#####################

##Stack\_Sender(EW\_Sender)

#####################

*#  三通道一起处理*

        temp\_file = self.img\_path

        rgb\_list = ['r', 'g', 'b']

        temp\_file\_list = [temp\_file + '\_' + ii *for* ii *in* rgb\_list]  *#3个通道rgb名字而非内容的列表*

# 创建一个列表，列表内容是带有各个通道 ‘RGB’后缀的str

#No1然后开始以下的图像预处理

第一步是，获取待处理图像的规格

img = load\_img(self.img\_path)           *#加载图像，返回的应该是一个图像对象*

(width, height) = img.size              *#图像大小，二维的维度，width和height不用在意*

mat\_r = np.empty((width, height))       *#创建三个通道的数据阵*

mat\_g = np.empty((width, height)) #width height是指各行各列上面的像素个数

mat\_b = np.empty((width, height))

第二步是，遍历图像中每个像素，然后获取像素中rgb3个通道的数据，得到3个单色通道图像数据

*for* i *in* range(width):

*for* j *in* range(height):

[r, g, b] = img.getpixel((i, j))

             mat\_r[i, j] = r *#分别读入三个通道的像素值*

             mat\_g[i, j] = g *#原来每个像素点都包含ri+gi+bi*

             mat\_b[i, j] = b *#现在是将每个像素的rgb通道分开*

第三步是，对3个单色通道数据分别进行小波变换生成3个对应的小波系数矩阵，然后对这三个小波系数矩阵进行多级树集合分裂编码，得到3各对应通道的二进制编码。都分别存在相应的列表中

self.img\_mat = [mat\_r, mat\_g, mat\_b]    *#rgb三个通道的列表*

self.dwt\_coeff = [func\_DWT(ii) *for* ii *in* self.img\_mat]  *#按顺序处理rgb数据，然后3个通道的生成小波系数矩阵的列表*

self.spiht\_bits = [spiht\_encode(ii, self.eng) *for* ii *in* self.dwt\_coeff] *#3个通道列表分别进行多级树集合分裂编码，输出3个通道的二进制编码列表*

第四步是，把这3个通道的二进制编码存到对应的文件中，文件名字是带后缀的temp\_file\_list

a =[code\_to\_file(self.spiht\_bits[ii],

    temp\_file\_list[ii], # 已经存在这3个文件中，文件名存在这个列表里

*add\_to*=self.fountain\_chunk\_size / 3 \* 8) # 编程细节是这里是最后一个参数，不需要逗号

*for* ii *in* range(len(rgb\_list))] # 为了方便迭代所有才这么写

第五步是，将这3个通道的数据合在一起，形成准备要发送的数据

self.m, \_chunk\_size = self.\_321(temp\_file\_list, *each\_chunk\_size*=self.fountain\_chunk\_size / 3)

这里321\_形成数据的过程如下：

1、Each\_chunk\_size是指每个通道所占用一个数据块中的多少

2、将一个通道二进制编码文件的大小除以each\_chunk\_size就可以得到数据块的个数

3、每个数据块按顺序包含rgb各个通道的二进制编码，所对应的一个像素的rgb值编码

4、最终形成的m是：| r0g0b0 | r1g1b1 | r2g2b2 | … | rngnbn |

m\_list = []

m\_list.append(open(file\_list[0], 'r').read())

m\_list.append(open(file\_list[1], 'r').read())

m\_list.append(open(file\_list[2], 'r').read()) # 最终m\_list是有3个元素

*for* i *in* range(int(ceil(len(m\_list[0]) / float(each\_chunk\_size)))):

start = i \* each\_chunk\_size

        end = min((i + 1) \* each\_chunk\_size, len(m\_list[0]))

        m += m\_list[0][start: end]

        m += m\_list[1][start: end]

        m += m\_list[2][start: end]

