一、使用到的自定义类

1.booker 类

存放 booker 的 name，ID，需求的口罩数量，booker 接收消息的地址和端口，booker 是否在线以及 booker 所在的行政区名字

```
class booker:
    def __init__(self):
        self.name = ''
        self.ID = -1
        self.needs = 50
        self.addr = ('localhost',54321)
        self.online = False
        self.District = ''
```

2.district 类

存放行政区的名字，行政区内的预定者 ID 的列表(BOOKERS)，行政区是否处于预约之中，行政区剩余的口罩数量，行政区当前的 booker 数量

```
class district:
    DEFAULT_maskNUm = 100000
    def __init__(self,name = ''):
        self.Dname = name
        self.BOOKERS = []
        self.inRound = False
        self.BookerNum = 0
        self.maskNum = district.DEFAULT_maskNUm
```

3.ins 类

即 instruction，存放了指令类别，参数数组，以及指令的发送者的地址(如果指令是来自于一个客户端的话)。

```
def __init__(self):
    self.from_addr = ('localhost',54321)
    self.OP = ''     #指令类别
    self.args = []  #参数数组
```

ins 类有一个 Read 函数，用来将字符串形式的命令转换为 ins 对象，即根据字符串形式的命令设置 OP 和得到 args 数组。若转换成功，Read 函数返回 True，否则返回 False

ins 类还指定了所有可用的指令类型，详细代码见 masksystem.py

二、服务端概述

服务端运行两个线程，一个线程用于管理员的交互，一个线程用于接收用户的消息(包括处理客户端指令以及接收用户法来的讯息)

```
def run(self):
    #处理客户端消息的线程
    booker_thread = Thread(target=self.thread_of_booker,args=())
    booker_thread.setDaemon(True)
    booker_thread.start()
    #与管理员交互的线程
    self.thread_of_admin()
```

服务端需要保存行政区信息，所以 server 类里有一个 DISTRICTS 字典，key 为 distirctName，value 为 district 对象。服务端还要保存所有 booker 的信息，所以有一个 all_bookers 列表，其为 booker 对象的列表，下标即为 bookerID。然后 server 端还包括一个用于接收的套接字和两个用于不同进程发送操作的套接字。

以及，server 端还有一个邮箱，用来存放用户发送的信息。

```
class server:
    def __init__(self):
        self.DISTRICTS = dict()
        self.all_bookers = []
        self.current_district = ''
        self.port = 12345
        self.ip = 'localhost'
        self.recv_socket = socket(AF_INET,SOCK_DGRAM)
        self.send_socket = socket(AF_INET,SOCK_DGRAM)
        self.send_socket_another = socket(AF_INET,SOCK_DGRAM)
        self.newest_text = ''
        self.text = []
```

1.与管理员交互的线程

与管理员是通过指令进行交互的，所以该线程的主要工作是，给出输入提示，输入指令，解释指令

```
def thread_of_admin(self):
    Ins = ins()
    while True:
        #命令提示
        str_ins = input('Admin@MaskSystem:{}$ '.format(self.current_district)).strip('/')
        #输入指令，若输入成功，则解释指令
        if Ins.Read(str_ins):
            self.admin_ins_explain(Ins)
        else:
            print('invalid command')
```

可以看到这里初始化了一个 ins 对象，并且调用了 read 函数，若 read 为 true，则进行指令解释。

该线程最重要的部分是对指令的解释，即函数 admin_ins_explain(ins)。admin_ins_explain(ins)检查传给他的 ins 对象的指令种类是什么，然后做出相应的动作。比如指令种类为 append，则相应的操作为新建一个 district 对象，加入到 DISTRICTS 字典之中。

```
if ins.OP == 'append' and len(ins.args)==1:
    if ins.args[0] not in self.DISTRICTS.keys():
        self.DISTRICTS[ins.args[0]] = district(ins.args[0])
    else:
        print('district {} already exist!'.format(ins.args[0]))
```

