**OAR游戏联机代理平台v1.2.3**

**软**

**件**

**源**

**代**

**码**

**节**

**选**

**韦若枫**

*# Aisle的核心,需要同级目录下的bin文件夹以正常启动对应模块*

*import* shutil

*import* socket

*from* time *import* sleep

*from* config *import* \*

*import* NATTypeDetector

*import* platform

*import* os

*from* subprocess *import* Popen, PIPE, STDOUT

*import* chardet

*import* shortuuid

*from* socket *import* gethostname

*from* base64 *import* b64encode, b64decode

*import* \_thread as thread

def decodeB64String(raw: str):

    """

    :param raw: 源被编码的字符串

    :return:  解码后的字符串

    """

*return* str(b64decode(raw), 'utf-8')

def encodeB64String(raw: str):

    """

    :param raw: 原正常字符串

    :return:  编码后的字符串

    """

*return* str(b64encode(raw.encode('utf-8')), 'utf-8')

*# 使用logging记录subprocess的输出，来自 https://stackoverflow.com/questions/21953835/run-subprocess-and-print-output-to-logging*

def logSubprocessOutput(pipe, logger, \_codec):

    LOG.debug(f'PIPE开头：{pipe.readline()}')

*for* line in iter(pipe.readline, b''):

        line = line.decode(\_codec).replace('\n', '')  *# 删去行末的/n，logging自动会换行*

        logger.info(line)

class Aisle(AisleDefault, object):  *# 核心控制类，对应vps/用户电脑*

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.logger.debug('初始化Aisle...')

*# 抛弃*

*# # 详见 https://docs.python.org/3.7/library/multiprocessing.html?highlight=process#multiprocessing.freeze\_support*

*# multiprocessing.freeze\_support()*

        self.NATType = ''

        self.localIP = socket.gethostbyname(socket.gethostname())

        self.clientModuleInstance = {}

        self.CMIHandler = {}  *# 存储各个模块的线程实例的字典*

        self.logger.debug('初始化完成')

    def \_\_del\_\_(self):

*for* \_ in self.clientModuleInstance.values():

            self.logger.debug(f'删除对象{\_.\_\_class\_\_.\_\_name\_\_}, {\_}')

            del \_

    def getNATType(self, ifCute=False):

        self.NATType = NATTypeDetector.test()[0]  *# 进行一次NAT测试*

*if* ifCute:  *# ifCute为真时，使用SSR等抽卡名称*

*if* self.NATType in NAT\_TYPE\_MAP.keys():

                natQuality = NAT\_TYPE\_MAP[self.NATType]

*else*:

                natQuality = f"???\_{self.NATType}"

*return* natQuality

*else*:

*return* self.NATType

    def \_\_phaseAisleCode(self, \_code: str):

        """

        将AisleCode联机码转换为各个参数

        :param \_code: AisleCode

        :return: 参数均为字符串,所有B64编码在此已解码

        """

        mode, \_rest = \_code.split('://', maxsplit=1)

        serverInfo, \_rest = \_rest.split('/', 1)

*# 不再使用联机码存储token*

*# token, \_rest = \_rest.split('/', 1)*

*if* '/' in \_rest:

            payload = \_rest.split('/')[0]

            self.logger.error(f'超出预期的联机码，联机码可能有错误 {\_code} ，超出预期的部分：/{\_rest}')

*else*:

            payload = \_rest

*# 处理serverInfo部分*

        serverInfoStr = decodeB64String(serverInfo)

        serverIP, port = serverInfoStr.split(':')

*# 联机码不再包含token部分*

*# token = decodeB64String(token)*

*# 处理payload部分*

        payload = decodeB64String(payload)

*return* mode, serverIP, port, payload

    def joinAisleCode(self, \_code, localPort, \_token, tls):

        """

        使用AisleCode加入一个主机

        :param \_code: 联机码，形如： ProtocolName://(B64\_ServerInfo)/(B64\_ServerToken)/(B64\_Payload)

        :param localPort:  指定Aisle所绑定的远程服务到本地的端口

        :param \_token: 传入服务器的鉴权码

        :param tls: None | tls加密文件目录

        :return: 直接尝试加入服务器，无返回值

        """

*try*:

            \_mode, \_serverIP, \_port, \_payload = self.\_\_phaseAisleCode(\_code)

