CMDB介绍

以下是根据某公司的CMDB项目的学习和分析，通过学习来进行运维开发方向的学习

介绍:

CMDB：配置管理数据库(Configuration Management Database,CMDB)是一个逻辑数据库，包含了配置项全生命周期的信息以及配置项之间的关系(包括物理关系、实时通信关系、非实时通信关系和依赖关系)，主要用于服务器资产的采集

背景:

开发运维自动化的平台

- 发布系统
- 监控
- 配置管理
- 自动装机
- 堡垒机

学后总结

各类技术点 + 流程

```
1. 配置
    用户自定义的配置 + 项目的默认配置
    类  os.environ
    importlib.import_module('文件路径')  导入模块
    反射
    - getattr 获取属性
    - setattr 设置属性、__setattr__()
    dir  获取对象的属性名

2. 支持多种模式 + 可扩展
    - agent
    - ssh
    - salt

    - 配置
        ENGINE = 'agent'
        ENGINE_DICT = {
            'agent': 'src.engine.agent.AgentHandler',
            'ssh': 'src.engine.ssh.SshHandler',
            'salt': 'src.engine.salt.SaltHandler',
            'ansible': 'src.engine.ansible.AnsibleHandler',
        }

    - importlib 导入模块  + 反射

3. 类的约束
    - 抽象类 + 抽象方法  abc
```

```
    - 继承 + 抛出异常

4. 资产采集的插件  可插拔式的设计
    - 配置
        PLUGINS_DICT = {
            'disk': 'src.plugins.disk.Disk',
            'memory': 'src.plugins.memory.Memory',
            'nic': 'src.plugins.nic.NIC',
        }
    - importlib 导入模块  + 反射

5. 根据不同的模式 执行命令的方式不一样
    给handler对象定义cmd的方法
    agent  --  subprocess
    ssh    --  paramiko
    salt   --  salt  subprocess
    给插件传入handler对象，执行cmd的方法

6. 支持 win 和 linux
   通过命令的执行获取操作系统
   插件中定义win和linux方法

7. debug 模式
    本地测试时 读取文件
8. requests
    requests.get(url)
    ret = requests.post(url,data='json字符串', headers={})
    ret.text 文本
    ret.json()  反序列化

9. 线程池
    from concurrent.futures import ThreadPoolExecutor
    pool = ThreadPoolExecutor(20)
    pool.submit(task, args)

10. djangorestframework
    api -  CBV
    APIView
    Response
    request.data
```