指令种类为 bookers，为管理员打印 all_bookers 中存放的所有在线 booker 信息

```
elif ins.OP == 'bookers':
    print('name\tID\tdistrict')
    for b in self.all_bookers:
        if b.online:
            print('{}\t{}\t{}'.format(b.name,b.ID,b.District))
```

指令种类为 erase，参数为 districtName，存放在 args[0]中，则 DISTRICTS 字典删去 key==args[0]的一项

```
elif ins.OP == 'erase' and len(ins.args)==1:
    if ins.args[0] in self.DISTRICTS.keys():
        self.DISTRICTS.pop(ins.args[0])
        if self.current_district == ins.args[0]:
            self.current_district = ''
        print('erase {} successfully'.format(ins.args[0]))
    else:
        print('district not exist!')
```

指令种类为 district，直接打印 DISTRICTS 中的信息，不截图，详细代码见 server.py 的第 50 行

指令种类为 enter，将 server 类中一个名为 current_district 的对象更改为要进入的 district 的名字，该对象总是记录当前所在的行政区的名字，若当前不在任何行政区，则该对象为一个空字符串。

```
elif ins.OP == 'enter' and len(ins.args)==1:
    if ins.args[0] in self.DISTRICTS.keys():
        self.current_district = ins.args[0]
    else:
        print("district not exist!")
```

进入行政区后的操作有 openNewround, list, handout, kickout, leave，解释器首先会检查是否在一个行政区内，即通过检查 current_district != ''来判断。

指令若为 openNewround，则将对应的 district 对象的 inRound 置为 True

```
elif ins.OP == 'openNewround':
    if self.current_district != '':
        cd = self.current_district
        if self.DISTRICTS[cd].inRound == False:
            self.DISTRICTS[cd].inRound = True
            print('open new round successfully.')
        else:
            print('already in Round!')
    else:
        print('You are not in a district!')
```

指令若为 list，则取出该行政区中所有 booker 的 ID(这些 ID 存放在 district 对象中的 BOOKERS 列表中)，并通过 ID 引用 all_bookers 列表中的 booker 对象，打印相关信息。

```
elif ins.OP == 'list' :
    if self.current_district != '':
        if self.DISTRICTS[self.current_district].inRound ==True:
            print('the numbers of booker: ',end='')
            print(self.DISTRICTS[self.current_district].BookerNum)
            print('the numbers of masks: ',end = '')
            print(self.DISTRICTS[self.current_district].maskNum)
            print('booker\tID\tneeds')
            #通过BOOKERS中存放的ID，引用all_bookers中的booker对象
            for i in self.DISTRICTS[self.current_district].BOOKERS:
                print(self.all_bookers[i].name,end='\t')
                print(self.all_bookers[i].ID,end='\t')
                print(self.all_bookers[i].needs,end='\n')
        else:
            print('not in Round')
    else:
        print('You are not in a district!')
```

指令若为 kickout，则 args[0]中为要进行的操作的 bookerID，将 district 对象中的 BOOKERS 中相应的项删除即可

指令若为 handout，则给该行政区 BOOKERS 中的所有 booker 发送一条消息，消息指明了 booker 成功预定到的口罩数量(当剩余口罩不足时，不等于 booker 的需求)，每处理完一个 booker，当前行政区的 maskNum 就要减去这个 booker 的需求量，代码见 server.py 的 99 行。

其余命令逻辑较为简单，就不一一介绍了，代码见 server.py 中的 admin_ins_explain 函数(line 25 开始)