*except* ValueError:

            self.logger.critical(f'无法识别的联机码：{\_code}')  *# 错误处理*

*return*

        self.logger.debug(f'信息：{\_mode}, {\_serverIP}, {\_port}, {\_payload}')

*if* \_mode == 'XTCP':

            self.clientModuleInstance[\_mode] = XTCP(serverIP=\_serverIP, serverPort=\_port, token=\_token, tls=tls)

*# 抛弃*

*# self.CMIHandler[\_mode] = multiprocessing.Process(*

*#     target=self.clientModuleInstance['XTCP'].startVisitor,*

*#     args=(\_payload, localPort, self.localIP)*

*# )*

*# self.CMIHandler[\_mode].start()*

            self.CMIHandler[\_mode] = thread.start\_new\_thread(

                self.clientModuleInstance[\_mode].startVisitor,

                (\_payload, localPort, self.localIP)

            )

*elif* \_mode == 'STCP':

            self.clientModuleInstance[\_mode] = STCP(serverIP=\_serverIP, serverPort=\_port, token=\_token, tls=tls)

*# 抛弃*

*# self.CMIHandler[\_mode] = multiprocessing.Process(*

*#     target=self.clientModuleInstance[\_mode].startVisitor,*

*#     args=(\_payload, localPort, self.localIP)*

*# )*

*# self.CMIHandler[\_mode].start()*

            self.CMIHandler[\_mode] = thread.start\_new\_thread(

                self.clientModuleInstance[\_mode].startVisitor,

                (\_payload, localPort, self.localIP)

            )

*else*:

            self.logger.error(f'未兼容的协议{\_mode}，Aisle版本{VERSION}')

    def startXTCPHost(self, serverIP, serverPort, token, sk, localPort, tls):

        \_mode = 'XTCP'

        self.clientModuleInstance[\_mode] = XTCP(serverIP=serverIP, serverPort=serverPort, token=token, tls=tls)

        self.CMIHandler[\_mode] = thread.start\_new\_thread(

            self.clientModuleInstance[\_mode].startHost,

            (sk, localPort)

        )

*#*

*# self.CMIHandler[\_mode] = multiprocessing.Process(*

*#     target=self.clientModuleInstance[\_mode].startHost,*

*#     args=(sk, localPort)*

*# )*

*#*

        self.logger.debug('XTCP进程开始')

        sleep(5)

*return* self.clientModuleInstance[\_mode].generateAisleCode()  *# 返回联机码的Payload*

    def startSTCPHost(self, serverIP, serverPort, token, sk, localPort, tls):

        \_mode = 'STCP'

        self.clientModuleInstance[\_mode] = STCP(serverIP=serverIP, serverPort=serverPort, token=token, tls=tls)

        self.CMIHandler[\_mode] = thread.start\_new\_thread(

            self.clientModuleInstance[\_mode].startHost,

            (sk, localPort)

        )

*# self.CMIHandler[\_mode] = multiprocessing.Process(*

*#     target=self.clientModuleInstance[\_mode].startHost,*

*#     args=(sk, localPort)*

*# )  # 用multiprocess，析构函数正常触发*

*#*

        self.logger.debug('STCP进程开始')

        sleep(5)  *# 等待文件生成*

*return* self.clientModuleInstance[\_mode].generateAisleCode()  *# 返回联机码的Payload*

class AisleClientModuleMixin(AisleDefault):

    def \_\_init\_\_(self, serverIP, serverPort, token):

        AisleDefault.\_\_init\_\_(self)

        self.stopFlag = False  *# 中止外部进程标志*

        self.mode = ''

        self.serverIP = serverIP

        self.serverPort = serverPort

        self.token = token

        self.payload = ''

        self.AisleCodePath = 'share.aislecode'

        self.logger.debug(f'将code写入：{self.AisleCodePath}')

    def makeAisleCode(self):

*with* open(self.AisleCodePath, mode='w', encoding='utf-8') as f:

            self.logger.debug(self.\_generateAisleCode())

            f.write(self.\_generateAisleCode())

    def generateAisleCode(self):

*with* open(self.AisleCodePath, mode='r', encoding='utf-8') as f:

            \_ = f.readline()

*return* \_

    def \_generateAisleCode(self):

        """

        所有的Client模块都要有生成AisleCode的功能，都要有self.payload

        :return: AisleCode联机码

        """

        code = f'{self.mode}://'

