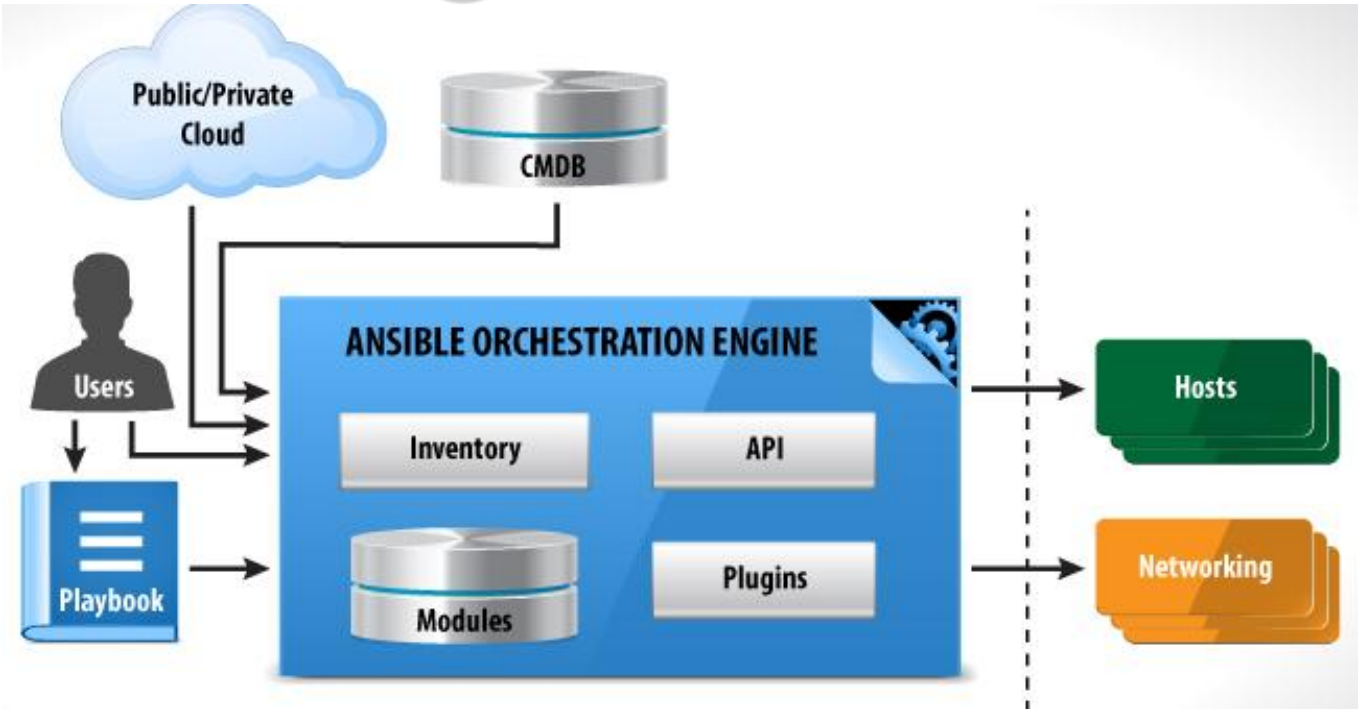
任务调度系统mschedule

概述

运维管理的几个阶段：
1、人工阶段
人工盯着服务器，出了问题，跑到机器前，翻日志，查状态，手动操作
2、脚本阶段
开始写一些自动化脚本，启动计划任务，自动启动服务，监控服务等
3、工具阶段
脚本功能太弱，开发了大量工具，某种工具解决某个特定领域的问题，常用的有ansible、puppet等
4、平台阶段
将工具整合，自主开发，实现标准化，实现自动化流程控制。
现今，平台已经向着智能化方向发展。

