



四川大学

计算机网络 项目开发报告

题 目 基于泛洪查询的文件共享

学 院 建筑与环境学院

专 业 工程力学与软件交叉

学生姓名 尤敬尧

学号 2019141470415 年级 2019

指导教师 宋万忠

二〇二一年 六月 二十六 日


目录

- 一、 功能及说明
- 二、 实现功能的流程及说明
- 三、 项目开发中遇到的问题和体会

一、 功能及说明：


本项目参考了 Github 上的两份源码，在此基础上进行了改进：

 [Cisen07/Query-Flooding-Based-Resource-Sharer](#)
Project of Computer Network
● Python Updated on 30 Nov 2019

 [Jiangluyu/query-flooding-based-resource-sharer](#)
computer network class design
☆ 6 ● Python Updated on 19 Dec 2019

本项目实现了不同对等方之间的资源共享，在每个对等方自己的文件夹中通过修改配置的 ini 文件即可构建不同节点之间的连接关系，为了测试者测试，本项目维护了三个对等方之间的联系，通过模拟网络中的拓扑结构，模拟不同对等方的操作。

在同一台主机的测试上，我们通过设置不同节点的端口号来模拟不同的对等方的操作，以下为每个对等方自身的 ini 文件：

 config.ini - 记事本 文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H) [Peer-0] ID=1 ip_addr= 127.0.0.1 port_number= 10810 share_dir=1/ ttl=4 [Netpeers] peer_ID=[2] peer_addr=[127.0.0.1] peer_ports=[10811]	[Peer-0] ID=2 ip_addr= 127.0.0.1 port_number= 10811 share_dir=2/ ttl=4 [Netpeers] peer_ID=[3] peer_addr=[127.0.0.1] peer_ports=[10812]	[Peer-0] ID=3 ip_addr= 127.0.0.1 port_number= 10812 share_dir=3/ ttl=4 [Netpeers] peer_ID=[4] peer_addr=[127.0.0.1] peer_ports=[10813]
1	2	3

本项目通过 config 可以查看本地节点的信息，包括 ip，端口号，ID，ttl，文件夹列表等；

通过 help 选项提示用户当前操作；get 1.txt(2.txt)(3.txt)分别通过不同的对等方实现文件的查询、共享和下载。

Exit 退出当前系统

Modify 修改当前对等方的信息

利用 md5 加密文件，保证传输过程中文件内容没有改变。

二、 实现功能的流程和说明：

以下为本人整理的源码的所有类与方法的详细信息：

一、 config.ini，列出了对于每个节点的基本信息：节点编号[Peer-0]、ip 地址、服务端端口号、客户端端口号、分享文件以及该节点维护的邻接节点的列表（可从其邻接对等方 1 查询文件）

对于对等方之间的关系，该项目简化了对等方之间建立连接的过程，在配置文件 config.ini 中分配好了各个节点的邻接对等方。

```

≡ config.ini
1  [Peer-0]
2  ip_addr = 127.0.0.2
3  server_port = 10800
4  client_port = 10810
5  share_dir = 0/
6  peer_list = [1, 2]
7
8  [Peer-1]

```

二、

Config.py:导入了 configparser 库;

首先在 configparser 模块读取 config.ini 文件, 并设置 ttl 为 0:

```

__cf = configparser.ConfigParser()
__ttl = 0

```

定义 get_attr(self,i):函数

获取对等方的信息并打包以字典的形式返回。

```

: return: dict {'ip_addr': ip
def get_attr(self, i):

```

```

while True:
    try:
        peer = dict()

```

```

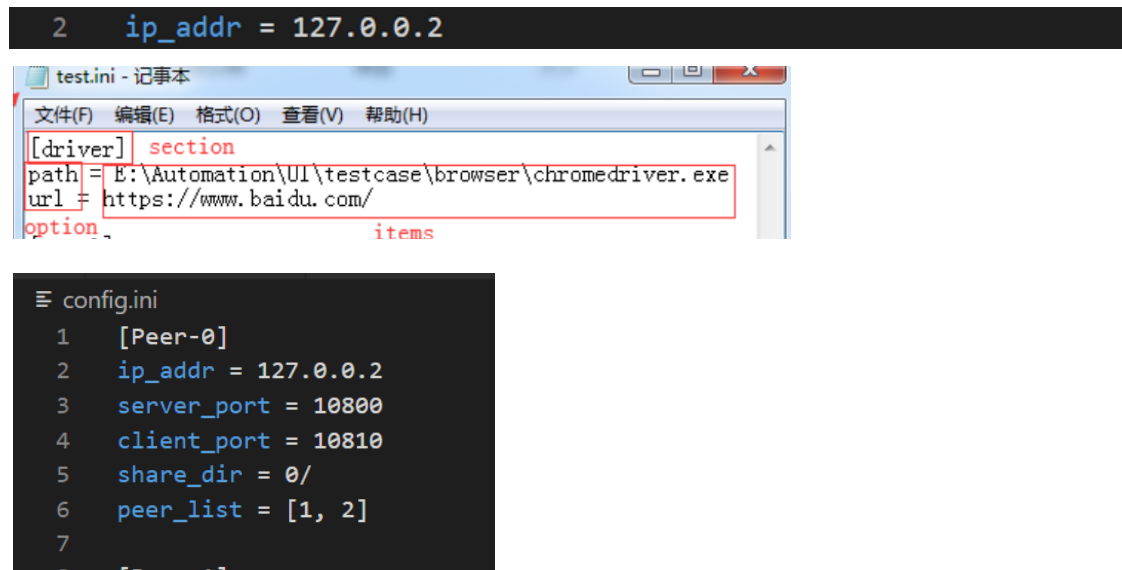
peer = dict()
peer['ip_addr'] = self.__cf.get("Peer-%s" % i, "ip_addr")#cf.get(section, option)
peer['server_port'] = self.__cf.get("Peer-%s" % i, "server_port")
peer['client_port'] = self.__cf.get("Peer-%s" % i, "client_port")
peer['share_dir'] = self.__cf.get("Peer-%s" % i, "share_dir")
peer_str = self.__cf.get("Peer-%s" % i, "peer_list")
peer_str = peer_str[1:len(peer_str) - 1]
peer_list = peer_str.split(',')