        ServerInfo = encodeB64String(f'{self.serverIP}:{self.serverPort}')

        code += ServerInfo

        '''不再在联机码中隐式存储token

        code += '/'

        Token = encodeB64String(self.token)

        code += Token

        '''

        code += '/'

*if* self.payload == '':

            self.logger.warning('模块提供了一个空的payload')

        Payload = encodeB64String(self.payload)

        self.logger.debug(f'生成payload{Payload}')

        code += Payload

        self.logger.debug(f'返回code{code}')

*return* code

class FrpCtl(AisleDefault):  *# 用来创建、控制单个frp进程的类，*

    def \_\_init\_\_(self):

        AisleDefault.\_\_init\_\_(self)

        self.logger.debug('初始化frp模块...')

*# 确定frp相对路径，仅支持windows和linux识别*

        self.hostname = gethostname()

*# 处理操作系统名称*

        self.system = platform.system().lower()

*# 处理架构*

        \_arch = platform.architecture()[0][0:2]

*if* \_arch == '64':

*pass*

*elif* \_arch == '32':

            \_arch = ''

        self.binDir = f'./bin/' \

                      f'frp\_' \

                      f'{FRP\_VERSION}\_' \

                      f'{self.system}\_' \

                      f'amd{\_arch}'  *# 直接写入amd作为临时解决方案*

*if* os.path.exists(self.binDir):

*pass*

*else*:

            self.logger.warning(f'没有找到对应的frp文件，frp将无法启动，请联系开发者以获取帮助。检测到的操作系统:{self.system} {\_arch}')

*# 初始化一个字符串作为工作模式*

        self.mode = ''

*# 抛弃！初始化一个字典作为额外的启动参数*

*# self.startArgs = {}*

*# 配置文件路径*

        self.configFilePath = ''

*# 使用config字典存储配置*

        self.config = {

            'common': {},  *# 存储common*

            'proxy': {}  *# 存储多个proxy*

        }

*# 用以存储Popen的实例*

        self.handler = None

    def \_\_del\_\_(self):

*if* os.path.exists(self.configFilePath):

            self.logger.debug(f'存在临时配置文件，删除')

*if* LOG\_LEVEL == 'DEBUG':

*try*:

*with* open(self.configFilePath, mode='r', encoding='utf-8') as f:

                        self.logger.debug('-------临时配置文件内容开始-------')

*for* line in f.readlines():

                            line = line.split('\n')[0]

                            self.logger.debug(line)

                        self.logger.debug('-------临时配置文件内容结束-------')

*except* NameError:

                    self.logger.info('GC已回收\_\_buildins\_\_方法，临时文件无法读取；请不要在主线程中直接实例化Aisle')

*if* not NO\_DEL\_TEMP:

                    self.logger.warning(f'删除临时配置文件')

                    os.remove(self.configFilePath)

                    shutil.rmtree(TEMP\_DIR)

    @staticmethod

    def \_phaseDirPath(path):

        """

        将文件目录处理为以/结尾

        :param path: 未处理的目录

        :return: 以/结尾的目录

        """

*if* path[-1:] == '/':

*return* path

*else*:

*return* path + '/'

    @staticmethod

    def \_item2Config(item):

        """

        将字典的item变为frp.ini中的一行字符串

        :param item: 字典的一个item

        :return: 一行字符串

        """

        name, val = item

*return* f'{name} = {val}'

    def writeConf(self):

*if* not os.path.exists(TEMP\_DIR):

            os.mkdir(TEMP\_DIR)

        self.configFilePath = f'{TEMP\_DIR}/frpc.ini'

        self.logger.debug(f'配置文件路径：{self.configFilePath}')

*with* open(self.configFilePath, mode='w', encoding='utf-8') as f:

*# 写入common部分*

            f.write('[common]\n')

*for* item in self.config['common'].items():  *# 遍历字典写入所有参数*

                f.write(self.\_item2Config(item) + '\n')

*# 写入各个proxy部分*

*for* proxyName, proxyConfig in self.config['proxy'].items():

                f.write(f'[{proxyName}]\n')

*for* item in proxyConfig.items():

                    f.write(self.\_item2Config(item) + '\n')

class FrpClient(AisleClientModuleMixin, FrpCtl):  *# 所有Client和一个Server通信*

    def \_\_init\_\_(self, serverIP, serverPort, token, tls=False):