体会一下ansible

马哥Linux团队荣誉出品
http://www.ansible.com.cn/index.html



管理主机Server的要求
2.2+支持 python 3.5+

虚拟环境3.5.3安装ansible

```
$ pip install -v ansible==2.3.3
```

安装完成之后，先来配置下配置项~/.ansible.cfg。ansible执行的时候会按照以下顺序查找配置项:

- ANSIBLE_CONFIG (环境变量)
- ansible.cfg (当前目录下)
- ~/.ansible.cfg (用户家目录下)
- /etc/ansible/ansible.cfg

inventory文件
这个文件包含了托管节点的配置，默认ansible通过/etc/ansible/hosts查找，修改到指定的位置。
~/.ansible.cfg中指定inventory文件位置

```
[defaults]
inventory = /home/python/ansible_hosts
```

ansible_hosts文件

```
[remotes]
192.168.142.140
```

段remote表示分组。

```
(magedu353) [python@nodex cmdb]$ ansible all -m ping
```

all 表示所有分组。
执行出错了。一般来说，你要访问远程节点，需要权限。最简单的方式ssh，ansible恰好就是使用ssh的，需要配置ssh免密码登录。当然也可以使用ansible的其他组件完成。

```
(magedu353) [python@nodex cmdb]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/python/.ssh/id_rsa):
Created directory '/home/python/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/python/.ssh/id_rsa.
Your public key has been saved in /home/python/.ssh/id_rsa.pub.
The key fingerprint is:
5f:a2:58:fd:23:2d:d1:b1:96:ea:7f:fe:bd:4c:bb:72 python@nodex
The key's randomart image is:
```

```
+--[ RSA 2048]----+
|                 |
|                 |
|            .    |
|         . . +   |
|        S + *    |
|       o o O     |
|      . . = +  . |
|         . o o+Eo|
|          ...o+*=|
+-----------------+

(magedu353) [python@nodex cmdb]$ ssh-copy-id -i ~/.ssh/id_rsa.pub root@192.168.142.
140
The authenticity of host '192.168.142.140 (192.168.142.140)' can't be established.
RSA key fingerprint is da:da:db:c7:bb:a5:3f:4d:d4:a9:b1:4c:a6:4a:59:91.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.142.140' (RSA) to the list of known hosts.
root@192.168.142.140's password:
Now try logging into the machine, with "ssh 'root@192.168.142.140'", and check in:

  .ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.

(magedu353) [python@nodex cmdb]$ ansible all -m ping
192.168.142.140 | UNREACHABLE! => {
    "changed": false,
    "msg": "Authentication failed.",
    "unreachable": true
}
```

依然失败，使用ssh登录远程节点

```
(magedu353) [python@nodex cmdb]$ ssh root@192.168.142.140
```

成功登录远程节点，应该是配置问题。

ansible_hosts文件修改如下，指定登录的账户

```
[remotes]
```

```
192.168.142.140 ansible_ssh_user=root
```

```
(magedu353) [python@nodex cmdb]$ ansible all -m ping
192.168.142.140 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

执行成功。
测试一下，在远程节点执行命令