2.与客户端交互的线程

首先，先明确“与客户端交互“的意思。这里客户端会将它接收的来自客户的大部分指令附加一些信息后直接发送给服务端，服务端抽取消息中的指令部分，就将它作为自己的一条指令来解释(就好像解释管理员的指令一样)，然后服务端将解释后的结果发送回客户端。所以该线程的工作就是：接收客户端消息，抽取出其中的指令，解释指令，发送响应结果

```
def thread_of_booker(self):
    Ins = ins()
    self.recv_socket.bind(('localhost',12345))
    while True:
        #接收客户端消息
        id,message,addr = self.recv_user()
        #得到该指令的来源地址
        Ins.from_addr = addr
        #将message所代表的指令通过Read转化为ins对象，并解释
        if Ins.Read(message):
            self.booker_ins_explain(id,Ins)
```

其中，客户端与服务端交互的消息格式(也可以说是本应用的协议)放在后面叙述，先来关心客户端传来的指令的解释。解释的思路和之前是一样的，即根据指令种类做出响应，这部分定义在 booker_ins_explain(bookerID,ins)中(server.py,line 148)

首先是指令种类为 login 时，此时根据协议(稍后详述)，会有 bookerID 为-1 以及附加了客户端的接收端口信息，args 数组为，args[0]=bookerName, args[1]=booker_recvPort

```
if bookerID == -1:
    if ins.OP == 'login' and len(ins.args) == 2:
        bname = ins.args[0]
        new_booker = booker()
        booker_recv_port = int(ins.args[1])
        From_addr =  list(ins.from_addr)
        From_addr[1] = booker_recv_port

        new_booker.addr = tuple(From_addr)
        new_booker.name = bname
        new_booker.ID = len(self.all_bookers)
        new_booker.online = True
        self.all_bookers.append(new_booker)
        self.send_user(new_booker.ID,'ID\n{}'.format(new_booker.ID))
```

若指令种类为 district，和之前管理员 district 指令设计类似，不过原本的 print 变成了朝对应的 booker 客户端发送消息。之前介绍过 booker 类，其中存有其客户端的接收端口，这样就能够借助 booker 对象来发送 udp 消息了。

包括 list，msg，join 等指令的设计和之前的管理员指令都是大同小异的，一般只有发送的对象有所不同，略过不表。

当接收到的指令为 exit 或者 logout 时，server 将该 booker 从 all_booker 中“去除“，实际上不是删除，只是将该 booker 的在线状态改为 false，并移除其在相应行政区中的记录

```
elif ins.OP == 'exit' or ins.OP == 'logout':
    dName = self.all_bookers[bookerID].District
    if dName in self.DISTRICTS.keys():
        self.DISTRICTS[dName].BOOKERS.remove(bookerID)
    self.all_bookers[bookerID].online = False
```

当接收到的指令为 setMaskNeeds 时，server 更改相应 booker 对象中存放的口罩需求量

```
elif ins.OP == 'setMaskNeeds' and len(ins.args)==1:
    self.all_bookers[bookerID].needs = int(ins.args[0])
    self.send_user(bookerID,'type:reply\x00set mask number successfully.')
```

三、客户端概述

客户端也包括两个线程，一个用于向服务端发送消息，另一个用于从服务端接收消息

```
def run(self):
    #接收消息线程
    recv_thread = Thread(target=self.thread_interactive_recv,args=())
    recv_thread.setDaemon(True)
    recv_thread.start()
    #发送消息线程
    self.thread_interactive_send()
```

1.发送消息线程

首先，要求客户 login，调用了 login 函数，login 函数接受了指令 login bookerName 后，将其发送给服务端，且附上了本客户端的接收消息端口(即另一个线程所用的端口)。当然也可以接收 exit 指令，直接退出程序。登录成功前(服务器返回 ID 前)，本客户端的 bookerID 默认为-1，bookerID 的信息其实也包含在消息中，-1 指示了发送消息的 booker 还未在服务器上有过记录。

```
def login(self):
    Ins = ins()
    while True:
        login_ins = input('please login with command "/login UserName":\n')
        if Ins.Read(login_ins) and Ins.OP == 'login':
            self.name = Ins.args[0]
            self.send_server(login_ins+' '+str(recv_port))
            break
        elif Ins.OP == 'exit':
            exit(0)
```