        """

        初始化一个客户端Frpc实例

        :param serverIP: 服务器IP

        :param serverPort: 服务器Port

        :param token: 服务器token,弱国不指定则使用config中的值

        :param tls: 是否启用tls；使用config.py提供的路径

        """

*# 多继承要一个个轮流初始化，super()只会横向搜索同深度的第一个构造函数*

        FrpCtl.\_\_init\_\_(self)

        AisleClientModuleMixin.\_\_init\_\_(self, serverIP=serverIP, serverPort=serverPort, token=token)  *# 这里尤其注意加self*

        self.binPath = self.binDir + '/frpc'

*if* self.system == 'windows':

            self.logger.debug('对windows系统下的执行路径进行替换')

            self.binPath = self.binPath.replace('/', '\\')

            self.binPath += '.exe'

        self.logger.debug(f'使用二进制文件{self.binPath}')

*if* os.path.exists(self.binPath):

*pass*

*else*:

            self.logger.error(f'没有找到对应的frpc软件，frp将无法启动，请联系开发者以获取帮助。'

                              f'使用二进制文件{self.binPath}')

*# 使用config字典存储*

        \_common = {

            'server\_addr': serverIP,

            'server\_port': serverPort,

            'token': token

        }

        self.config['common'].update(\_common)

*# 处理tls*

*if* tls:

            self.logger.info(f'---启用客户端传输层安全---')

            self.tlsDir = TLS\_DIR  *# 将tlsDir处理为以/结尾的字符串*

            self.tlsDir = self.\_phaseDirPath(self.tlsDir)

            \_tlsCrt = self.tlsDir + 'client.crt'

            \_tlsKey = self.tlsDir + 'client.key'

*if* os.path.exists(\_tlsCrt) and os.path.exists(\_tlsKey):

                self.config['common'].update(

                    {

                        'tls\_enable': 'true',

                        'tls\_cert\_file': \_tlsCrt,

                        'tls\_key\_file': \_tlsKey

                    }

                )

*else*:

                self.logger.error(f'tls目录配置出错，请检查 {\_tlsCrt} & {\_tlsKey} 是否存在')

                self.logger.info('---客户端tls启动失败，欲连接OAR服务器则必须启用客户端tls---')

*else*:

            self.logger.info('---未启用客户端tls，欲连接OAR服务器则必须启用客户端tls---')

        self.logger.debug('初始化frp完成')

    def \_\_del\_\_(self):

        self.logger.debug('FrpCtl析构函数开始运行')

        FrpCtl.\_\_del\_\_(self)  *# 继承FrpCtl的析构函数*

*if* self.handler:

            self.handler.terminate()

    def startSubprocess(self):

        self.logger.debug('FrpClient启动子进程')

*if* self.handler is None:

*pass*

*else*:

            self.logger.warning('同一个实例存在已经实例化的外部进程，将要关闭。请不要用同一个frp实例创建多个外部进程！')

            self.handler.terminate()

        self.logger.info('启动frpc外部进程')

        self.writeConf()  *# 将配置写入文件*

        \_args = [self.binPath, '-c', self.configFilePath]

        self.logger.debug(f'\_args: {\_args}')

*# 显示魔法代码*

*if* LOG\_LEVEL == 'DEBUG':

            \_magicString = ''

*for* i in \_args:

                \_magicString += f'{i} '

            self.logger.debug(f'魔法代码：{\_magicString}')

        self.logger.info('frp外部进程开始')

*# TODO 似乎在Linux不工作*

        self.handler = Popen(

            args=\_args,

            stdout=PIPE,

            shell=True,

            stderr=STDOUT

        )

*# 临时解决方案：立即删除配置文件,\_\_del\_\_方法在使用\_thread或multiprocessing时，貌似不起作用*

*# 关于作者（RuofengX）对frpc的隐私强化建议详见：https://github.com/fatedier/frp/issues/2582*

        sleep(1)  *# 等待frpc1秒钟*

        self.logger.debug(f'删除临时配置文件')

        os.remove(self.configFilePath)

        shutil.rmtree(TEMP\_DIR)

*with* self.handler.stdout as \_pipe:

            \_codec = chardet.detect(\_pipe.readline(24))['encoding']  *# 获取编码方式*

            self.logger.debug(f'检测的代码为{\_codec}')

            logSubprocessOutput(\_pipe, self.logger, \_codec=\_codec)

*for* line in iter(\_pipe.readline, b''):

                line = line.decode(\_codec).replace('\n', '')  *# 删去行末的/n，logging自动会换行*

                self.logger.info(line)

        self.logger.critical('frp外部进程结束')

class XTCP(FrpClient):

    def \_\_init\_\_(self, serverIP, serverPort, token, tls):

        super(XTCP, self).\_\_init\_\_(serverIP=serverIP, serverPort=serverPort, token=token, tls=tls)

        self.uid = ''

        self.mode = 'XTCP'

    @classmethod

    def generatePayload(cls, uid: str, sk: str):

        \_payload = uid + sk

*return* \_payload

    @classmethod

    def phasePayload(cls, payload):

        uid = payload[:UID\_LENGTH]

        sk = payload[UID\_LENGTH:]

*return* uid, sk

    def startHost(self, sk, localPort, localIP='127.0.0.1'):

        """

        :param self:

        :param sk: 可选自定义密码

        :param localPort: 绑定的本地链接

        :param localIP: 本机IP， 后备选项为127.0.0.1

        :return: 无返回值即代表正常运行，返回0代表外部进程结束，但是要保证实例在启动之后，self.payload为有效的分享码的payload

        """

        self.logger.info(f'将使用XTCP作为主机')

        self.uid = shortuuid.ShortUUID(alphabet="0123456789ABCDEF").random(UID\_LENGTH)

        self.config['proxy'][self.uid] = {

            'type': self.mode.lower(),  *# 配置文件中小写mode*

            'role': 'server',

            'local\_ip': localIP,

            'local\_port': localPort

        }

*# 兼容空字符串的sk*

*if* sk == '':

*pass*

*else*:

            self.config['proxy'][self.uid]['sk'] = sk

*# self.logger.debug(f'启动参数 {self.startArgs}')*

        self.logger.debug(f'启动参数 {self.config}')

*# 启动前生成XTCPCode*

        self.logger.debug('生成了payload：' + self.payload)

        self.payload = self.generatePayload(uid=self.uid, sk=sk)

        self.logger.debug(self.payload)

        self.makeAisleCode()  *# 将分享码写入文件*

*# 启动外部进程*

        self.startSubprocess()

    def startVisitor(self, payload, \_localPort, \_localIP='127.0.0.1'):

        self.logger.info(f'使用XTCP作为访客')

        \_uid, \_sk = self.phasePayload(payload=payload)

        self.config['proxy'][f'{\_uid}-visitor'] = {

            'type': self.mode.lower(),

            'role': 'visitor',

            'server\_name': \_uid,

            'bind\_address': \_localIP,

            'bind\_port': \_localPort

        }

*if* \_sk == '':

*pass*

*else*:

            self.config['proxy'][f'{\_uid}-visitor']['sk'] = \_sk

        self.startSubprocess()

class STCP(FrpClient):

    def \_\_init\_\_(self, serverIP, serverPort, token, tls):

        super(STCP, self).\_\_init\_\_(serverIP=serverIP, serverPort=serverPort, token=token, tls=tls)

        self.uid = ''

        self.mode = 'STCP'

    @classmethod

    def generatePayload(cls, uid: str, sk: str):

        payload = uid + sk

*return* payload

    @classmethod

    def phasePayload(cls, payload):

        uid = payload[:UID\_LENGTH]

        sk = payload[UID\_LENGTH:]

*return* uid, sk

    def startHost(self, sk, localPort, localIP='127.0.0.1'):

        """

        :param self:

        :param sk: 可选自定义密码

        :param localPort: 绑定的本地链接

        :param localIP: 本机IP， 后备选项为127.0.0.1

        :return: 无返回值即代表正常运行，返回0代表外部进程结束，但是要保证实例在启动之后，self.payload为有效的分享码的payload

        """

        self.logger.info(f'将使用STCP作为主机')

        self.uid = shortuuid.ShortUUID(alphabet="0123456789ABCDEF").random(UID\_LENGTH)

        self.config['proxy'][self.uid] = {

            'type': self.mode.lower(),  *# 配置文件中小写mode*

            'role': 'server',

            'local\_ip': localIP,

            'local\_port': localPort

        }

*# 兼容空字符串的sk*

*if* sk == '':