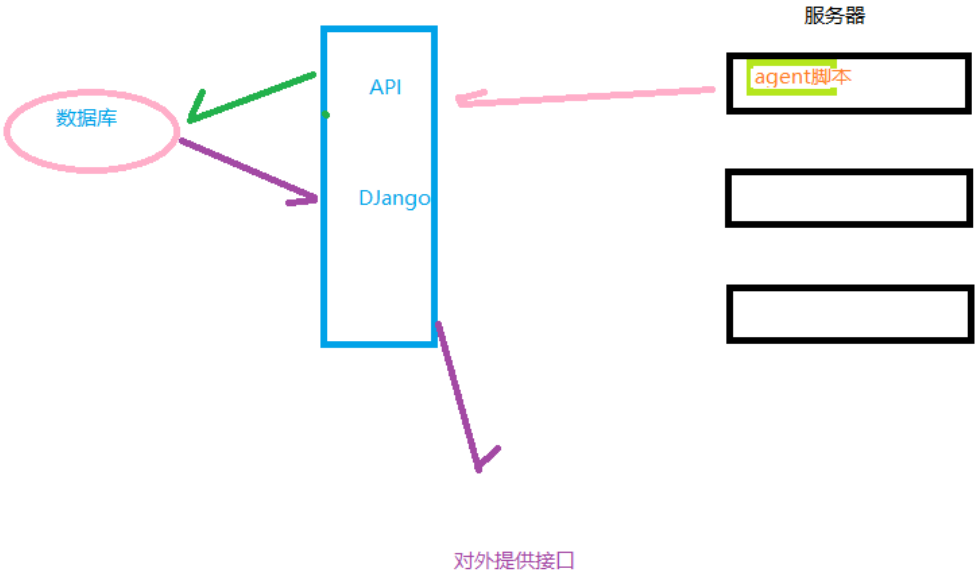
方式一：agent模式：

主要实现思路：

- agent：一个脚本，用于采集服务器资源

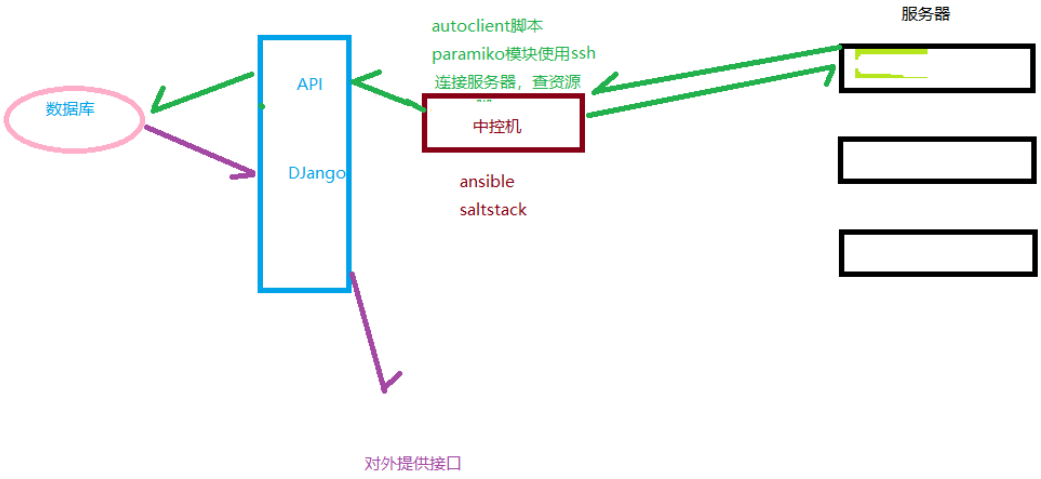
  ```
  1. 每台服务器装一个agent脚本
  2. 每天定时启动这个脚本
  3. 采集完成信息后发送一个机器 保存信息
  ```

- agent脚本（通过agent主动的发送数据）
- api（django）
- 后台管理



方式二：ssh模式：主动的获取服务器数据

- 使用ssh 实现



服务器端需要使用Paramiko模块

```
import requests
import paramiko

#密钥
# private_key = paramiko.RSAKey.from_private_key_file('/home/auto/.ssh/id_rsa')

# 创建SSH对象
ssh = paramiko.SSHClient()
# 允许连接不在know_hosts文件中的主机
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
# 连接服务器
# ssh.connect(hostname='c1.salt.com', port=22, username='maple',
key=private_key)
ssh.connect(hostname='47.95.217.144', port=22,
username='root',password='44771!')

# 执行命令
stdin, stdout, stderr = ssh.exec_command('ls')
```

```
# 获取命令结果
result = stdout.read()

# 关闭连接
ssh.close()

url = 'http://127.0.0.1:8000/api/asset/'
ret = requests.post(
    url=url,
    # 注意回复格式，urlencode
    data={'msg':result},
)

print(ret.text)
print(ret.json())
```

相对于的django的视图逻辑

```
from django.http import HttpResponse,JsonResponse

def asset(request):
    print(request.POST)
    return JsonResponse({'code':200,'msg':'保存成功'})
```

SaltStack

安装和配置

官网地址

- 安装配置源，主的一方

  ```
  sudo yum install https://repo.saltstack.com/py3/redhat/salt-py3-repo-latest.el7.noarch.rpm

  sudo yum install salt-master  #安装主仆中的主，的服务器
  ```

  配置

  ```
  [root@MyHost ~]# vim /etc/salt/master
  ...
  # The address of the interface to bind to:  #修改
  interface: 0.0.0.0
  ```

- 仆的一方

  ```
  sudo yum install https://repo.saltstack.com/py3/redhat/salt-py3-repo-latest.el7.noarch.rpm

  sudo yum install salt-minion
  ```

  配置

```
[root@MyHost ~]#vi /etc/salt/minion
...
# resolved, then the minion will fail to start.
master: 47.95.217.144
```

- 启动

```
[root@MyHost ~]# systemctl start salt-master  #主启动
```

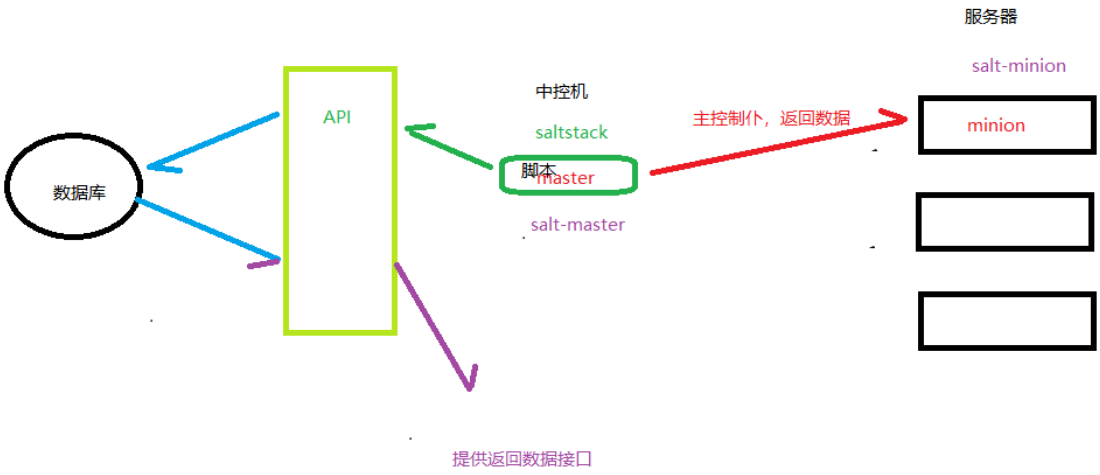
仆启动

```
[root@MyHost ~]# systemctl start salt-minion
```