本线程的接下来的工作，实际上就是将用户输入的指令封装后发送给服务端，等待服务端返回结果(除了少数特殊指令在本地也进行处理)

其中一个比较重要的部分是，发送指令后，客户端如何等待服务端的响应，在本 pj 的设计中，服务端的响应是由另一个线程完成接收的，所以两个线程之间必须通信。此处采用的是设置一个全局变量 last_reply_got，每次发送完指令，就将其设置为 False，并且进入一个有时间上限的循环等待，另一个线程接收到消息之后，将 last_reply_got = True，本线程的循环等待会检测到这个改变，并退出等待；若超时，也自动退出等待。

```
while True:
    str_ins = input('{}@maskclient:{}$ '.format(self.name,self.District))
    if Ins.Read(str_ins):
        self.send_server(str_ins)
        if Ins.OP == 'exit':
            exit()
        elif Ins.OP == 'logout':
            self.clear()
            self.login()
        else:
            self.last_reply_got = False
            start_wait_time = time()
            #循环等待3s
            while self.last_reply_got != True and time()-start_wait_time<=3:
                pass
            #若3s内无回应
            if self.last_reply_got == False:
                print('server no reply.')
```

2.接收消息线程

本线程首先等待服务端返回的 bookerID，随后进入正常工作流程：等待一个消息，检测该消息是指令的响应，还是单纯的讯息(来自管理员或别的用户)，若是响应，则直接打印，若是讯息，则存放在客户端维护的一个邮箱(history_text)中，方便用户随时查看(client.py, line 95)

```python
def thread_interactive_recv(self):

    while True:     #waiting for ID
        message = self.recv_server()
        if 'ID\n' in message:
            self.ID = int(message.split('\n')[1])
            break
    while True:
        message = self.recv_server()
        if 'type:text\n\x00' in message:
            self.newest_text = message
            self.history_text.append(message)
        elif 'type:reply\x00' in message:           #if message is the reply of client's command, print the reply
            if 'in district ' in message:
                self.District = message.split('in district ')[1]
            self.last_reply_got = True
            print(message.strip('type:reply\x00'))
```

四、交互使用的消息格式

1.服务端发往客户端的消息格式

前面已经提到，服务端发往客户端的消息分为两种，一种是服务端对客户指令的响应(reply)，另一种是讯息，即管理员发送的或者其他 booker 发送的短讯(text)。

于是，服务端发往客户端的消息会有一个 type 首部，即消息格式为：

“type:text\n”+’\0’+text_content 或 “type:reply” + “\0”+reply_content

注意，前者有一个回车符，不过只是方便客户端显示消息时换个行而已。‘\0’作为分割符。

基本上在 booker_ins_explain()函数中的响应动作中，都能看到这种格式消息的构造。这样客户端就能识别消息到底是一个 reply 还是一个 text，从而进行相应操作(打印或者放进邮箱)

2.客户端发往服务端的消息格式

首要的，服务端需要知道给他发消息的客户是谁，因此消息中要包含客户的 ID。同时，服务端要知道，如果它要对客户进行响应，应该朝哪个端口进行发送，因为 recvfrom 只能获得客户的发送端口，不能获得客户的接收端口，所以接收端口也要附加在消息中。

其中，ID 是每次都要附加的信息，而接收端口只需要在第一次交互时附加，即在 login 的指令消息中附加。因此，消息格式为：

若指令为 login，’-1’ + ‘\n’ + ins_content + recv_port

若指令不为 login， bookerID + ‘\n’ +ins_content

此处用’\n‘作分隔符，是因为指令中不会出现回车也不会出现其他不可打印字符，因此就不使用’\0’了。

客户发送消息的封装工作由函数 send_server(message)完成(client.py, line 37)

```python
def recv_server(self):
    message,addr = self.recv_socket.recvfrom(1024)
    message = message.decode()
    return message
```