*pass*

*else*:

            self.config['proxy'][self.uid]['sk'] = sk

        self.logger.debug(f'启动参数 {self.config}')

*# 启动前生成STCPCode*

        self.logger.debug('生成了payload：' + self.payload)

        self.payload = self.generatePayload(uid=self.uid, sk=sk)

        self.logger.debug(self.payload)

        self.makeAisleCode()  *# 将分享码写入文件*

*# 启动外部进程*

        self.startSubprocess()

    def startVisitor(self, payload, \_localPort, \_localIP='127.0.0.1'):

        self.logger.info(f'使用STCP作为访客')

        \_uid, \_sk = self.phasePayload(payload=payload)

        self.config['proxy'][f'{\_uid}-visitor'] = {

            'type': self.mode.lower(),

            'role': 'visitor',

            'server\_name': \_uid,

            'bind\_address': \_localIP,

            'bind\_port': \_localPort

        }

*if* \_sk == '':

*pass*

*else*:

            self.config['proxy'][f'{\_uid}-visitor']['sk'] = \_sk

        self.startSubprocess()

*# Aisle的命令行程序，默认和OAR的服务器通信*

*import* Aisle

*import* click

*from* config *import* \*

class AisleHandler(AisleDefault):  *# 一个core的操作句柄*

    def \_\_init\_\_(self):

        super(AisleHandler, self).\_\_init\_\_()

        self.core = None

        self.code = ''

    def create(self):

*if* self.core is None:

            self.core = Aisle.Aisle()

*else*:

            self.logger.warning(f'存在Aisle.Aisle实例，将不再创建:\t {self.core}')

*return* self.core

    def hold(self):

*if* self.core is None:

            LOG.warning(f'core实例为None')

*return*

        LOG.debug(f'挂起CL进程')

        self.\_\_loop()

    def \_\_loop(self):

*try*:

            LOG.debug('CL主进程挂起')

*while* 1:

*pass*

*except* Exception as \_:

            LOG.critical(f'[致命错误]{\_}结束挂起')

            self.\_\_core\_terminal()

    def \_\_core\_terminal(self):

*if* self.core is None:

            self.logger.debug(f'销毁实例\t {self.core}')

            del self.core

*else*:

            self.logger.warning(f'不存在Aisle.Aisle实例，不操作{self.core}')

    def startXTCPHost(self, server\_ip, server\_port, token, secret\_key, local\_port='25565', tls=True):

        self.create()

        self.code = self.core.startXTCPHost(serverIP=server\_ip, serverPort=server\_port, token=token, sk=secret\_key,

                                            localPort=local\_port, tls=tls)

    def startSTCPHost(self, server\_ip, server\_port, token, secret\_key, local\_port='25565', tls=True):

        self.create()

        self.code = self.core.startSTCPHost(serverIP=server\_ip, serverPort=server\_port, token=token, sk=secret\_key,

                                            localPort=local\_port, tls=tls)

    def join(self, \_code, token, local\_port='25565', tls=True):

        self.create()

        self.code = self.core.joinAisleCode(\_code=\_code, \_token=token, localPort=local\_port, tls=tls)

    @property

    def NAT(self):

        self.create()

        self.core.getNATType(ifCute=True)

*return* self.core.getNATType(ifCute=True)

@click.group()

def AisleCL():

*pass*

@AisleCL.command(help='查看软件信息')

def info():

    click.echo(f'这里是 OAR Aisle CL {VERSION}')

    click.echo(f'{INFO}')

@AisleCL.command(help='我的回合，抽卡！ --对家宽质量做出评价，结果决定了能否能够联机')

def test():

    hd = AisleHandler()

    click.echo(f'{NAT\_TEST\_START}')

    click.echo(f'{NAT\_HELP}')

    click.echo(f'{NAT\_TYPE\_IS} ' + hd.NAT)

@AisleCL.command(help='使用XTCP作为主机')

@click.option('--server\_ip', default=SERVER\_IP)

@click.option('--server\_port', default=SERVER\_PORT)

@click.option('--token', default=SERVER\_TOKEN)

@click.option('--local\_port', default='25565')