#No2建立应用层套接字

HOST = '127.0.0.1' # 应用层IP地址

PORT = 9999 *# 应用层端口号是这个9999*  # 最后才发现这里的端口号是随便写的

self.write\_fd = self.socket\_builder()# 建立一个套接字对象，这里使用的是子类的socket\_builder，port!=9999

def socket\_builder(*self*):

        socket\_send = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        socket\_send.bind((HOST, self.port)) # 这里端口是应用层套接字9999

        socket\_send.listen(1)

        print('waitting for connect........')

        write\_fd, addr = socket\_send.accept()

        print('connected')

*return* write\_fd

#No3 建立喷泉对象，主要用扩展窗喷泉码啦

self.fountain = self.fountain\_builder()

def fountain\_builder(*self*):

*if* self.fountain\_type == 'normal':

*return* Fountain(self.m, *chunk\_size*=self.fountain\_chunk\_size, *seed*=self.seed)

*elif* self.fountain\_type == 'ew':

*return* EW\_Fountain(self.m,

*chunk\_size*=self.fountain\_chunk\_size,

*w1\_size*=self.w1\_p,  *# 重要窗口大小*

*w1\_pro*=self.w1\_pro, *# 重要窗口概率*

*seed*=self.seed)

在顶层初始化时用得是’ew’，因此需要把这些参数传入，包括如下：

数据块字节大小、重要窗口大小、重要窗口概率、种子数

在建立扩展窗喷泉对象，主要建立窗口选择生成器（负责选择窗口）和建立度分布生成器（负责各个窗口生成度数），过程如下

第一步是，建立一个窗口选择生成器

self.windows\_id\_gen = self.windows\_selection() # 依概率选择两个窗口中的其中一个

第二步是，计算这两个窗口的大小

先根据数据量、编码块大小来计算得到编码块数量

self.data = data

self.chunk\_size = chunk\_size

self.num\_chunks = int(ceil(len(data) / float(chunk\_size)))  *#数据块数量*

然后计算依照概率各个窗口的大小（就是用数据块个数衡量窗口大小），重要窗口包含窗口概率\*数据块数量，次要窗口所有数据块

self.w1\_size = int(round(self.num\_chunks \* self.w1\_p))  *# 重要窗的规模大小，既可以容下多少个编码块*

self.w2\_size = self.num\_chunks                          *# 次要窗的规模大小*

第三步是，根据度分布函数建立两个窗口各自的度分布生成器

self.w1\_random\_chunk\_gen = robust\_soliton(self.w1\_size),*# 这个逗号是认真的*

self.w2\_random\_chunk\_gen = robust\_soliton(self.w2\_size) *# 返回的是度数，代表编码块个数*

#No4 打印相关信息

self.show\_info()

def show\_info(*self*):

        self.fountain.show\_info()

def show\_info(*self*):

        logging.info('Fountain info')

        logging.info('data len: {}'.format(len(self.data)))

        logging.info('chunk\_size: {}'.format(self.chunk\_size))

        logging.info('num\_chunks: {}'.format(self.num\_chunks))

#####到这一步为止就是Stack\_Sender()的初始化过程

#####下一步开始是sender\_main(self)方法的具体过程

#No1 建立并发送参数数据包，等待ACK

第一步是，建立参数数据包，并通过应用层套接字发送的

contant\_para\_packet = [

                PORT\_LIKE,

                SIGNIFICANT,

                RECV\_ID,

                PARA\_PACKET,

                self.fountain.chunk\_size]

*# 将contant\_para\_packet格式化成"integer", "B"'s Ctype is unsigned char*

        para\_packet = ''.join([struct.pack("B", ii) *for* ii *in* contant\_para\_packet]) + FILLING

        self.write\_fd.send(para\_packet)     *# 1、先发送参数数据包，重要*

第二步是，发送参数数据包，等待ACK的到来

*while* True:

read\_byte = self.write\_fd.recv(self.fountain\_chunk\_size + 3)

*if* not read\_byte:   *# 2、等待接收ACK*

*pass*

#No2 收到参数数据包的ACK后，发送水滴包

self.send\_drops\_use\_socket()    *# 3、然后发喷泉码数据包*

def send\_drops\_use\_socket(*self*):

        recv\_stop\_app = False

*while* not recv\_stop\_app:

            time.sleep(self.drop\_interval)          *#sleep for drop\_interval seconds*

            recv\_stop\_app = self.catch\_stop\_app()

            print('drop id : ', self.drop\_id)

            a\_drop = self.a\_drop() # 这里生成了一个水滴数据 #!!!!ATTENTION!!!!!!

