OAR 游戏联机代理平台 v1.2

软

件

源

代

码

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('1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPLKJHGFDSAZ
XCVBNM', 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()
```