@click.option('--secret\_key', default='')

def start\_xtcp(server\_ip, server\_port, token, local\_port, secret\_key):

    hd = AisleHandler()

    hd.startXTCPHost(server\_ip=server\_ip, server\_port=server\_port, token=token, secret\_key=secret\_key,

                     local\_port=local\_port)

    click.echo(SPLIT\_LINE)

    click.echo(f'本次联机码为： {hd.code}')

    click.echo('联机码是访问您电脑的唯一凭证，请妥善保管')

    click.echo(SPLIT\_LINE)

    LOG.debug(f'开始阻塞')

    hd.hold()

@AisleCL.command(help='使用STCP作为主机')

@click.option('--server\_ip', default=SERVER\_DOMAIN)

@click.option('--server\_port', default=SERVER\_PORT)

@click.option('--token', default=SERVER\_TOKEN)

@click.option('--local\_port', default='25565')

@click.option('--secret\_key', default='')

def start\_stcp(server\_ip, server\_port, token, local\_port, secret\_key):

    hd = AisleHandler()

    hd.startSTCPHost(server\_ip=server\_ip, server\_port=server\_port, token=token, secret\_key=secret\_key,

                     local\_port=local\_port)

    click.echo(SPLIT\_LINE)

    click.echo(f'本次联机码为： {hd.code}')

    click.echo('联机码是访问您电脑的唯一凭证，请妥善保管')

    click.echo(SPLIT\_LINE)

    LOG.debug(f'开始阻塞')

    hd.hold()

*# TODO 添加鉴权机制，适当对流量转发收费*

@AisleCL.command(help='使用分享码加入别的主机')

@click.option('--localPort', default='25565')

@click.option('--token', default=SERVER\_TOKEN)

@click.option('--aislecode', default='')

def join(localport, aislecode, token):

*if* aislecode == '':

        aislecode = input('请在这里输入或粘贴联机码，回车键结束\n')

        aislecode.strip()  *# 必须删除两端空格*

    hd = AisleHandler()

*try*:

        hd.join(\_code=aislecode, token=token, local\_port=localport)

        LOG.debug(f'开始阻塞')

        hd.hold()

*except* Exception as exp:

        LOG.critical(f'发生{exp}错误')

    click.echo('进程结束')

*if* \_\_name\_\_ == '\_\_main\_\_':

    LOG.critical(SPLIT\_LINE)

    LOG.critical(f'您正在使用一个未经充分测试的预发布版本')

    LOG.critical(f'发现任何Bug请与作者weiruofeng@outlook.com联系')

    LOG.critical(SPLIT\_LINE)

    AisleCL()

*# config.py将被编译到二进制代码中，但会被版本控制同步，请不要存放敏感数据*

*import* os.path

*import* colorlog

*import* dns.resolver

*import* random

*# -------DEBUG区域-------*

LOG\_LEVEL = 'INFO'  *# in ['CRITICAL', 'ERROR', 'WARNING', 'INFO', 'DEBUG', 'NOTSET']*

NO\_DEL\_TEMP = False  *# 启用后：临时文件夹的文件将不会删除，以供程序运行结束后查看*

*# -------抽象元类-------*

class AisleDefault(object):

    def \_\_init\_\_(self):

        self.logger = colorlog.getLogger(self.\_\_class\_\_.\_\_name\_\_)

        self.logger.setLevel(LOG\_LEVEL)

*if* not self.logger.handlers:

            self.logger.addHandler(CONSOLE\_HANDLER)

        self.logger.debug('日志功能启动')

*# -------模块设置-------*

*# ---logging相关配置---*

*# 配置着色的Handler*

CONSOLE\_HANDLER = colorlog.StreamHandler()

CONSOLE\_HANDLER.setFormatter(

    colorlog.ColoredFormatter('%(log\_color)s%(asctime)s [%(levelname)s] %(name)s(%(funcName)s):\t%(message)s'))

*# 配置全局logger*

LOG = colorlog.getLogger('GLOBAL')  *# 添加全局LOG，避免使用logging.debug()时创建新的handler混淆日志*

LOG.setLevel(LOG\_LEVEL)

LOG.addHandler(CONSOLE\_HANDLER)

*# -------AisleCL配置-------*

*# ---私有模块导入---*

*if* os.path.exists('./OAR'):