- 相关的授权

```
salt-key -L   查看所有授权
salt-key -A   接受所有的授权

[root@localhost ~]# salt-key -L   # 主中执行结果
Accepted Keys:
Denied Keys:
Unaccepted Keys:
192.168.109.128
Rejected Keys:

[root@localhost ~]# salt-key -A   #接受所有仆ip
The following keys are going to be accepted:
Unaccepted Keys:
192.168.109.128
Proceed? [n/Y] y
Key for minion 192.168.109.128 accepted.
[root@localhost ~]# salt-key -L   #这就接受了那个ip了
Accepted Keys:
192.168.109.128
[root@localhost ~]# salt '192.168.109.128' cmd.run 'ls'  # 执行命令
192.168.109.128:
    anaconda-ks.cfg

[root@localhost ~]#salt '*' cmd.run 'ls'   #所有仆人执行ls命令

ps:没有ip或主机名的时候关闭防火墙
    iptables -F
    systemctl stop firewalld
    systemctl disable firewalld
```

方式三：通过 saltstack/ansible 的主仆关系进行控制



方式一：在主控机上（中控机）执行

```
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import subprocess
>>> ret = subprocess.getoutput("salt '{}' cmd.run '{}'".format('*','ls'))
ret>>> ret
'192.168.109.128:\n    anaconda-ks.cfg'
```

方式二： 由于saltstack是python开发的，可以使用salt模块完成

```
import salt.client   # 需要安装模块
local = salt.client.LocalClient()
result = local.cmd('c2.salt.com', 'cmd.run', ['ifconfig'])  #命令
```

总结

```
agent  （适用于机器多）----给每个主机装脚本
    1. 每台服务器都要装agent脚本
    2. 每天定时启动，自动采集资产
    3. 通过subprocess模块在本地执行命令，获取到硬件的信息
    4. 对数据进行处理，通过requests模块发送到api
    5. api接受数据保存到数据库中

ssh （适用于机器少） ----- 脚本放在中控机中，不需要安装其他软件
    1. 中控机上装agent脚本
    2. 每天定时启动，从api中获取到今天要采集那些主机，自动采集资产
    3. 通过ssh远程（paramiko模块）连接上主机，在被控机上执行命令，获取到硬件的信息
    4. 对数据进行处理，通过requests模块发送到api
    5. api接受数据保存到数据库中


salt （必须装软件）----- 中控机中
    1. 中控机上装agent脚本(中控机装salt-master,被控机装salt-minion)
    2. 每天定时启动，自动采集资产
    3. 通过saltstack远程连接上主机，在被控机上执行命令，获取到硬件的信息
    4. 对数据进行处理，通过requests模块发送到api
    5. api接受数据保存到数据库中
```

补充：

在使用salt方式时，如果被控主机全是ip，会显得比较麻烦，难得记，所以这里我们使用的是主机名去。

- 首先在被控主机中修改主机名

```
[root@localhost ~]# hostnamectl  set-hostname minion1
[root@localhost ~]# init 6

[root@localhost ~]# rm -rf /etc/salt/minion_id   # 删除之前自动创建的id记录，重启服务
```

- 在主的一方先干掉之前的用ip的仆

```
[root@localhost ~]# salt-key -d 192.168.109.128  # 删除ip仆人
The following keys are going to be deleted:
Accepted Keys:
192.168.109.128
Proceed? [N/y] y
Key for minion 192.168.109.128 deleted.
[root@localhost ~]# salt-key -L
Accepted Keys:

[root@localhost ~]# salt-key -A
The following keys are going to be accepted:
Unaccepted Keys:
minion1

[root@localhost ~]# salt 'minion1' cmd.run 'ls'
minion1:
    anaconda-ks.cfg
```

以上介绍了三种方式采集服务器信息，那么为了开发一种能结合三种模式采集信息的工程，我们可以使用类来进行功能拆分，使用软件开发规范，拆分项目：

报错回溯模块

之前我们使用 try ... except Exception as e: print(e) 这种方式获得信息，并不完整，也不知道哪里报错，所以这里我们使用回溯报错模块