```
(magedu353) [python@nodex cmdb]$ ansible remotes -m shell -a 'hostname'
192.168.142.140 | SUCCESS | rc=0 >>
node
```

命令解释
-m 模块，shell执行shell命令。
-a args，模块参数。

托管节点
支持ssh就行，默认使用sftp，如果没有开启sftp，可以配置ansible.cfg配置成scp方式。

ansible更强大的功能，使用playbook（YAML）配置，请参考最上面的中文链接学习。

mschedule设计

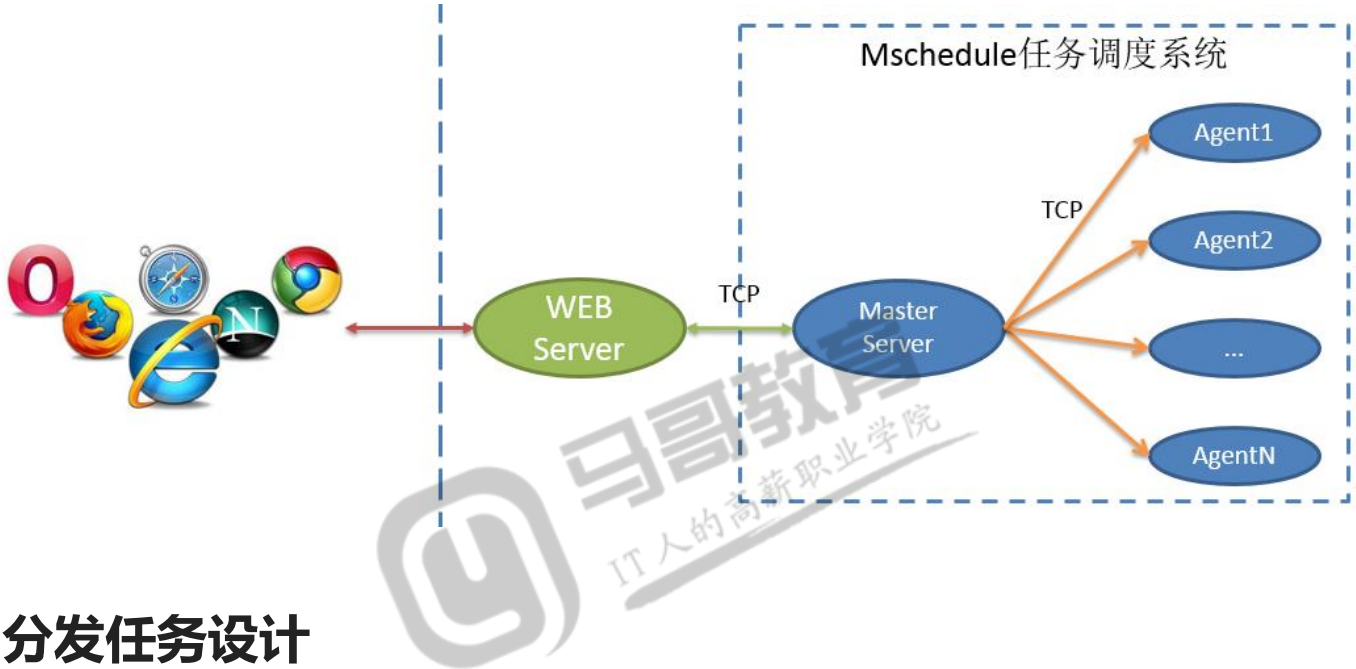
类似于ansible功能，更加简单，满足企业需求。
1、分发任务
分发脚本到目标节点上去执行
2、控制
控制并发，控制多少个节点同时执行
对错误做出响应。由用户设定，最多允许失败的比例或者数量，当超过范围时，需要终止任务执行。
可以终止正在执行的任务。
3、能跨机房部署
4、能对作业做版本控制，这个是辅助的需求，可以以后实现。

本项目的出发点，是只需要会使用shell脚本就可以了，而ansible、salt等需要学习特定的内部语言。
如果觉得ansible这样的工具不能满足需求，二次开发难度较高，其代码量不小，本身它们开发接

口不完善，而且熟悉它的架构也比较难，就算开发出来维护也难。
从这些项目上二次开发，等于拉一个分支，如果主分支有了新的特性，想合并也很困难。

自己开发，满足自己需求，完全适合自己需求。代码规模可控，便于他人接手维护。
自己开发就是造轮子，造轮子不是不好，不要一上来就设计打造一个超级牛的轮子，结果能力不足，项目失败。先构建一个适合的、能力所及、满足需求的轮子，后来再逐步完善。
我们经常听说，不要自己造轮子，结果发现轮子都被外国项目造出来了。
一般来说，越是自动化运维程度越高的公司，自己写的系统越多，因为满足他们需要的工具少。

做运维开发，只有深入到工具的底层原理，才能真正的理解其本质，才能去做新的工具的研发，或者工具的二次开发。当然，做其他开发也是同样的道理。



分发任务设计

分为有Agent、无Agent。

有Agent，被控节点需要安装或运行特殊的软件，和服务器端通信，服务器端把脚本、命令传给Agent端，有Agent来控制执行。

无Agent，被控节点不需要安装或运行特殊的软件，例如通过ssh。这其实也是有Agent，不过不是自己写的程序。
1、通用、简单、易实现。但管理不善，容易出安全问题。
2、并行效率不高。有Agent的并行执行可以不和管理服务器通信，可以并发很高。ssh执行要和master之间通信。
3、ssh连接是有状态的。任务执行的时候，master不能挂了，否则任务执行失败。

执行脚本

python有很多运行进程的方式，不过都过时了。
建议使用标准库subprocess模块，启动一个子进程。

`class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=())`

shell为True，则使用shell来执行args，建议args是一个字符串。