```

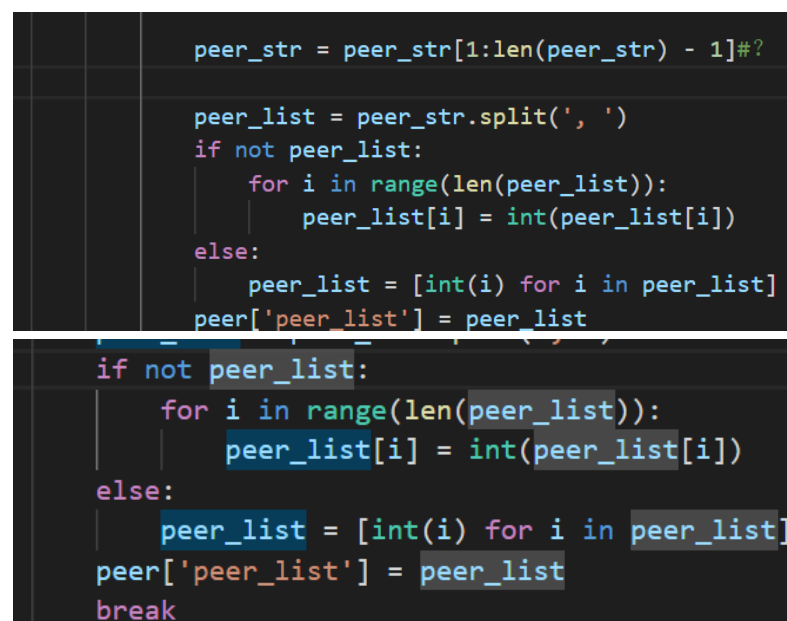
cf.get(section,option): 得到 section 中 option 的值, 返回 string 类型的结果

如下图, 将对等方 peer 的各个 attribute 分别赋予在已配置好的.ini 文件中的值;

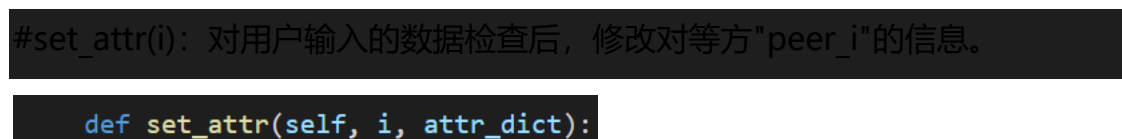
比如, ip_addr 赋予对于 peer 的编号模 i 后的对等方的 ip_addr 的 option, 即对应的:



将字符数组 `peer_list` 对应的对等方添加进去，此时将 `peer_str`(字符串放到字符数组 `peer_list` 中，之后再将每个字符强制类型转换为 `int` 型存储)：



定义 `set_attr` 函数:



首先 `self.__init__()`，初始化 `self`，读取 `config.ini` 文件，并设置 `ttl` 为 2:

```
def __init__(self):
    self.__cf.read("config.ini")
    self.__ttl = 2
```

将当前 attr 中的信息与原始 attr 进行比对，如果发现 diff_val 的值为空，则返回 “No modification found”

```
print(origin_attr)
diff_val = [(k, origin_attr[k], attr_dict[k]) for k in
            # e.g. [('diff_key', original_attr[diff_key], attr_dict[diff_key])
            if not diff_val:
                return "No modification found"
            for t in diff_val:
                ori = []
```