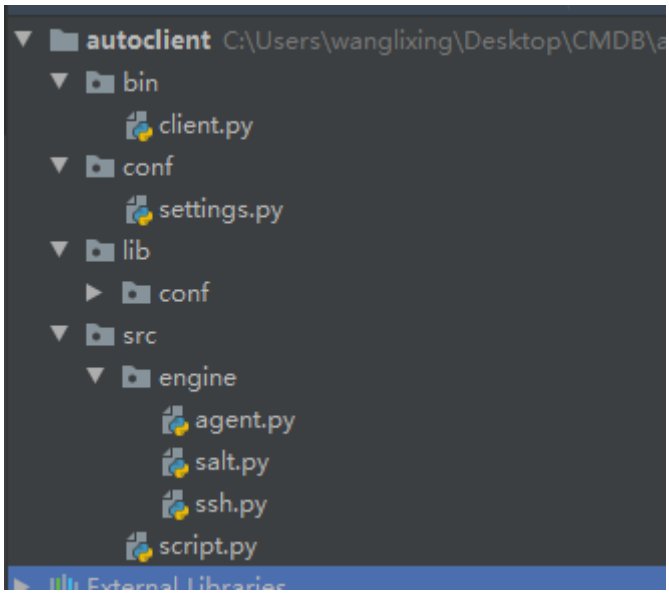
```
import traceback

def disk():
    int('saaa')

def run():
    try:
        disk()
    except Exception:
        ret = traceback.format_exc()
        print(ret,type(ret))  #写入日志，以后

run()
```

客户端项目目录结构

- bin 可执行文件
- log 日志
- config 配置
- lib 公共库
- src 逻辑代码

借鉴于Django的settings懒加载

通过懒加载settings，将系统settings和用户settings赋值个一个setting对象，那么就可以实现属性的完整性和安全性，同时也借助了os.envirn来执行用户可调的settings文件目录：



- 代码如下：

  client.py文件

```
import os
import sys
ret = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
sys.path.insert(0,ret)  #手动添加环境路径，Pycharm默认添加


if __name__ == '__main__':
    # os.environ是系统的环境变量,借鉴于django的manager.py,也就是相当于字典，可以设置键值，后面后可以找到该键值
    os.environ.setdefault('USER_SETTINGS','conf.settings')
    from src.script import run
    run()
```

setting.py文件

```
# 用户配置settings
USER = 'root'
PWD='root1234'

# 设置采集模式
ENGINE = 'agent'
```

lib/conf/__init__.py

```
import os
import importlib
from . import global_settings

class Settings:
    def __init__(self):
        #系统settings
        for i in dir(global_settings): # 加载系统 设置，系统应该放最前面，会进行覆盖
            if i.isupper():  # 全局设置必须大写
                v = getattr(global_settings, i)  # 使用反射获取属性/方法
                self.__setattr__(i, v)  # 也可以使用setattr(self,i,v)

        # 用户settings
        path = os.environ.get('USER_SETTINGS')
        # 通过path路径拿到模块对象 module,导入用户的settings，在项目启动时加载到了环境变量中
        module = importlib.import_module(path)
        # 但是不不知道用户settings中 保证他会写这些属性，可以通过dir看module属性
        for i in dir(module):
            if i.isupper():  # 全局设置必须大写
                v = getattr(module,i)  # 使用反射获取属性/方法
                self.__setattr__(i,v)  # 也可以使用setattr(self,i,v)

    # def __setattr__(self, key, value):
    #     # 通过__setattr__可以做二次开发
    #     if key == 'USER':
    #         key = 'OK'
    #     super().__setattr__(key,value)


# 借鉴于django的懒加载setting+global_settings, 全部的setting配置都给了settings
```

```
settings = Settings()
# 下次在其他模块中通过import 导入settings默认是单例
```

global_settings.py

```
# 系统settings
USER = 'root'
PWD='root1234'
XXX ='X'
```

src/script.py

```
from lib.conf import settings
#  懒加载settings，将触发所有的配置，并加载给settings

def run():
    print('ok',settings.USER)
    print('ok',settings.XXX)
```

采集ENGINE的设计

和其他项目类似，为了在run() 程序中使用到反射，省去过多的if判断，以及满足项目的 "开放封闭"原则，那么我们可以通过抽象类来对，采集数据的类进行约束。

- 方式一： 抽象类+抽象方法，类似于go语言中的接口

```
import abc

class Person(metaclass=abc.ABCMeta):  #指定元类，以固定__call__来定制方法
    @abc.abstractmethod
    def talk(self):
        print('xx')

class Chinese(Person):
    def talk(self):
        print('中国话')

p = Chinese()
```

- 方式二(推荐,简单)：继承 + 抛出异常，强制性，没有就找父亲，然后抛错

```
class Person():
    def talk(self):
        raise NotImplementedError('talk()  must be Implemented')

class Chinese(Person):
    pass

    # def talk(self):
    #     print('中国话')

p = Chinese()
p.talk()
```

注意：if的可扩展性很差，工作中尽量使用反射，也就是说：要么创建对象，要么创建字典，在模板中 ——> 字典 + 字符串切割 + importlib + 类的反射

动态加载采集方式

- 代码如下：

  src/engine/base.py

```
class BaseHandler:

    def handler(self):
        raise NotImplementedError('handler 函数必须显示实现')
```

  src/engine/agent.py： 其他模式也是这样

```
from .base import BaseHandler

class AgentHandler(BaseHandler):

    def handler(self):
        """
        收集硬件信息 汇报给API
        :return:
        """
        print('agent')
```

  conf/settings.py

```
USER = 'root'
PWD='root123'

# 设置采集模式
ENGINE = 'ssh'

# 采用字典可以支持反射
ENGINES_DICT = {
    'agent':'src.engine.agent.AgentHandler',
    'ssh':'src.engine.ssh.SshHandler',
    'salt':'src.engine.salt.SaltHandler',
}
```