```
from subprocess import Popen
p = Popen('echo "hello"',shell=True)
```

构建项目

项目名称mschedule。

构建一个模块agent，模块下建一个executor.py。

```
import subprocess

class Executor:
    def __init__(self, script):
        self.script = script

    def run(self):
        p = subprocess.Popen(self.script, shell=True)
        p.wait()
```

项目根目录下，建立一个app.py

```
from agent.executor import Executor

if __name__ == '__main__':
    executor = Executor('echo "hello python"')
    executor.run()
```

运行成功，看来可以成功运行脚本。

agent设计

用户和Master Server通信，提交任务。
Master按照用户要求将任务分发到指定的节点上，这些节点上需有一个Agent和Master通信，接收Master发布的任务，并执行这些任务。

设计Agent时，应当注意，越简单越好，越简单Bug越少越稳定。

从本质上来说，Master、Agent设计是典型的CS编程模式。
Master作为CS中的Server，Agent作为CS中的Client。

1、注册信息
Agent启动后，需要主动联系Server，注册自己的信息。
信息包括：
我是谁，hostname、UUID。UUID保证唯一，因为主机名有可能重复。
来自哪里，IP地址是多少。需要Agent主动在信息中告诉Master。
其他信息，这个根据情况而定。
2、心跳信息
Agent定时向Master发送心跳包，包含UUID这个唯一标识，附带hostname和ip。
hostname和ip都可能变动，但是这个Agent不变，UUID也就不变。
其他信息也可以附加，例如增加一个flag，表示Agent上是否有任务在跑。
3、任务消息
Master分派任务给Agent，发送任务描述信息到Agent。
注意脚本字符串使用Base64编码。
4、任务结果消息
当Agent执行完任务，返回给Master该任务的状态码和输出结果。

以上Master、Agent之间需要传送消息，消息采用json 格式

消息设计

注册消息

```
{
  "type":"register",
  "payload":{
    "id":"uuid",
    "hostname":"xxx",
    "ip":[]
  }
}
```

心跳消息

```
{
  "type":"heartbeat",
  "payload":{
    "id":"uuid",
```

```
    "hostname":"xxx",
    "ip":[]
  }
}
```

任务消息

```
{
  "type":"task",
  "payload":{
    "id":"task-uuid",
    "script":"base64encode",
    "timeout":0,
    "parallel":1,
    "fail_rate":0,
    "fail_count":-1
  }
}
```

parallel：并行，表示同时执行的任务。
fail_rate：失败率，0表示不允许失败。
fail_count：失败数，-1不关心失败的数量和失败率。

执行结果消息

```
{
  "type":"result",
  "payload":{
    "id":"task-uuid",
    "agent_id":"agent-uuid",
    "code":0,
    "output":"base64encode"
  }
}
```

id，任务id。
agent_id：Agent是谁。
code：返回的状态码，0正常，非零错误，和linux的命令返回值一样。
output：输出的结果。字符串，Base64编码后返回。

代码实现

配置

agent.config模块，配置信息放这里

日志

项目根目录下utils.py