Ps:未写完该函数后续功能

定义 modify 方法，将对等方"peer_i"的修改后信息写入配置文件

通过 flag 和 data 及 line 来定位到每个 peer 的每一行，将修改后的信息写入配置文件

在其中调用的 find 方法是检测是否包含 “=”，若包含则返回开始的索引值，否则返回-1；replace()方法把字符串中的旧字符串替换成新字符串，返回生成的新字符串。

```

with open("config.ini", "r") as f:
    for line in f:
        if flag > 0:
            flag -= 1

        if "Peer-%s" % i in line:
            flag = 6

        if flag == 5:
            line = line.replace(line[line.find("=") + 2:len(line) - 1], attr_dict["ip_addr"])
        elif flag == 4:
            line = line.replace(line[line.find("=") + 2:len(line) - 1], attr_dict["server_port"])
        elif flag == 3:
            line = line.replace(line[line.find("=") + 2:len(line) - 1], attr_dict["client_port"])
        elif flag == 2:
            line = line.replace(line[line.find("=") + 2:len(line) - 1], attr_dict["share_dir"])
        elif flag == 1:
            line = line.replace(line[line.find("=") + 2:len(line) - 1], attr_dict["peer_list"])

        data += line
    with open("config.ini", "w") as f:
        f.write(data)

```

将 peer 对应的每个属性修改后，再通过 f.write(data)将 data 写入文件中。

在 config 类中最后三个函数

get_ttl(self):

#get_ttl(): 返回 time to live 值

get_peer_num():

#get_peer_num(): 获得当前网络环境中对等方的数量。

__init__(self):

self.__cf.read("config.ini")

初始化 self

```

def get_ttl(self):
    #get_ttl(): 返回time to live值
    return self.__ttl

    @staticmethod
def get_peer_num():
    #get_peer_num(): 获得当前网络环境中对等方的数量。
    with open("config.ini", "r") as f:
        count = 0
        for line in f:
            count += 1
    return int((count + 1) / 7)

def __init__(self):
    self.__cf.read("config.ini")
    self.__ttl = 2

```

三、connection.py 中导入的库：import socket

import threading

import os

import struct

import json

import config

import zipfile

import filemd5

在 connection 类中，首先定义了自身属性、当前对等方所有邻居的信息：

```

__num = None

__ip_addr = None

```



```
__server_port = None

__client_port = None

__peer_list = None

__share_dir = None

__peer_num = 0

__query_res = dict()

__path_list = dict()

__request_flag = dict()

__peer_attr = []
```

将服务器收到的命令定义为_cmd,并初始化源 ip 及端口号分别为 127.0.0.1 及 None:

```
__cmd = []

__source_ip = '127.0.0.1'

__source_port = None
```

定义函数:

```
def set_ip(self, ip_addr):

    #设置节点地址

def set_server_port(self, server_port):

    #设置节点的服务端端口
```

```
def set_client_port(self, client_port):
```

```
    #设置节点的客户端端口
```

```
def set_peer_list(self, peer_list):
```

```
    #设置节点的邻居列表
```

```
def set_share_dir(self, share_dir):
```

```
    #设置节点可以分享的文件
```

定义查询函数 query:

os.listdir 将 root 路径下的文件及文件夹列表赋予 items，之后使用 for 循环遍

历 root 路径下的每个文件，通过 path.join 连接每条路径，之后通过

path.split 将 “\” 或 “/” 将路径分开，检查是否有文件名对应 filename。若

对应，则将其 query_res[filename]置为 1，并将 path_list 中的 “\” 改为

“/” 。

注：“/” 是虚拟路径，“\” 是实际物理路径的写法，“//” 一般用于网络协议中，“\\” 一般用于局域网之间的互访

elif os.path.isdir()用于返回一个由文件名和目录名组成的列表，需要注意接收的参数必须是绝对路径。我猜测这里应该是先检查 root 下的文件列表，看看有无匹配的 filename,对于 root 下的文件夹，采用递归查询，故该函数比对了 filename 与该对等方 root 路径下的所有信息对应的文件。

对应于 self.query 里的两个参数分别是 def 中的后两个参数，第一个参数默认为实例变量 self（类中定义函数）

```
def query(self, root, filename):

    items = os.listdir(root)

    for item in items:

        path = os.path.join(root, item)

        if path.split("\\")[-1] == filename or path.split('/')[ -1] == filename:

            self.__query_res[filename] = 1

            self.__path_list[filename] = path.replace("\\", '/')

        elif os.path.isdir(path):

            self.query(path.replace("\\", '/'), filename)
```

定义函数 tcp_handler:

```
def tcp_handler(self, conn, addr):
```

```
res = conn.recv(1024)

self.__cmd = res.decode('utf-8').split()

首先判断 cmd[0]的位置是否是 get 方法，如果是，则更新 self 的 ttl(?)

并将 cmd 中的 cmd[2]、cmd[3]中的值分别赋予 self 的源端口号和源 ip，

self.__query_res[self__cmd[1]]被置为 0

if self.__cmd[0] == 'get':

    res = self.update_ttl(res)
```

```
self.__source_port = self.__cmd[2]

self.__source_ip = self.__cmd[3]

self.__query_res[self.__cmd[1]] = 0

self.query(self.__share_dir, self.__cmd[1])

print("%s: Query %s at self" % (self.__num, self.__cmd[1]))
```