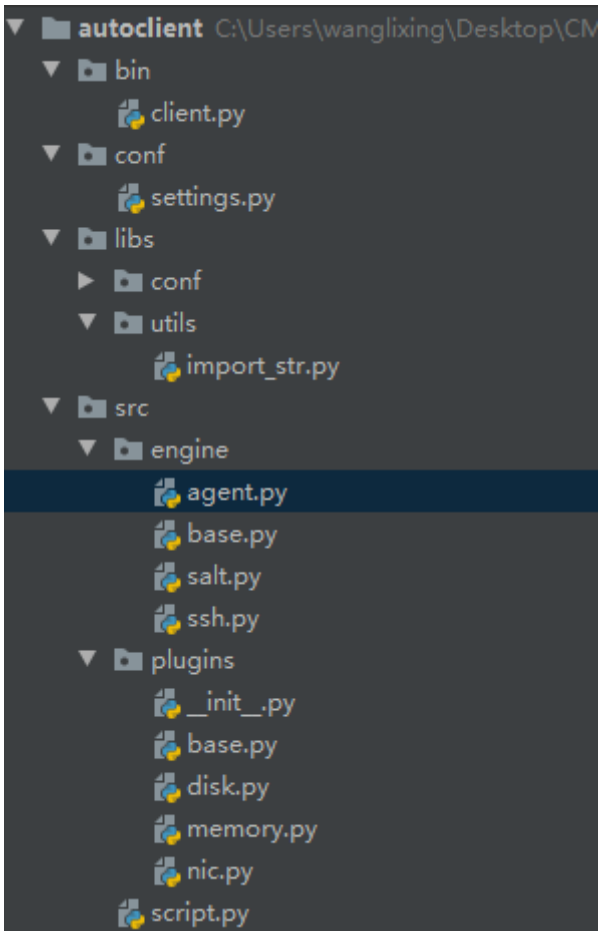
  src/script.py: 反射

```
import importlib
from lib.conf import settings
#  懒加载settings，将触发所有的配置，并加载给settings

def run():
    """
    程序的入口
    :return:
    """
    # 获得对应的采集的类路径
```

```
    class_path = settings.ENGINES_DICT.get(settings.ENGINE)
    module_str,cls_str = class_path.rsplit('.',maxsplit=1)
    # 通过importlib模块来动态路径获取 模块对象，并反射取类
    module = importlib.import_module(module_str)
    cls = getattr(module,cls_str)
    obj = cls()
    obj.handler()
```

采集硬件可插拔设计

原理和采集模式可拓展类似，通过在setting中配置，可以完成相关的硬件的插拔



- 代码如下： settings.py

```
# 用户配置settings
USER = 'root'
PWD='root1234'

SSH_USER='root'
SSH_PWD='root1234'


# 设置采集模式
ENGINE = 'agent'

# 采集模式，采用字典可以支持反射
ENGINES_DICT = {
    'agent':'src.engine.agent.AgentHandler',
```

```
    'ssh':'src.engine.ssh.SshHandler',
    'salt':'src.engine.salt.SaltHandler',
}

# 采集数据类型
PLUGINS_DICT = {
    'disk':'src.plugins.disk.Disk',
    'memory':'src.plugins.memory.Memory',
    'nic':'src.plugins.nic.NIC',
}
```

动态加载类，单独放在libs/utils中了，因为经常使用：`import_str.py`

```
import importlib

def get_class(path_str):
    module_str,cls_str = path_str.rsplit('.',maxsplit=1)
    # 通过importlib模块来动态路径获取 模块对象，并反射取类
    module = importlib.import_module(module_str)
    cls = getattr(module,cls_str)
    return cls
```

script.py

```
import importlib
from libs.conf import settings
#  懒加载settings，将触发所有的配置，并加载给settings
from libs.utils.import_str import get_class

def run():
    """
    程序的入口
    :return:
    """
    # 获得对应的采集的类路径
    class_path = settings.ENGINES_DICT.get(settings.ENGINE)
    cls = get_class(class_path)
    obj = cls()
    obj.handler()
```

disk.py 类似的插件写法如下

```
from .base import BassPlugin

class Disk(BassPlugin):
    def process(self,handler,hostname=None):
        ret = handler.cmd('wmic logicaldisk',hostname)
        return ret[:20]
```

plugins/__init__.py, 用于根据setting中的plugins配置加载对应的处理模块，使用反射

```
from libs.conf import settings
from libs.utils.import_str import get_class

def get_sever_info(handler,hostname=None):
```

```
    """
    根据配置，采集对应的插件信息
    :return:
    """
    info = {}
    for name,plugin in settings.PLUGINS_DICT.items():
        cls = get_class(plugin)
        obj = cls()
        ret = obj.process(handler,hostname)
        info[name] = ret

    return info  # 每个插件的信息返回给handler
```

