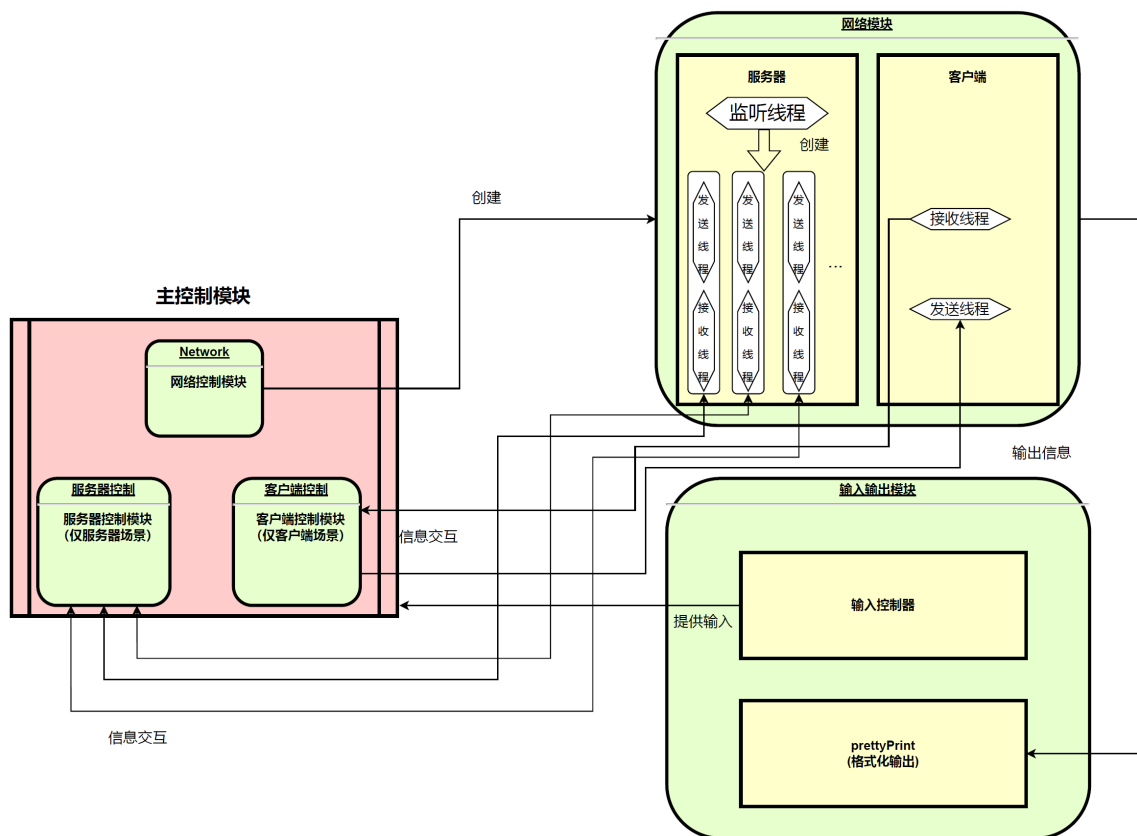
“群组”是该项目中用户进行聊天的基本单位，群组中的用户必须具有用户名。一个用户某一时刻只能向其参与的一个群组发送消息，但是可以接收来自其加入的其他群组的消息。

用户登录服务器之后，发送的第一条消息首部的opcode必须为用户注册。之后服务器会自动向用户返回群组列表，之后用户必须选择一个群组加入，即发送join类型消息。

用户加入群组后，可以使用plan_text类型消息发送文字内容。在用户使用过程中，可以随时发送退出群组、请求群组列表、退出程序等消息以实现相关功能。

功能模块

本程序的大致功能模块划分如下图所示。



主控制模块：主控制模块是程序的入口，它将负责启动网络控制模块、输入输出模块以及服务器控制模块或客户端控制模块。

网络控制模块：网络控制模块负责实例化网络模块中的服务器或客户端，并调用其接口以实现网络模块从初始化、构建连接到释放资源的过程

服务器控制模块：该模块根据接收到的不同消息首部的opcode执行不同的行为，如向对应的群组转发消息、维护储存群组信息、用户信息的数据结构等等。

客户端控制模块：该模块根据接收到的不同消息首部的opcode执行不同的行为，并且在登录时提示用户填写用户名、选择初始群组。

网络模块：

服务器：服务器具有对应的接口实现了加载Winsock DLL、创建绑定Socket、打开监听、创建新的Socket、创建对应Socket的接收线程、发送线程等。其中，接收线程负责从对应的Socket接收消息，之后进行封装通过阻塞队列发送给服务器控制模块；发送线程从其所属的阻塞队列中接收服务器控制模块发给它的消息，之后进行发送。