之后开始调用 self 自己的 query 函数比对其分享文件夹的 cmd 中的第二个元组，并输出 “self.num:Query self.__cmd[1] at self” 通知已将此文件在对应的节点找到

```
self.query(self.__share_dir, self.__cmd[1])

print("%s: Query %s at self" % (self.__num, self.__cmd[1]))
```

如果本地未找到 self.__query_res[self.__cmd[1]]==0，则查询邻居节点：

如果 self.cmd 的最后一个元组强制转换成 int 型 >= 0 (最后一个值是该节点的 ttl，需要保证其 ttl >= 0)，则代表该节点仍可以继续查找，Querying neighbors: 开始进行迭代查询，如果在 self 的 peer 表中的 i 号节点对应的服务器端口号 (server port) == self.cmd[2]，则跳出本次循环；

否则即输出查询几号节点在 query neighbor，并调用 self.tcp_client_notice 来建立 self 和 i 节点的 tcp 连接，保存从 i 节点中查找到的所需文件。

在 tcp_client_notice 中：

程序在使用 socket 之前，必须先创建 socket 对象，可通过该类的如下构造器

来创建 socket 实例：首先利用 socket 套接字建立两个进程之间的连接，
AF_INET 是因特网地址簇名，SOCK_STREAM 用于建立 TCP 套接字；

Connect:作为客户端使用的 socket 调用该方法连接远程服务器。

socket.send(bytes[, flags]): 向 socket 发送数据，该 socket 必须与远程 socket 建立了连接。该方法通常用于在基于 TCP 协议的网络中发送数据。

If 语句判断第一个空格分割开后前面的字符串是否为 request，若是，则通过自己定义的_save 函数，将该文件拷贝到本地：

```
def tcp_client_notice(self, ip, port, msg):  
    tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    tcp_client.connect((ip, int(port)))  
    tcp_client.send(msg.encode())  
    if msg.split()[0] == 'request':  
        self._save(tcp_client)  
    tcp_client.shutdown(2)  
    tcp_client.close()
```

save 函数：

将连接每四个字节放入一个缓冲区，给 obj;

```
obj = conn.recv(4)  
  
header_size = struct.unpack('i', obj)[0]
```

Header_size=struct.unpack('i' ,obj)[0]: struct.unpack(fmt,string), 用 unpack 解包,返回一个由解包数据(string)得到的一个元组(tuple), 即使仅有一

个数据也会被解包成元组。其中 `len(string)` 必须等于 `calcsize(fmt)`，这里面涉及到了一个 `calcsize` 函数。`struct.calcsize(fmt)`：这个就是用来计算 `fmt` 格式所描述的结构的大小。这里描述的是 `'i'` 的格式大小。

之后我们利用 `header_bytes` 来按照 `header_size` 的缓冲区大小接收内容。

接下来解析包头，先将 `header_bytes` 按照 `utf-8` 解码，`json.loads()` 函数是将字符串转化为字典，再赋予 `header_dic`，`total_size` 是字典中的 `file_size` 项，`filename` 是对应的 `filename` 项，`cur_md5` 是当前字典中存储的 `md5` 的值。

```
# 接收头

header_bytes = conn.recv(header_size)


# 解包头

header_json = header_bytes.decode('utf-8')

header_dic = json.loads(header_json)

total_size = header_dic['file_size']

filename = header_dic['filename']

cur_md5 = header_dic['md5']
```

接下来接收数据：

首先 `with open` 以二进制打开一个文件 `filename`（本地需要拷贝的目标文件的文件名，只是里面内容为空），读取方式为 `"wb"`，`wb`：以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。本地没有 `filename` 时，程序会为我们自动创建该文件名的空文件，将 `recv_size` 置为 0，利用 `while` 语句，当 `recv_size < total_size` 的时候，

将 recv 缓冲区每次通过连接接收 1024Bytes 赋予 res,再将 res 中的内容写入本地的 filename 文件中, recv_size 每次长度+len(res)的长度, 最后输出 total_size (total_size) 是多少, already download(recv_size): %.

之后比较本地文件的 filemd_5 与 cur_md5 的值进行比较, 如果返回值为 1, 则比较成功, 之后利用导入的 zipfile 包:

```
z = zipfile.ZipFile("%s%s" % (self.__share_dir, filename), 'r')
z.extractall("%s" % self.__share_dir)
z.close()
```

将该文件解压, 并关闭 zipfile。

如果 md_5 的值不同, 则打印:传输过程中文件损坏

保证了传输过程中文件的一致性, 即下载过程中是否会被别的进程影响。

在损坏之后, 通过 while True 语句调用 os.remove 方法删除本地的 filename 文件 (md_5 值不匹配, 导致文件错误。)

```
while True:
    try:
        os.remove("%s%s" % (self.__share_dir, filename))
        break
    except:
        continue
```

调用完 self_save 函数后继续执行 tcp_client_notice 函数, tcp_client.shutdown(2), 禁用被封装的 Socket 上的发送和接收数据;

之后 tcp.client.close(),关闭被封装的 Socket 连接并释放所有关联此 Socket 的资源.