系统平台区分（Win/Linux）

主要基于插件中的父类base.py进行扩展,plugins/base.py

- 代码如下：

```
class BassPlugin:

    def get_os(self,handler,hostname=None):
        ret = handler.cmd('uname',hostname) #linux 会返回Linux
        return ret

    # 会自动触发父级的process方法，在此判断操作系统，并调用self的linux，但必须自己有
    def process(self,handler,hostname=None):
        os = self.get_os(handler,hostname)
        if os == 'Linux':
            return self.linux(handler,hostname)
        else:
            return self.win(handler,hostname)

    def linux(self,handler,hostname=None):
        raise NotImplementedError('linux() must be Implement')

    def win(self,handler,hostname=None):
        raise NotImplementedError('win() must be Implement')
```

之后写插件的使用，只需要写两种模式下面的命令即可，如：disk.py

```
from .base import BassPlugin

class Disk(BassPlugin):
    # 必须显式实现，否则到了父类那里会抛错
    def linux(self,handler,hostname=None):
        ret = handler.cmd('ls',hostname)
        return 'Linux disk'

    def win(self,handler,hostname=None):
        ret = handler.cmd('dir',hostname)
        return ret[:20]
```

开发debug模式，便于调试

为了便于在win上开发，可以设置一个debug模式

- 基于插件的base.py

```
from libs.conf import settings

class BasePlugin:
    def __init__(self):
        self.debug = settings.DEBUG   #加载全局设置，以后继承类都可以使用
        self.base_dir = settings.BASE_DIR

    def get_os(self,handler,hostname=None):
        ret = handler.cmd('uname',hostname) #linux 会返回Linux
        return 'Linux'

    # 会自动触发父级的process方法，在此判断操作系统，并调用self的linux，但必须自己有
    def process(self,handler,hostname=None):
        os = self.get_os(handler,hostname)
        if os == 'Linux':
            return self.linux(handler,hostname)
        else:
            return self.win(handler,hostname)

    def linux(self,handler,hostname=None):
        raise NotImplementedError('linux() must be Implement')

    def win(self,handler,hostname=None):
        raise NotImplementedError('win() must be Implement')
```

disk.py

```
import os,re
from .base import BasePlugin

class Disk(BasePlugin):

    def linux(self, handler, hostname=None):
        if self.debug:
            # 读取文件
            with open(os.path.join(self.base_dir, 'files', 'disk.out')) as
f:
                ret = f.read()

        else:
            ret = handler.cmd('sudo MegaCli  -PDList -aALL', hostname)
        return self.parse(ret)

    def win(self, handler, hostname=None):
        ret = handler.cmd('wmic logicaldisk', hostname)
        return ret[:20]

    def parse(self, content):
        """
        解析shell命令返回结果
        :param content: shell 命令结果
```

```
        :return:解析后的结果
        """
        response = {}
        result = []
        for row_line in content.split("\n\n\n\n"):
            result.append(row_line)
        for item in result:
            temp_dict = {}
            for row in item.split('\n'):
                if not row.strip():
                    continue
                if len(row.split(':')) != 2:
                    continue
                key, value = row.split(':')
                name = self.mega_patter_match(key)
                if name:
                    if key == 'Raw Size':
                        raw_size = re.search('(\d+\.\d+)', value.strip())
                        if raw_size:
                            temp_dict[name] = raw_size.group()
                        else:
                            raw_size = '0'
                    else:
                        temp_dict[name] = value.strip()
            if temp_dict:
                response[temp_dict['slot']] = temp_dict
        return response

    @staticmethod
    def mega_patter_match(needle):
        grep_pattern = {'Slot': 'slot', 'Raw Size': 'capacity', 'Inquiry':
'model', 'PD Type': 'pd_type'}
        for key, value in grep_pattern.items():
            if needle.startswith(key):
                return value
        return False
```

完善处理函数handler

在agent模式下，因为是每个主机脚本，所以直接提交就是了，不需要考虑同步/异步，但是在ssh/salt方式下，会涉及到根据后端API返回要查询的配置来进行获取资源，那么就会出现同步效果，这里我们起一个线程池来解决问题

