/\*\*

\* stack\_file.py

\* rs\_image\_lib.py

\*/

//#################

//###发送部分#######

//#################

# 应用层头部

---------------------------------------------------------------------------------------------------------------

| 端口 | 关键1/普通0 | 目的节点ID | 类型BMP/JPG/RS | 数据部分大小 | 总数据包个数 | 数据包编号 |     数据部分  |

+------+-------------+------------+----------------+--------------+--------------+------------+---------------+

| 1字节|    1字节    |    1字节   |      1字节     |     1字节    |     2字节    |    2字节   | chunk\_size字节|

---------------------------------------------------------------------------------------------------------------

'''

在 Python 中，不同的字符所占的字节数不同，数字、英文字母、小数点、下划线以及空格，各占一个字节，而一个汉字可能占 2~4 个字节，具体取决于采用的编码方式。例如，汉字在 GBK/GB2312 编码中占用 2 个字节，而在 UTF-8 编码中一般占用 3 个字节。

self.file\_contain = self.read\_file()

     self.file\_size = len(self.file\_contain)

read\_file()函数中包含两个逻辑，一个是RS编码后的字节，一个直接打开文件读入的字节。

RS编码后会形成一个新的文件路径，然后读入这个文件的字节。

With open as 是直接方便close 文件句柄

最后返回的是读入的字节

    def read\_file(*self*):

*if* self.RS :

            self.file\_path = rs\_encode\_image(self.file\_path, self.chunk\_size)

*with* open(self.file\_path, 'rb') *as* file\_fd: # ‘rb’是以二进制的形式读入整个文件

            file\_contain = file\_fd.read()

*return* file\_contain # 字节类型

主要发送函数方法

Send\_main()

        byte\_to\_send = len(self.file\_contain) # 纯图片部分的数据大小，int类型，就是字节的个数

        send\_tap = 0 # 发送数据的偏移量

        send\_serial = 0

        NUM\_PACKET = ceil(self.file\_size / self.chunk\_size) #file\_size同byte\_to\_send

self.set\_send\_type()

app\_header = [

                PORT\_LIKE,*#0*

                SIGNIFICANT,*#1*

                RECV\_ID,*#2*

                self.type\_,*#3*

                self.chunk\_size,*#4*

                ]

# “B”表示将每个元素用unsigned char编码，各个是1个字节

        app\_header = ''.join([struct.pack("B", ii) *for* ii in app\_header])

        app\_header += struct.pack("H", int(NUM\_PACKET)) # “H”表示用unsigned short编码，2个字节

接下来就是发送了：

*while* byte\_to\_send > 0: # int类型，数据包

            time.sleep(self.packet\_interval) # 休眠一段时间让接收方能收到

            packet\_to\_send = app\_header + \

                    struct.pack("H", send\_serial) + \

self.file\_contain[send\_tap : min((send\_tap + self.chunk\_size), self.file\_size)]

# chunk\_size就是数据部分，大小不包含头部，纯数据大小

            self.stack\_fd.send(packet\_to\_send)

            logging.info("send {}\tall  {}".format(send\_serial, NUM\_PACKET))

            send\_serial += 1

            send\_tap += self.chunk\_size # 偏移量逐段叠加

            byte\_to\_send -= self.chunk\_size # 剩下的数据int数量

理一下：

读入图片数据：file\_contain，字节类型数据

以下都是int类型，代表的是file\_contain的字节数目

Byte\_to\_send = len(file\_contain)

Self.file\_size = len(file\_contain)

Chunk\_size是每次要发送的纯数据字节大小，不含应用层包头

Ceil是向上取整，2 == ceil( 10 / 8)

Num\_packet = ceil(self.file\_size / chunk\_size)

Send\_tap是偏移量，从0开始，每次加chunk\_size大小

self.file\_contain[send\_tap : min((send\_tap + self.chunk\_size), self.file\_size)]表示每次发送的那一段数据，共chunk\_size个字节大小

//#################

//###接收部分#######

//#################

def \_\_init\_\_(*self*,

*stack\_port* = 9079 + RECV\_ID,

            ):

        self.stack\_port = stack\_port

        self.stack\_fd = self.socket\_builder()

        self.testFile = time.asctime().replace(' ', '\_').replace(':', '\_') # 文件夹名称

        self.test\_dir = os.path.join(

                SIM\_PATH,

                "file\_recv\_dir",

                self.testFile

                )

*if* not os.path.exists(self.test\_dir):

            os.mkdir(self.test\_dir) # 创建文件夹

self.img\_fd , self.img\_name = self.img\_fd\_builder()

def img\_fd\_builder(*self*):

        img\_name = os.path.join(

                self.test\_dir,

                self.testFile

                )

        img\_fd = open(img\_name, "wb") # 创建文件对象，以二进制形式写入

*return* img\_fd, img\_name # 返回文件对象，文件名

self.drop\_interval = 60 # socket的连接接收时间限制

self.recv\_by\_now = 0

self.recv\_packet = []

ready = select.select([self.stack\_fd], [], [], self.drop\_interval) # ready[0]可以作为判断条件

参数原型:  
int select(int maxfdpl, fd\_set \* readset, fd\_set \*writeset, fd\_set \*exceptset, const struct timeval \* tiomeout)

第一个是最大的文件描述符长度  
第二个是监听的可读集合  
第三个是监听的可写集合  
第四个是监听的异常集合  
第五个是时间限制

python 就简单了, 直接调用封装好的select , 其底层处理好了文件描述符的相关读写监听, 我们在Python 中只需这么写:  
can\_read, can\_write, \_ = select.select(inputs, outputs, None, None)

第一个参数是我们需要监听可读的套接字, 第二个参数是我们需要监听可写的套接字, 第三个参数使我们需要监听异常的套接字, 第四个则是时间限制设置.

如果监听的套接字满足了可读可写条件, 那么所返回的can,read 或是 can\_write就会有值了, 然后我们就可以利用这些返回值进行随后的操作了。相比较unix 的select模型, 其select函数的返回值是一个整型, 用以判断是否执行成功.

第一个参数就是服务器端的socket, 第二个是我们在运行过程中存储的客户端的socket, 第三个存储错误信息。  
重点是在返回值, 第一个返回的是可读的tuple, 第二个存储的是可写的tuple, 第三个存储的是错误信息的tuple。

self.img\_fd.seek(0, 0)

self.img\_fd.seek(packet\_offset, 1)

语法

fileObject.seek(offset[, whence])

参数

offset----偏移量（相对于文件某个位置偏移的字节数）

whence----offset的辅助（可选，默认值为 0）

0：从文件头开始

1：从文件当前位置开始

2：从文件尾开始

返回值

成功：返回新的文件位置

失败：返回 -1

*if* ord(packet\_byte[3]) == BMP\_TYPE:

os.rename(self.img\_name, self.img\_name + ".bmp")

logging.info("recv done , img name: {}".format(self.img\_name + ".bmp"))

*elif* ord(packet\_byte[3]) == JPG\_TYPE:

os.rename(self.img\_name, self.img\_name + ".jpg")

logging.info("recv done , img name: {}".format(self.img\_name + ".jpg"))

普通发送jpg和bmp的解码时直接在接收文件的末尾添加文件后缀名字就可以恢复图像