Notice 函数调用完后, 我们继续返回上一级:如果 self 没找到资源, 则它与其余的每个 peer 节点建立 tcp 连接。

```
if self.__query_res[self.__cmd[1]] == 0:
```

与其对应的 else 代表本地找到对应的文件，并输出

```
msg = "found %s at %s %s %s" % (self.__cmd[1], self.__num,  
self.__ip_addr, self.__server_port)
```

并利用 tcp_notice 建立连接，向请求源发送成功的消息。

如果 __cmd[0] 的位置为 found，则代表对应的文件已经找到，则向 x 的 sever 端发送请求，将 msg 开头更新为 request，之后就可以将通过 conn 连接将 server 的文件内容拷贝到 self 上。

如果对应的 __cmd[0] 的位置对应的值为 request，则直接将其 request_flag 字典对应的文件名的元组的值置为 1，之后通过 send，向请求源发送源文件。

其他 cmd 命令被认为是非法输入，直接舍弃：

```
# 过滤其它命令  
else:  
    msg = "invalid content"  
    conn.send(msg.encode())
```

总的来说，tcp_handler 的功能即利用了深度优先搜索，首先查看本地文件，若没有匹配的文件则深度优先搜索其 peer 节点，知道查找到该文件后，保存文件所在节点 x 的 peer_num, 源 ip, 端口号，并递归向初始节点发送找到成功的消息，直到 self 节点收到该消息，通过 conn, x 向请求源发送该文件。

下面定义 tcp_server 函数：

实例化 socket，利用 socketopt：任意类型、任意状态套接字的设置选项值

```
tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
tcp_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

bind() 函数把一个地址族中的特定地址赋给 socket，tcp_bind 将本地 ip 与

self.sever_port 绑定在一起

```
tcp_server.bind(("127.0.0.1", self.__server_port))
```

作为一个服务器，在调用 socket()、bind()之后就会调用 listen()来监听这个 socket，如果客户端这时调用 connect()发出连接请求，服务器端就会接收到这个请求。

```
tcp_server.listen(5)
```

listen 函数的第一个参数即为要监听的 socket 描述字，第二个参数为相应 socket 可以排队的最大连接个数。

之后 try 方法：

```
conn,addr=tcp_server.accept() #服务端阻塞,等待客户端来连接
```

conn 用于双向连接，客户端地址是 addr

调用线程 thread，第一个参数是线程函数变量，第二个是数组变量的参数

```
try:
    conn, addr = tcp_server.accept()
except ConnectionAbortedError:
    print("server except")
    continue
t = threading.Thread(target=self.tcp_handler, args=(conn, addr))
t.start()
```

定义_send 函数，传入参数为 self，conn，以及文件名

```
def __send(self, conn, filename):
```

filepath 设置为 self 的 filename 的 path_list，之后调用 find 方法查看 filename 中是否含有“.”，将 filename 换成 filename “.” 之前的名字 + “.zip” ，

之后创建一个 zipfile 对象，调用 isdir 判断 filename 路径是否是一个目录。若

是，则将该路径下的所有文件全部写入 z:

```
if os.path.isdir(filepath):  
    for d in os.listdir(filepath):  
        z.write(filepath + os.sep + d, d)  
else:
```

Else, 将该文件的路径加上 filename 的完整名写入 z,z.close().

之后初始化 header_dic:

```
header_dic = {  
    'filename': filename,  
    'md5': filemd5.get_file_md5(filename),  
    'file_size': z.infolist()[0].file_size  
}  
  
header_json = json.dumps(header_dic)  
header_bytes = header_json.encode('utf-8')
```

其中 json.dumps 将 python 对象编码成 Json 字符串,json 按照 utf-8 加码后

基于 header_bytes

```
# 打包文件头  
conn.send(struct.pack('i', len(header_bytes)))  
  
# 发送头  
conn.send(header_bytes)
```

(为什么将头打包, 与数据端分开分别发送?)

之后发送数据:

读写二进制文件

```
# 发送数据

send_size = 0

with open(filename, 'rb') as f:

    for b in f:

        conn.send(b)

        send_size += len(b)

while True:

    try:

        os.remove(filename)

        break

    except:

        continue
```

???

定义 update_peer_attr(更新 self 的 peer 的属性)

初始化 config 对象, l1 为字典列表, 遍历 peer_list, 将 i 对应的属性通过 get_attr(i)方法先添加到了 l1 中, 之后将 l1 赋予 self._peer_attr, 并返回该属性 List.