客户端：客户端具有对应的接口实现了加载Winsock DLL、创建Socket、连接服务器、创建发送线程、接收线程等。其中，接收线程负责从Socket接收消息，之后进行封装调用格式化输出模块将消息打印到屏幕上；发送线程从阻塞队列中接收客户端控制模块发给它的消息，之后进行发送。

输入控制器：输入控制器从键盘接收用户输入，识别控制命令，并且将输入封装，通过阻塞队列发送给服务器或客户端控制模块。

prettyPrint：负责提供较为规整的输出。

阻塞队列：主控制模块和网络模块各线程以及输入输出模块间的通信是通过阻塞队列实现的，也就是形成了常说的“生产者-消费者模型”。这主要是考虑到多线程的需求：在服务器模式下将创建众多的接收、发送线程对（接收和发送不能写在同一个线程中，因为他们都是阻塞操作！），在客户端模式下也将有一对的接收、发送线程。用户输入、接收消息到来的时间都是不可预测的，直接要求对应的线程随时处于准备状态是不可能的。阻塞队列就提供了这样一种相对简单的方式，控制模块只需要不断地获取队列

中的消息，并将其转发到目标队列中；而网络接收线程只需要不断地将接收到的消息放入队列，网络发送线程将消息取出发送即可。

代码展示和讲解

消息头和opcode:

```
#pragma pack (1)
struct message{
    unsigned int opcode;
    char user_name[128];
    time_t time;
    unsigned int length;
    char *message;
};
#pragma pack ()

enum opcode {
    user_registration = 2,
    join = 1,
    plan_text = 0,
    user_group_info = 3,
    quit_group = 4,
    program_exit = 254,
    error = 255
};
```

服务器控制模块:

该线程由主控制模块启动，从阻塞队列中获得消息并放入对应的（一个或多个）队列中去

```
void* serverController(void* arg){
    Task t, newT;
    bool noExit = true;
    while(noExit){
        //从N2M队列取数据
        bqN2M->Get(t);

        switch(t.b.m.opcode){
            case user_registration:
                //用户名注册
                //两个map要同步
                if(socket_username.count(t.b.socket) == 0)
                    groups[groups.size()].insert(t.b.socket);
                socket_username[t.b.socket] = t.b.m.message;
                cout<<"registered: "<<t.b.m.message<<endl;
                // 发送可用群组信息表（写阻塞队列）
                newT.b = getGroupInfo();
                pthread_mutex_lock(&server::bqM2NLock);
                (*bqM2N)[t.b.socket]->Put(newT);
                pthread_mutex_unlock(&server::bqM2NLock);
                continue;

            case join:
                //用户加入，修改对应的数据结构
```

```

        if(socket_username.count(t.b.socket)) {
            groups[atoi(t.b.m.message)].insert(t.b.socket);
            currGroup[t.b.socket] = atoi(t.b.m.message);
            cout<<"successfully joined"<<endl;
        }
        continue;

    case quit_group:
        //用户退出
        groups[currGroup[t.b.socket]].erase(t.b.socket);
        // 发送可用群组信息表 (写阻塞队列)
        newT.b = getGroupInfo();
        pthread_mutex_lock(&server::bqM2NLock);
        (*bqM2N)[t.b.socket]->Put(newT);
        pthread_mutex_unlock(&server::bqM2NLock);
    case plan_text:
        //普通文字信息
        //根据保存的群组信息, 将消息装入对应的阻塞队列中
        for (auto i:groups[currGroup[t.b.socket]]) {
            pthread_mutex_lock(&server::bqM2NLock);
            (*bqM2N)[i]->Put(t);
            pthread_mutex_unlock(&server::bqM2NLock);
        }
        continue;
    case program_exit:
        // 服务器退出
        noExit = false;
        // 通知每一个线程
        pthread_mutex_lock(&server::bqM2NLock);
        for(auto& bqPtr : *bqM2N){
            bqPtr.second->Put(t);
        }
        pthread_mutex_unlock(&server::bqM2NLock);
        break;
    default:
        break;
    }
}
cout<<"Server Controller Quit"<<endl;
}

```

网络部分服务器主循环:

负责处理连接请求

```

int server::startMainLoop() {
    /*在端口进行监听, 一旦有客户机发起连接请求
    就建立与客户机进行通信的线程*/
    while (!shouldExit) {
        /*监听是否有连接请求*/
        clientVec.emplace_back();
        sockaddr_in& client = clientVec.back();
        // 创建新的sClient
        sClient = accept(sListen, (struct sockaddr*)&client, &AddrSize);
        if (sClient == INVALID_SOCKET) {

```

```

        printf("accept() 失败: %d\n", WSAGetLastError());
        break;
    }
    printf("接受客户端连接: %s:%d\n",
        inet_ntoa(client.sin_addr),
        ntohs(client.sin_port));

    dwThreadVec.emplace_back();
    //创建一个接收线程
    hThreadVec.push_back(CreateThread(NULL, 0, ClientThread,
        (LPVOID)sClient, 0, &dwThreadVec.back()));

    //创建一个发送线程
    sendThreadVec.emplace_back();
    // 每个从main接收的信道都是一个生产、一个发送, 不需要在生产发送过程中抢锁
    pthread_t& newSend = sendThreadVec.back();
    // 避免同时操作vector
    pthread_mutex_lock(&bqM2NLock);
    (*bqM2N)[sClient] = new BBlockQueue(5);
    pthread_mutex_unlock(&bqM2NLock);
    channel c{(*bqM2N)[sClient], sClient};
    pthread_create(&newSend, NULL, sendToClient, (void*)&c);

    HANDLE& hThread = hThreadVec.back();
    if (hThread == NULL) {
        printf("CreateThread() 失败: %d\n", GetLastError());
        break;
    }
    //处理完后关闭
    //回收句柄而不是关闭线程
    CloseHandle(hThread);
}

cout<<"Server Main Loop Quit"<<endl;
}

```

网络部分服务器发送线程:

负责从对应的阻塞队列中拿出消息并发送。

```

void *server::sendToClient(void *arg) {
    channel* c = (channel*)arg;
    BBlockQueue* myQueue = c->in;
    SOCKET mySocket = c->out;
    char Buffer[DEFAULT_BUFFER];
    cout<<"send thread create finished"<<endl;
    bool noExit = true;
    while(noExit){
        //取数据
        Task t;
        myQueue->Get(t);
        // 判断是否应该退出
        if(t.b.m.opcode == program_exit){
            noExit = false;
            shouldExit = true;
        }
    }
}

```

```

// 拷贝头部和消息本体到Buffer
memset(&Buffer, 0, sizeof(char)*DEFAULT_BUFFER);
memcpy(Buffer, &t.b.m, HEADER_LENGTH);
memcpy(&Buffer[HEADER_LENGTH], t.b.m.message, sizeof(char)*
(strlen(t.b.m.message) + 1));
int ret = send(mySocket, Buffer, HEADER_LENGTH + strlen(t.b.m.message) +
1, 0);
if (ret == 0) {
    break;
} else if (ret == SOCKET_ERROR) {
    printf("send() 失败: %d\n", WSAGetLastError());
    break;
}
printf("Server Send %d bytes\n", ret);
}

cout<<"Server Sender Quit"<<endl;
return nullptr;
}

```

网络部分服务器接收线程:

负责接收消息并放入对应的阻塞队列。

```

//从client接收消息
DWORD server::ClientThread(LPVOID lpParam) {
    SOCKET sock = (SOCKET) lpParam;
    // 每个线程有一个
    char Buffer[DEFAULT_BUFFER];
    int ret;
    while (!shouldExit) {
        /*接收来自客户机的消息*/
        // 清空Buffer
        memset(&Buffer, 0, sizeof(char)*DEFAULT_BUFFER);
        //接收消息
        ret = recv(sock, Buffer, DEFAULT_BUFFER, 0);
        if (ret == 0)
            break;
        else if (ret == SOCKET_ERROR) {
            printf("recv() 失败: %d\n", WSAGetLastError());
            break;
        }
        Buffer[ret] = '\0';
        //消息长度
        cout<<"ret: "<<ret<<endl;
        //拆出首部
        message* m = extractHeader(&Buffer[0]);
        //第一层封装, 线程间通信使用的Bean
        auto bean = new struct bean;
        bean->socket = sock;
        bean->m = *m;
        //打印接收到的消息
        prettyPrint::pretty(*bean);
        //第二层封装, 阻塞队列中的Task
        Task t(*bean);
    }
}