```
import logging

def getlogger(mod_name:str, filepath:str):
    logger = logging.getLogger(mod_name)
    logger.setLevel(logging.INFO)  # 单独设置
    logger.propagate = False  # 阻止传送给父logger
    handler = logging.FileHandler(filepath)
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter(fmt="%(asctime)s [%(name)s %(funcName)s] %(messag
e)s")
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    return logger
```

通信模块

原生的Socket编程太过底层，少使用。任何一门语言都要避免直接使用socket库开发，太过底层，难写难维护。

这次使用一个非常轻巧的、跨语言的RPC通信模块zerorpc。
它基于ZeroMQ和MessagePack。
http://www.zerorpc.io/

```
$ pip install zerorpc
```

Server代码

```
import zerorpc

class MyRPC:
    def hello(self, text): # RPC对外的接口，其中处理客户端传来的数据
```

```
        return "send back {}".format(text)


s = zerorpc.Server(MyRPC())
s.bind("tcp://0.0.0.0:9000") # 绑定
s.run() # 持久运行监听
```

Client代码

```
import zerorpc

c = zerorpc.Client()
c.connect('tcp://127.0.0.1:9000')
print(c.hello('test client'))
```

Client循环发送代码

```
import zerorpc
import threading

c = zerorpc.Client()
c.connect('tcp://127.0.0.1:9000')
e = threading.Event()

while not e.wait(3):
    print(c.hello('test client'))
    print('~~~~~~~~~~~~~~')
```

在agent模块下，创建cm模块，负责网络连接。

注意：不要把zerorpc的方法随便的放到线程中，会抛异常

注册消息实现

uuid
使用uuid.uuid4().hex获取一个uuid。一个节点起始运行的时候是没有uuid的，一旦运行会生成一个uuid，并持久化到一个文件中，下次运行先找这个文件，如果文件中有uuid，就直接读取，没有uuid，就重新生成并写入到该文件中。

hostname
windows和Linux取主机名方式不一样。

可以在所有平台使用socket.gethostname()获取主机名。

ip列表

```
$ pip install netifaces
```

netifaces.interfaces() 返回接口列表

netifaces.ifaddresses(interface) 取指定接口上的IP地址，返回信息如下：

```
{-1000: [{'addr': '00:ff:18:dd:f9:dc'}], 23: [{'netmask': 'ffff:ffff:ffff:ffff::/64
', 'addr': 'fe80::c9a:acee:5032:8d8%28', 'broadcast': 'fe80::ffff:ffff:ffff:ffff%28
'}]}
{-1000: [{'addr': '54:27:1e:5c:df:c4'}], 23: [{'netmask': 'ffff:ffff:ffff:ffff::/64
', 'addr': 'fe80::9c53:89a8:2e11:f302%27', 'broadcast': 'fe80::ffff:ffff:ffff:ffff%
27'}]}
{-1000: [{'addr': '54:27:1e:5c:df:c4'}], 2: [{'netmask': '255.255.255.0', 'addr': '
192.168.11.218', 'broadcast': '192.168.11.255'}]}
{-1000: [{'addr': '00:ac:c7:37:5d:92'}], 23: [{'netmask': 'ffff:ffff:ffff:ffff::/64
', 'addr': 'fe80::8863:575a:2ede:6aaf%25', 'broadcast': 'fe80::ffff:ffff:ffff:ffff%
25'}]}
{-1000: [{'addr': '44:8a:5b:43:88:8d'}]}
{-1000: [{'addr': '00:50:56:c0:00:01'}], 2: [{'netmask': '255.255.255.0', 'addr': '
192.168.56.1', 'broadcast': '192.168.56.255'}], 23: [{'netmask': 'ffff:ffff:ffff:ff
ff::/64', 'addr': 'fe80::7d65:4413:d602:6953%16', 'broadcast': 'fe80::ffff:ffff:fff
f:ffff%16'}]}
{-1000: [{'addr': '00:50:56:c0:00:08'}], 2: [{'netmask': '255.255.255.0', 'addr': '
192.168.142.1', 'broadcast': '192.168.142.255'}], 23: [{'netmask': 'ffff:ffff:ffff:
ffff::/64', 'addr': 'fe80::704e:ac33:39b3:91b9%17', 'broadcast': 'fe80::ffff:ffff:f
fff:ffff%17'}]}
{-1000: [{'addr': ''}], 2: [{'netmask': '255.0.0.0', 'addr': '127.0.0.1', 'broadcas
t': '127.255.255.255'}], 23: [{'netmask': 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff/
128', 'addr': '::1', 'broadcast': '::1'}]}
{-1000: [{'addr': '00:00:00:00:00:00:00:e0'}]}
```

这是一个字典，key为2就是ipv4地址。

每一个接口返回的ipv4地址是一个列表，也就是说可以有多个，ipv4地址描述是在addr上。