```
def update_peer_attr(self):

    conf = config.Config()

    l1 = []

    for i in self.__peer_list:

        l1.append(conf.get_attr(i))
```

```
self.__peer_num += 1

self.__peer_attr = l1

return self.__peer_attr
```

定义 update_ttl,

Msg 按照空格解码, msg 的最后一个值-1 再强制类型转化为 str 类型, 之后将该信息赋予 new_msg,最后返回 new_msg

```
def update_ttl(self, msg):

    msg = msg.decode().split()

    msg[-1] = str(int(msg[-1]) - 1)

    new_msg = " ".join(msg)

    return new_msg #???
```

后面几个简单的函数对应的就是返回相应的值

```
def get_timeout_flag(self):

    return self.__timeout_flag


def get_peer_num(self):

    return self.__peer_num


def get_peer_list(self):

    return self.__peer_list


def get_num(self):
```

```
return self.__num
```

四、filemd5.py

定义函数 `get_file_md5`，如果该文件路径下的对应的 file 不是文件，则 return，
(hashlib 是对二进制数进行加密的)

每次读取 4096 个字节，计算相应的 md5 值，最后返回该 md5 计算值

```
def get_file_md5(file):  
    if not os.path.isfile(file):  
        return  
  
    my_hash = hashlib.md5()  
  
    with open(file, 'rb') as f:  
        while True:  
            b = f.read(4096)  
  
            if not b:  
                break  
  
            my_hash.update(b)  
  
    return my_hash.hexdigest()
```

定义函数 `compare_file_md5`:

```
def compare_file_md5(file, md5):  
    res = 0  
  
    cur_md5 = get_file_md5(file)  
  
    if md5 == cur_md5:  
        res = 1
```

```
return res
```

比较当前 md5 的值与 file 的 md5 的值是否匹配，如果匹配将 res 置 1 后返回。

五、process.py

定义 tcp_server:

实例化对象 config 对象，ID 对应的节点相应的属性值

实例化 peer_s 链接，调用 connection 类中的配置函数配置相应的 num、ip 、server_port、client_port、share_dir、peer_list，之后更新邻居节点的属性

```
def tcp_server(ID):  
    init_config = config.Config()  
    peer = init_config.get_attr(ID)  
    peer_s = connection.Connection()  
    peer_s.set_num(ID)  
    peer_s.set_ip(peer['ip_addr'])  
    peer_s.set_server_port(peer['server_port'])  
    peer_s.set_client_port(peer['client_port'])  
    peer_s.set_share_dir(peer['share_dir'])  
    peer_s.set_peer_list(peer['peer_list'])  
    peer_s.update_peer_attr()  
    peer_s.tcp_server()
```

定义 tcp_client 函数:

同样初始化 role_num 做为客户端的各种信息，对于 role_num 的所有 peer，client.query 将该节点与其列表中维护的 peer 节点建立链接，查询 filename

```
def tcp_client(role_num, filename):

    init_config = config.Config()

    peer_ccon = init_config.get_attr(role_num)

    peer_c = connection.Connection()

    peer_c.set_ip(peer_ccon['ip_addr'])

    peer_c.set_server_port(peer_ccon['server_port'])

    peer_c.set_client_port(peer_ccon['client_port'])

    peer_c.set_share_dir(peer_ccon['share_dir'])

    peer_c.set_peer_list(peer_ccon['peer_list'])

    peer_attr = peer_c.update_peer_attr()

    for i in peer_attr:

        print("client query %s" % i['server_port'])

        peer_c.tcp_client_notice(i['ip_addr'], i['server_port'], "get %s %s %s %s"

% (filename, peer_ccon['server_port'], peer_ccon['ip_addr'], init_config.g

et_ttl()))
```

六、

Main.py

```
import multiprocessing
```

```
import os

import process

import config

import multiprocessing as mp
```

导入了以上库：

Multiprocessing.freeze_support()

因为.ini 文件初始了 7 个 peer 节点，我们定义 attr_num=7

之后通过赋值语句将 role、opt 分别赋值，如果 opt 最开始的字符串是

“get” ，则调用 config 类获得网络中对等方的数量，

之后利用多线程库自带的 pool 函数（进程池）：

同时操作多个文件目录，或者远程控制多台主机，并行操作可以节约大量的时间。当被操作对象数目不大时，可以直接利用 multiprocessing 中的 Process 动态生成多个进程。

Pool 可以提供指定数量的进程供用户调用，当有新的请求提交到 pool 中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来它。

For 循环中：

```
for i in range(attr_num):

    p.apply_async(process.tcp_server, args=(i,))
```

apply_async 是异步非阻塞式，不用等待当前进程执行完毕，随时跟进操作系统调度来进行进程切换

q 被赋值的语句用来调用其 tcp_client 方法，而参数为 role 和 opt 的第二个值，之后 start 开始，join 用于等待子进程结束，子进程 close，父进程停止，父进程等待子进程结束后退出。

```
print('Waiting for all subprocesses done...')

    q = mp.Process(target=process.tcp_client, args=(role, opt.split()[
1]))

    q.start()

    q.join()

    q.close()

    p.terminate()

    p.join()

    print('All subprocesses done.')
```

elif 对应的操作时 config 的话，就打开 ini 文件，读取对应的节点的信息：