            header = [PORT\_LIKE, NORNAL, RECV\_ID] # 加上包头

            send\_buff = ''.join([struct.pack("B", ii) *for* ii *in* header]) + a\_drop + FILLING

            print("the size of drop\_packet(send\_buff) is : {}".format(sys.getsizeof(send\_buff)))

*# print("Send Raw len {} : {}".format(len(send\_buff), [ord(ii) for ii in send\_buff]))*

            self.write\_fd.send(send\_buff)

第一步是，判断是否有收到recv\_stop\_app，否则进行下一步来发包；

第二步是，睡眠一段时间，等待给发送和接收过程一些处理时间

第三步是，非阻塞地接收一个包，返回的是布尔类型的stop\_app

第四步是，生成一个水滴数据，加上包头一起通过套接字发出去，直到收到stop\_app

#No3 在上述第四步中水滴数据是由水滴对象的方法生成

第一步是，Stack\_manager()的

a\_drop = self.a\_drop() # 这里生成了一个水滴数据 #!!!!ATTENTION!!!!!!

第二步是，self.a\_drop是Sender()的方法

def a\_drop(*self*):

*return* self.fountain.droplet().toBytes()

第三步是，a\_drop方法调用了喷泉对象（EW\_Fountain）的方法droplet()

第四步是，返回droplet().toBytes()方法，返回一个包含度数、种子数和编码数据的str

*return* bitarray.bitarray(num\_chunks\_bits + seed\_bits).tobytes() + self.data

#No4 droplet()方法如下：

第一步是，更新种子，self.seed

self.updateSeed()

def updateSeed(*self*):

        self.seed = random.randint(0,2\*\*31-1)

        random.seed(self.seed)

        np.random.seed(self.seed)

第二步是，根据度函数选取数据块

chunk\_list = self.EW\_RandChunkNums(self.num\_chunks)

第三步是，将所选数据块进行异或编码处理

data = None

*for* num *in* chunk\_list:

*if* data is None:

data = self.chunk(num)

*else*:

data = xor(data, self.chunk(num))

第四步是，最后根据异或完成的data、种子数和度数，返回一个EW水滴对象

*return* EW\_Droplet(data, self.seed, self.num\_chunks)

#######################

##Stack\_Receiver(EW\_Receiver)

#######################

#No1 自生的初始化，stack\_port，建立套接字

有单个数据块的大小、协议栈端口号、初始化父类

self.stack\_port = stack\_port

EW\_Receiver.\_\_init\_\_(self)

self.chunk\_size = 0

stack\_receiver -> EW\_Receiver -> Receiver

#No2 看Receiver的初始化

第一步是，初始各自参数，然后开始循环接收抓包

*while* True:

self.begin\_to\_catch()   *# 这里调用的是stack\_receiver()的begin\_to\_catch()*

*if* self.glass.isDone():

self.socket\_recv.close()

               print('recv done')

*break*

第二步是，看看顶层stack\_receiver()的方法begin\_to\_catch()

首先接收端口发来的信息，循环接收

read\_byte = self.socket\_recv.recv(PORT\_LIKE)

然后，收到重要参数数据包后回复ACK，并跳出循环

self.socket\_recv.send(ACK\_app)

*break*;

接着，开启第二个循环，来接收水滴数据包

a\_drop = self.catch\_a\_drop\_use\_socket() #不断抓包

*if* len(a\_drop) >= self.chunk\_size + data\_offset:

a\_drop = a\_drop[:self.chunk\_size+data\_offset] # 只要这个水滴的

self.add\_a\_drop(a\_drop)

最后，接收完成，发送Stop\_app

*if* self.glass.isDone():

print('recv done drop num {}'.format(self.drop\_id))

*break*

self.socket\_recv.send(buff\_stop\_app)

self.socket\_recv.close()