- 代码如下：engine/base.py

```
from libs.conf import settings
import requests,json
from ..plugins import get_sever_info
from concurrent.futures import ThreadPoolExecutor

class BaseHandler:
    def __init__(self):
        self.asset_url = settings.POST_ASSET_URL

    def handler(self):
```

```
        """
        收集硬件信息 汇报给API
        :return:
        """
        raise NotImplementedError('handler 函数必须显示实现')


class SshAndSaltHandler(BaseHandler): #归一化处理，整合ssh/salt的handler

    def handler(self):
        # 获取要采集信息的主机
        ret = requests.get(
            url=self.asset_url
        )
        # 反序列化
        host_list = ret.json()
        pool = ThreadPoolExecutor(20)

        for host_name in host_list:
            pool.submit(self.task,host_name)

    def task(self,host_name):
        info = get_sever_info(self, host_name)
        print(info)
        ret = requests.post(
            url=self.asset_url,
            data=json.dumps(info).encode('utf-8'),  # 需要编码，否则过去解码不行
            headers={'content-type': 'application/json'}  # 没有会抛415
        )
        print(ret.text)
```

ssh.py

```
from .base import BaseHandler,SshAndSaltHandler
from libs.conf import settings

class SshHandler(SshAndSaltHandler):
    def cmd(self,command,hostname=None):
        import paramiko
        # 私钥
        # private_key =
paramiko.RSAKey.from_private_key_file('/home/auto/.ssh/id_rsa')

        # 创建SSH对象
        ssh = paramiko.SSHClient()
        # 允许连接不在know_hosts文件中的主机
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        # 连接服务器
        ssh.connect(hostname=hostname, port=settings.SSH_PORT,
username=settings.SSH_USER, password=settings.SSH_PWD)
        stdin, stdout, stderr = ssh.exec_command(command)
        result = stdout.read()
        ssh.close()
        return result
```

同时后端服务器的代码

```
from rest_framework.views import APIView
from rest_framework.response import Response

class Asset(APIView):
    # 处理ssh/salt，最初来此获取要检查的设备的ip
    def get(self,request):
        host_list = ['192.168.109.128','192.168.109.129']*10
        return Response(host_list)

    def post(self,request):
        print(request.data)
        return Response('xxx')
```

错误处理以及request封装数据

我在读项目中的关于错误处理的时候，发现了一个很有意义的东西，传统的错误处理，会将错误信息添加到一个字典中，在原有的数据结构中添加一层，如下：

```
import os,re
import traceback
from .base import BasePlugin

class Disk(BasePlugin):
    def linux(self, handler, hostname=None):
        # 在原有的数据结构中再次添加一层
        result = {
            'status':True,
            'error':'',
            'data':None
        }

        try:
            if self.debug:
                # 读取文件
                with open(os.path.join(self.base_dir, 'files', 'disk.out')) as
f:
                    ret = f.read()
            else:
                ret = handler.cmd('sudo MegaCli  -PDList -aALL', hostname)
            result['data'] = self.parse(ret)

        except Exception:
            error = traceback.format_exc()
            result['error'] = error
            result['status'] = False
        return result

    def win(self, handler, hostname=None):
        ret = handler.cmd('wmic logicaldisk', hostname)
        return ret[:20]

    def parse(self, content): pass
```

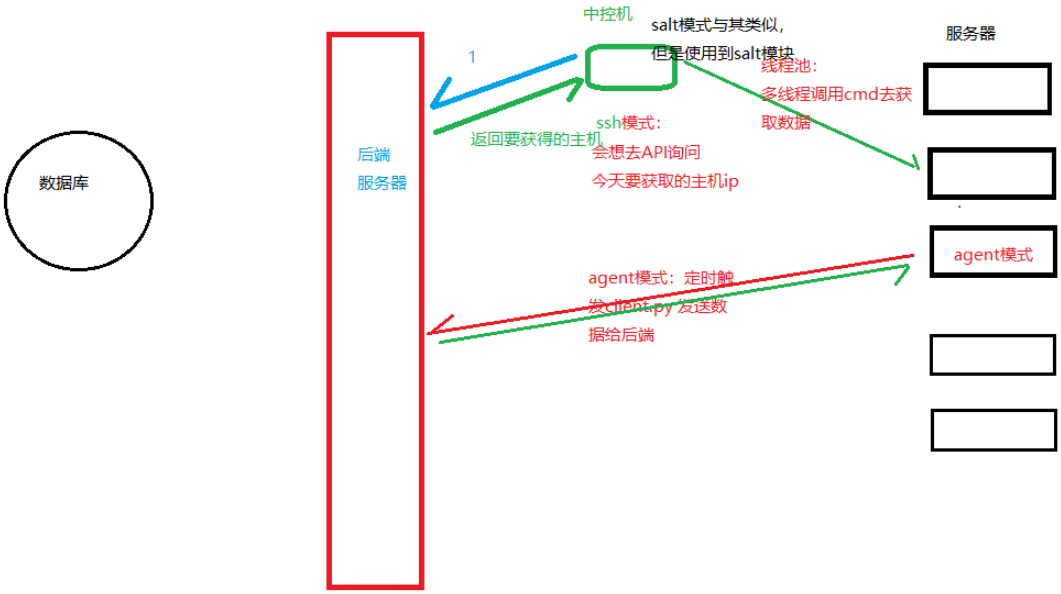
显然上面的方法是可以对错误信息进行整理的，但是我们这里想使用 类.__dict__ 来返回属性，会显得更整些。