```
elif opt.split()[0] == 'config':

    with open("config.ini", 'r') as f:

        line_num = 0

        while True:

            line = f.readline()

            line_num += 1

            if attr_num * int(role) <= line_num <= attr_num * int(role) +

6:

                print(line, end="")
```

```
if line_num > attr_num * int(role) + 6:  
  
    break
```

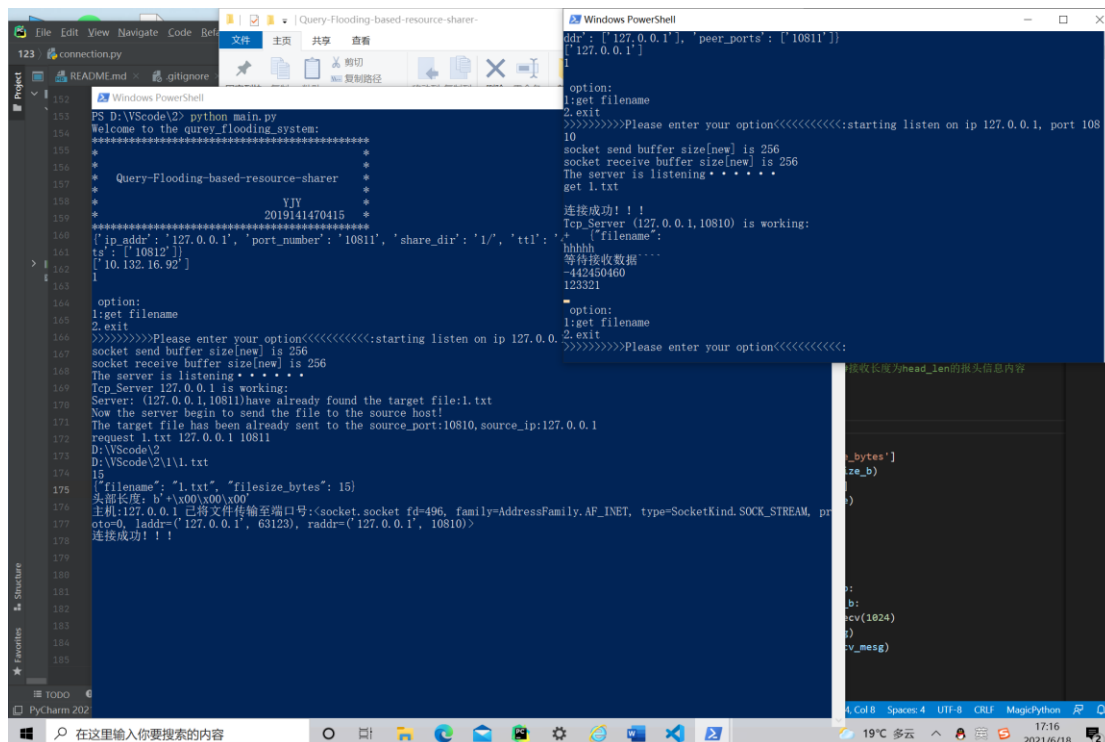
elif 对应的操作是 modify（修改对等方的属性信息），将对应的对等方的信息打包以字典形式返回，之后匹配相应的修改属性进行修改。

```
elif opt.split()[0] == 'modify':  
  
    mod_conf = config.Config()  
    attr_mod = mod_conf.get_attr(role)  
  
    if opt.split()[1] == 'ip':  
        attr_mod['ip_addr'] = opt.split()[2]  
  
    elif opt.split()[1] == 'serverport':  
        attr_mod["server_port"] = opt.split()[2]  
  
    elif opt.split()[1] == 'clientport':  
        attr_mod['client_port'] = opt.split()[2]  
  
    elif opt.split()[1] == 'sharedir':  
        attr_mod['share_dir'] = opt.split()[2]  
  
    elif opt.split()[1] == 'peerlist':  
        attr_mod['peer_list'] = opt.split()[2].split('/')  
  
        print(attr_mod['peer_list'])  
  
        attr_mod['peer_list'] = [int(i) for i in attr_mod['peer_list']]  
  
    else:  
  
        print("Input Error. Please check again.")  
  
    print(mod_conf.set_attr(role, attr_mod))
```

elif 对应的操作是 help，就打开 help.txt 显示相应文件内容：