*import* OAR.config  *# 模块化的私有参数才能让pyinstaller识别*

*# -OAR服务器的私有配置-*

    SERVER\_TOKEN = OAR.config.SERVER\_TOKEN

    SERVER\_DOMAIN = OAR.config.SERVER\_DOMAIN

    SERVER\_PORT = OAR.config.SERVER\_PORT

*else*:

    LOG.warning(f'缺失OAR配置模块！请自行搭建frps公网服务器并建立自己的私有模块')

*# -自建服务器的私有配置-*

    SERVER\_TOKEN = ''  *# frps的鉴权token*

    SERVER\_DOMAIN = ''  *# frps所在的公网服务器的域名 | 如果没有域名请自行修改DNS获取IP地址的逻辑*

    SERVER\_PORT = ''  *# frps监听的端口*

*if* SERVER\_DOMAIN == '' \

        or SERVER\_TOKEN == '' \

        or SERVER\_PORT == '':

    LOG.critical(f'请在config.py中填入自建公网服务器的参数')

*raise* SystemExit

*# ---获取域名对应IP---*

*try*:

    \_rrset = dns.resolver.resolve(SERVER\_DOMAIN, rdtype='A', raise\_on\_no\_answer=False).rrset

*except* dns.exception.Timeout:

    LOG.critical(f'无法解析域名，请检查网络连接和DNS服务器配置！')

*raise* SystemExit

SERVER\_IP = str(\_rrset).split(' ')[4]

*# ---多语言消息实现 | i18n Message---*

SPLIT\_LINE = '-------------------------------------------'  *# split line*

INFO = 'OAR Aisle是一个致力于打造沉浸式多人游戏联机体验的软件。'

NAT\_TEST\_START = '开始NAT-Type测试，请稍等'

NAT\_TYPE\_IS = '当前网络环境下的NAT-Type为：'

NAT\_TYPE\_MAP = {"Blocked": "-", "Open Internet": 'SP', "Full Cone": "SSR", "Restrict NAT": "SR",

                "Restrict Port NAT": "R", "Symmetric NAT": "N"}

NAT\_HELP = '家庭宽带如抽卡。家宽品质越高，网络连接越容易。一般来说，出现SSR品质则很容易建立连接；而如果出现N品质，则几乎不可能建立连接。'

*# -------Aisle配置-------*

VERSION = 'PRE V1.2.3'

LOG.info(f'版本号{VERSION}')

FRP\_VERSION = '0.37.1'  *# 兼容的FRP版本*

TEMP\_DIR\_ROOT = './temp'

TEMP\_DIR = TEMP\_DIR\_ROOT + '/' + ''.join(

    random.sample('1234567890poiuytrewqasdfghjklmnbvcxz', 16))  *# # 临时文件夹路径，附加随机一个子文件夹*

TLS\_DIR = './ssl/'  *# ssl证书文件夹路径*

UID\_LENGTH = 5  *# uid长度*

DEFAULT\_CODEC = 'gbk'

*import* PyInstaller.\_\_main\_\_

*import* random

*import* os

class Builder(object):

    def \_\_init\_\_(self, name, icon):

        self.name = name

        self.key = ''.join(random.sample('1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPLKJHGFDSAZXCVBNM', 16))

        self.icon = icon

        self.getIcon()

        self.startArgs = [

            self.name,

            '--onefile',

            f'--key={self.key}',

            f'-i={self.icon}'

        ]

    def getIcon(self):

        self.icon = ''

*return* ''

    def build(self):

        PyInstaller.\_\_main\_\_.run(self.startArgs)

class BuilderOnWindows(Builder):

    """

    目前只能在Windows平台编译，见下文：  |  同时Linux平台使用os.fork()也有问题。

    https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods

    Warning

    The 'spawn' and 'forkserver' start methods cannot currently be used with “frozen” executables (i.e., binaries produced by packages like PyInstaller and cx\_Freeze) on Unix. The 'fork' start method does work.

    """

    def \_\_init\_\_(self, name, icon):

        super(BuilderOnWindows, self).\_\_init\_\_(name, icon)

    def getIcon(self):

        self.icon = os.path.join('img', self.icon)

*if* \_\_name\_\_ == '\_\_main\_\_':

    buildList = [

        'AisleCL.py'

    ]

*for* i in buildList:

        build0 = BuilderOnWindows(name=i, icon='a7rz2-w8k73-001.ico')

        build0.build()