```
import netifaces

for interface in netifaces.interfaces():
    if 2 in netifaces.ifaddresses(interface).keys():
        for ip in netifaces.ifaddresses(interface)[2]:
            print(ip['addr'])
```

```
# 运行结果
192.168.11.218
192.168.56.1
192.168.142.1
127.0.0.1
```

如果拿到自己想要的ip，例如排除掉回环、多播等地址呢？

ipaddress库

```
import ipaddress

ips = ['127.0.0.1', '192.168.0.1', '169.254.123.1', '0.0.0.0', '239.168.0.255', '22
4.0.0.1', '8.8.8.8']
for ip in ips: # ips 是地址列表，ip是字典
    # ipaddress地址验证
    print(ip)
    ip = ipaddress.ip_address(ip)
    print("linklocal {}".format(ip.is_link_local)) # 169.254地址
    print("回环 {}".format(ip.is_loopback)) # 回环
    print("多播 {}".format(ip.is_multicast)) # 多播
    print("公网 {}".format(ip.is_global)) # 全球范围地址，公网的
    print("私有 {}".format(ip.is_private)) # 私有
    print("保留 {}".format(ip.is_reserved)) # 保留地址
    print("版本 {}".format(ip.version)) # 4
    print('-----------------')
```

在agent模块新建msg.py

```
import netifaces
import ipaddress
import os
import uuid
import socket

class Message:
    def __init__(self, myidpath):
        # 从文件中读取主机的UUID
        if os.path.exists(myidpath):
            with open(myidpath) as f:
                self.id = f.readline().strip()
```

```
        else:
            self.id = uuid.uuid4().hex
            with open(myidpath, 'w') as f:
                f.write(self.id)

    def _get_addresses(self):
        """获取主机上所有接口可用的IPv4地址"""
        addresses = []

        for interface in netifaces.interfaces():
            if 2 in netifaces.ifaddresses(interface).keys():
                for ip in netifaces.ifaddresses(interface)[2]:
                    # ipaddress地址验证
                    #print(ip)
                    ip = ipaddress.ip_address(ip['addr'])
                    if ip.version != 4: # 版本
                        continue
                    if ip.is_link_local: # 169.254地址
                        continue
                    if ip.is_loopback: # 回环
                        continue
                    if ip.is_multicast: # 多播
                        continue
                    if ip.is_reserved: # 保留
                        continue

                    addresses.append(str(ip))
        return addresses

    def reg(self):
        """生成注册消息"""
        return {
            'type':'reg',
            'payload':{
                'id':self.id,
                'hostname':socket.gethostname(),
                'ip':self._get_addresses()
            }
        }
```

完成心跳信息的方法，它应该和reg是相似的。