```
elif opt.split()[0] == 'help':  
  
    with open("help.txt", 'r', encoding='utf-8') as f:  
  
        while True:  
  
            line = f.readline()  
  
            print(line, end="")  
  
            if not line:  
  
                break  
  
elif opt.split()[0] == 'exit':  
  
    exit()  
  
else:  
  
    print("Invalid input. Please check your input again.")
```

注意：在 TCP 的 connect 方法中，我们需要将存储于.ini 文件中的端口号转化成 int 型!!!



发送的头部与接收的头部的值不一样：

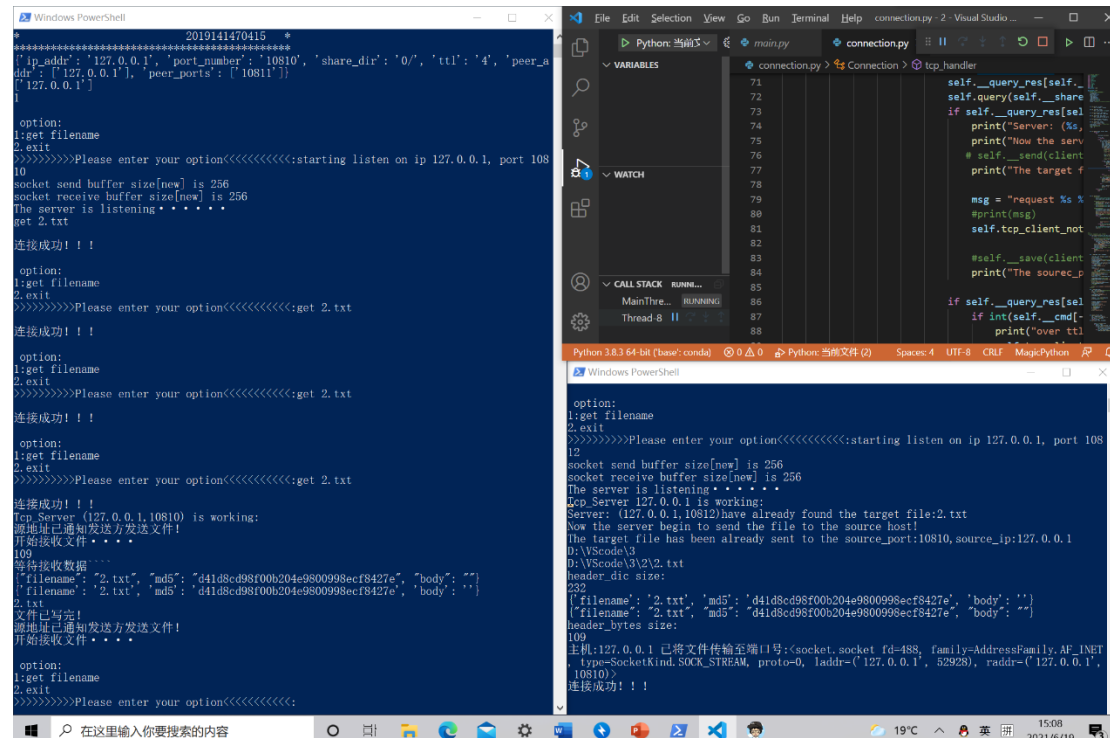
遇到了粘包问题：即 TCP 按字节流的方式发送数据，第一次我们要接收文件的一些属性，例如包的大小和文件名，md5 值等等。在这之后我们将文件的内容传到客户机对服务器的套接字里，但是在服务器接收端，我们通过 `conn.recv()` 一次性接收的字节流的数量是不确定的，无法通过每次的配置检验包头的大小，所以我将文件的各种信息及文件体写于一个字典中：这样一来只要保证接收的字节小于 1024 即可，这里调用了 `json` 库中的 `json.dumps` 将字典转化成字符串存储，之后将头部用“utf-8”编码，之后将头部传到套接字之中，之后头部将内容接收内容解码，之后调用 `json.loads` 将其解码后将内容还原成字

```
header_dic = {
    'filename': filename,
    'md5': md5.get_file_md5(filepath),
    'body': data1
}#已测试filename及filesize_bytes是正确的
```

```
print(header_json)#打印头部内容
header_bytes=header_json.encode('utf-8')#encode头部
#发送头的长度
print(["header_bytes size:"])
print(sys.getsizeof(header_bytes))
conn.send(header_bytes)
# 发送数据

print("主机:%s 已将文件传输至端口号:%s"%(self.__ip_addr,conn))
#send正确

def __recv(self,conn):
```



The screenshot displays a Windows desktop environment. On the left, a Windows PowerShell terminal window shows the output of a Python script. It indicates that the server is listening on IP 127.0.0.1, port 10810, and has successfully received a file named '2.txt' from a client. The output includes details about the file's MD5 hash and the connection parameters. On the right, a Visual Studio IDE window shows the Python code being executed. The code is part of a class that handles TCP connections and file transfers. The 'connection.py' file is open, showing the 'tcp_handler' class and its methods. The 'Run' button is visible, indicating the application is running.

三、项目开发中遇到的问题和体会

问题：较难实现的是在服务器接收到对等方请求后自身调用 tcp_notice 与源节点建立 tcp 连接时的逻辑关系，在接收文件时，由于 accept 函数阻塞的位置没有梳理清楚，调试了近一个星期对于接受文件无法写入的问题。

体会：对于 tcp 的建立过程有了更加清晰的认识，也在完善代码的时候收获了更多的 python 语言的知识。