```

```
pthread_mutex_lock(&toMainLock);  
bqN2M->Put(t);  
pthread_mutex_unlock(&toMainLock);  
}  
cout<<"Server Receiver Quit"<<endl;  
return 0;  
}
```

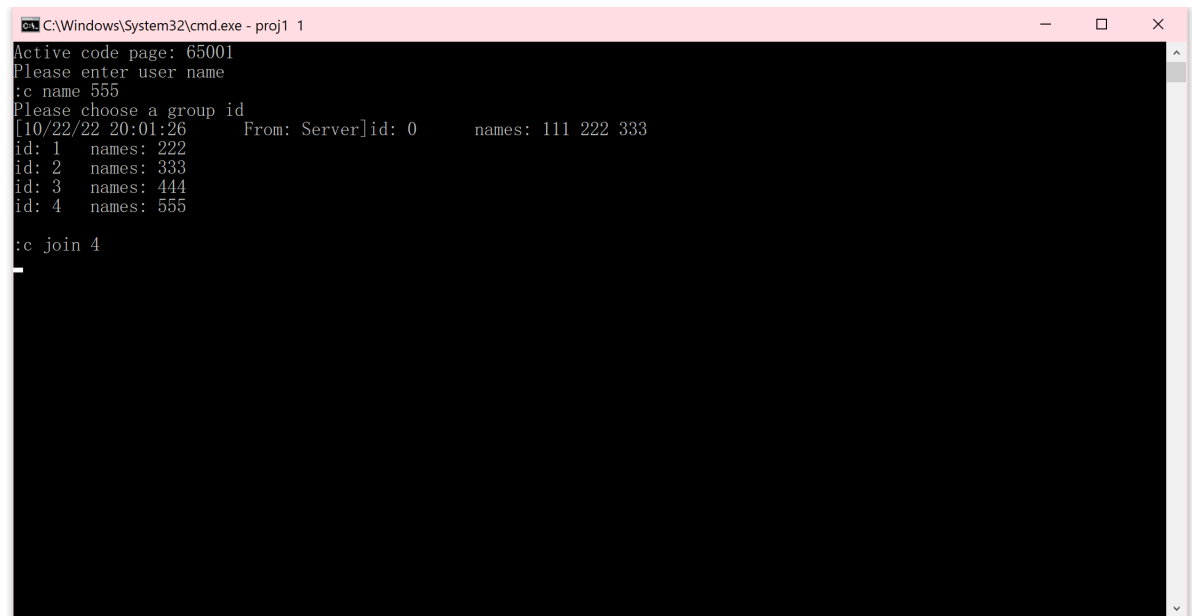
运行截图

先启动服务端（使用参数0），之后可以打开客户端（使用参数1）

需要进行指令控制时，用户需要遵循以下的格式输入：:c opname [option]

客户端启动后，用户需要使用:c name命令输入用户名；之后从给出的群组列表选择一个使用:c join命令加入之。

加入后客户端的显示情况如图所示。



```
C:\Windows\System32\cmd.exe - proj1 1  
Active code page: 65001  
Please enter user name  
:c name 555  
Please choose a group id  
[10/22/22 20:01:26 From: Server]id: 0 names: 111 222 333  
id: 1 names: 222  
id: 2 names: 333  
id: 3 names: 444  
id: 4 names: 555  
:c join 4
```

此时服务端显示了已加入的客户端信息


```
C:\Windows\System32\cmd.exe - proj1 0
Active code page: 65001
接受客户端连接: 127.0.0.1:6587
send thread create finished
ret: 156
[10/22/22 19:59:07 From: default]111
registered: 111
Server Send 171 bytes
ret: 154
[10/22/22 19:59:20 From: default]0
successfully joined
接受客户端连接: 127.0.0.1:6588
send thread create finished
ret: 156
[10/22/22 19:59:32 From: default]222
registered: 222
Server Send 189 bytes
ret: 154
[10/22/22 19:59:39 From: default]0
successfully joined
接受客户端连接: 127.0.0.1:6591
send thread create finished
ret: 156
[10/22/22 20:00:01 From: default]333
registered: 333
Server Send 211 bytes
ret: 154
[10/22/22 20:00:11 From: default]0
successfully joined
接受客户端连接: 127.0.0.1:6596
send thread create finished
ret: 156
[10/22/22 20:00:59 From: default]444
registered: 444
Server Send 233 bytes
ret: 154
[10/22/22 20:01:10 From: default]3
successfully joined
接受客户端连接: 127.0.0.1:6598
send thread create finished
ret: 156
[10/22/22 20:01:26 From: default]555
registered: 555
Server Send 251 bytes
ret: 154
[10/22/22 20:01:33 From: default]4
successfully joined
```