```
    def heartbeat(self):
        """生成心跳消息"""
        return {
            'type':'heartbeat',
            'payload':{
                'id':self.id,
                'hostname':socket.gethostname(),
                'ip':self._get_addresses()
            }
        }
```

问题
心跳信息是非常频繁的发送，几秒一次的。
每一次都需要查询一下IP列表，把IP都发过去一次，浪费计算资源。
后期可以考虑，单独跑一个线程（进程）处理IP地址和主机名等反复使用的信息。

消息发送

完成agent.cm模块代码。

一旦Agent启动，就会尝试和Master建立TCP连接发送数据。
假设Master有2个接口方法reg、heartbeart可供调用。

agent.config模块

```
MASTER_URL = "tcp://127.0.0.1:9000"
MYID_PATH = "o:/myid"
```

agent/cm.py

```
import zerorpc
import threading
from ..msg import Message

class ConnectionManager:
    def __init__(self, master_url, message:Message):
        self.master_url = master_url
        self.message = message # 对象
        self.client = zerorpc.Client()
```

```
        self.event = threading.Event()

    def start(self):
        self.client.connect(self.master_url)
        # 注册
        self.client.reg(self.message.reg())
        # 心跳
        while not self.event.wait(interval):
            self.client.heartbeat(self.message.heartbeat())

    def shutdown(self):
        self.event.set()
        self.client.close()

    def join(self): # 让主线程阻塞
        self.event.wait()
```

特别注意这个join方法，让调用的主线程阻塞。

Agent类

在agent模块的 `__init__.py` 中构建Agent类。
其作用是：管理连接，开启连接，负责重连，关闭连接。

```
from .cm import ConnectionManager
from .msg import Message
from .config import MASTER_URL, MYID_PATH
import threading

class Agent:
    def __init__(self):
        self.msg = Message(MYID_PATH)
        self.cm = ConnectionManager(MASTER_URL, self.msg)
        self.event = threading.Event()

    def start(self):
        while not self.event.is_set(): # 重连
            try:
                self.cm.start()
            except Exception as e:
                self.cm.shutdown()
```

```
            self.event.wait(3)

    def shutdown(self):
        self.event.set()
        self.cm.shutdown()
```

app.py中测试

```
from agent import Agent

if __name__ == '__main__':
    agent = Agent()
    try:
        agent.start()
    except KeyboardInterrupt:
        agent.shutdown()
```

运行后发现未连接到服务器，超时抛异常。

ConnectionManager重连机制实现

本来考虑使用多个接口实现reg、heartbeat等方法，从本质上来说，它们都是一个方法。
因此修改为使用同一个接口sendmsg方法和Master通信。

在connect、sendmsg方法调用的时候都有可能出现异常，一旦异常，就set event。Agent实例中理解不阻塞了，就可以重连了。
直到连接上为止，或者直到shutdown为止

agent/cm.py

```
import zerorpc
import threading
from .msg import Message
from utils import getlogger

logger = getlogger(__name__, 'o:/cm.log')

class ConnectionManager:
    def __init__(self, master_url, message:Message):
        self.master_url = master_url
        self.message = message # 对象
```

```
        self.client = zerorpc.Client()
        self.event = threading.Event()

    def start(self, hbinterval=5):
        try:
            self.event.clear() # 重置event
            # 连接
            self.client.connect(self.master_url)
            # 注册
            self._send(self.message.reg())
            # 心跳循环
            while not self.event.wait(hbinterval):
                self._send(self.message.heartbeat())
        except Exception as e:
            logger.error('Failed to connect to master. Error:{}'.format(e))
            raise e

    def shutdown(self):
        self.event.set()
        self.client.close()

    def _send(self, msg):
        ack = self.client.sendmsg(msg)
        logger.info(ack)

    def join(self): # 让主线程阻塞
        self.event.wait()
```

好，agent端先告一段落，开始Master端开发，等两边调试通再做进一步开发。

思考

上面的join方法还有用吗？需不需要去除？理由是什么