之后在客户端只需要正常的输入文本即可发送。

注意下面图片中的显示信息。第一张图片中222用户加入了两个群聊，能从两处接收消息。第二张图片中，只加入了群组0的用户只能从该群组接收消息。（333用户接收不到“hello from 4”）群组之间的消息是互相隔离的。

```
C:\Windows\System32\cmd.exe - proj1 1
Active code page: 65001
Please enter user name
:c name 222
Please choose a group id
[10/22/22 21:58:51 From: Server]id: 0 names: 111
id: 1 names: 222

:c join 0
[10/22/22 22:00:24 From: 333]hello from 3
[10/22/22 22:00:35 From: 111]hello from 1
hello from 2
[10/22/22 22:00:46 From: 222]hello from 2
[10/22/22 22:00:53 From: 444]hello from 4
[10/22/22 22:01:16 From: 111]again from 1
```

```
C:\Windows\System32\cmd.exe - proj1 1
Active code page: 65001
Please enter user name
:c name 333
Please choose a group id
[10/22/22 21:59:00 From: Server]id: 0 names: 111
id: 1 names: 222
id: 2 names: 333

:c join 0
hello from 3
[10/22/22 22:00:24 From: 333]hello from 3
[10/22/22 22:00:35 From: 111]hello from 1
[10/22/22 22:00:46 From: 222]hello from 2
[10/22/22 22:01:16 From: 111]again from 1
```

在聊天过程中使用:c quit命令可以退出群聊，服务器将返回最新的群组信息，之后可再次使用:c join命令加入群聊。

```
C:\Windows\System32\cmd.exe - proj1 1
Active code page: 65001
Please enter user name
:c name 111
Please choose a group id
[10/22/22 20:12:41] From: Server]id: 0      names: 111

:c join 0
[10/22/22 20:13:31] From: 333]hello
hi
[10/22/22 20:13:39] From: 111]hi
:c quit
Please choose a group id
[10/22/22 21:09:30] From: Server]id: 0      names: 222 333
id: 1      names: 222 444
id: 2      names: 333
id: 3      names: 444

:c join 3
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
[10/22/22 21:09:43] From: 111]hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
[10/22/22 21:11:09] From: 444]hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
```

聊天中其他用户则会收到有人退群的消息。

```
[10/22/22 21:09:30] From: 111]111 quits
```

但要注意的是每用户所发的消息都只能发送到唯一——一个群组中，这个群组是该用户最后一次加入的，即使一个用户处在多个群组中，也要先退出当前群组，再加入另一个群组，才能向其中发送信息。

用户可以使用:c exit退出程序

```
C:\Windows\System32\cmd.exe
Active code page: 65001
Please enter user name
:c name 111
Please choose a group id
[10/22/22 21:24:25] From: Server]id: 0      names: 111

:c join 111
123123123123
[10/22/22 21:24:36] From: 111]123123123123
[10/22/22 21:24:57] From: 222]55555555
:c exit
Input Controller Quit
Client Sender Quit
[10/22/22 21:25:05] From: Server]id: 0      names: 111 222
id: 2      names: 222
id: 111 names:
Client Receiver Quit

C:\Users\'Confidence\'F\Documents\dev_networks\lab1>
```

遇到的问题

1 在协议首部中加入的指针和时间戳引入了平台相关性

指针和时间戳的长度各平台是不同的（部分老旧机器的时间戳是32bits）。由于开发和测试过程中使用同一台64位机器，这使得该问题并没有暴露出来。但考虑到不论平台如何，第一次拿到首部后一定会对该指针重新赋值，以使其指向接收机内存中的储存消息本体的地址，也许也不会造成更大问题。然而在今后的协议设计过程中仍要注意保证平台无关性。

2 协议首部的储存对齐问题

从直观上看，一个好的协议首部在储存上应该是连续存放的。这一方面是节约空间，另一方面也提升了实现的简单性，比如可以使用地址+固定偏移量的方式读取首部。然而由于内存对齐的存在，这个首部在本机会上会向8字节对齐，最终占据160字节的空间。如果其他实现采用地址+固定偏移量的方式读取首部，将造成错位。

解决方法是在定义首部的结构体前后加上：

```
#pragma pack (1)    //按1字节对齐  
...  
#pragma pack ()     //恢复默认
```