再libs/utils/response.py

```
class BaseResponse:

    def __init__(self):
        self.status = True
        self.error = ''
        self.data = None

    @property
    def dict(self):
        return self.__dict__
```

engine/disk.py

```
import os,re
import traceback
from .base import BasePlugin
from libs.utils.response import BaseResponse

class Disk(BasePlugin):

    def linux(self, handler, hostname=None):
        # 将数据封装成一个对象中，调用__dict__返回
        response = BaseResponse()
        try:
            if self.debug:
                # 读取文件
                with open(os.path.join(self.base_dir, 'files', 'disk.out')) as
f:
                    ret = f.read()
            else:
                ret = handler.cmd('sudo MegaCli  -PDList -aALL', hostname)
            response.data = self.parse(ret)  #正常返回

        except Exception:
            error = traceback.format_exc()
            response.status = False
            response.error  = error
        return response.dict  #携带错误返回

    def win(self, handler, hostname=None):
        ret = handler.cmd('wmic logicaldisk', hostname)
        return ret[:20]

    def parse(self, content):pass
```



日志处理

Python中的日志处理模块的两种形式：

- 单文件日志：记录日志的功能，只能将日志记录在单文件中，如果想要设置多个日志文件，logging.basicConfig将无法完成，需要自定义文件和日志操作对象。

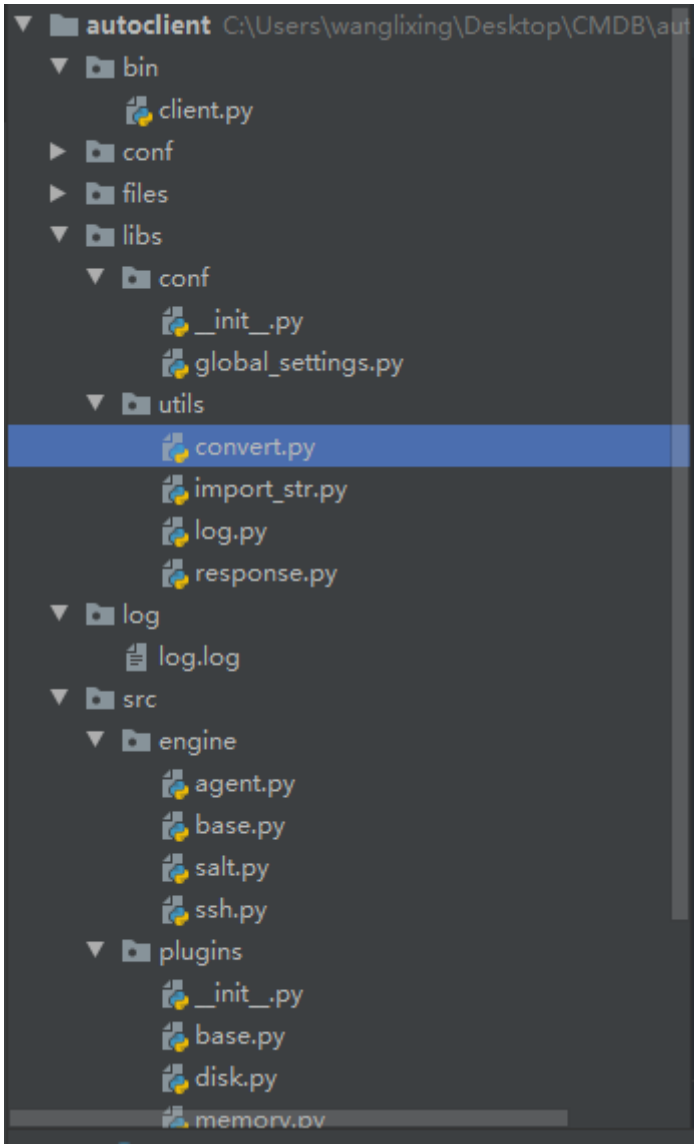
```
import logging

logging.basicConfig(filename='log.log',
                    format='%(asctime)s - %(name)s - %(levelname)s -%
(module)s:  %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S %p',
                    level=10)

logging.debug('debug')
logging.info('info')
logging.warning('warning')
logging.error('error')
logging.critical('critical')
logging.log(10,'log')
```

- 自定义文件句柄，用于多多个文件进行输出，如屏幕、文件1、等写入

```
import logging
# 定义文件
file_1_1 = logging.FileHandler('l1_1.log', 'a', encoding='utf-8')
fmt = logging.Formatter(fmt="%(asctime)s - %(name)s - %(levelname)s -%
(module)s:  %(message)s")
file_1_1.setFormatter(fmt)  #该文件句柄1绑定格式

file_1_2 = logging.FileHandler('l1_2.log', 'a', encoding='utf-8')
fmt = logging.Formatter()
file_1_2.setFormatter(fmt)  #该文件句柄2绑定格式

# 定义日志
logger1 = logging.Logger('s1', level=logging.ERROR)
logger1.addHandler(file_1_1)
logger1.addHandler(file_1_2)
```

```
# 写日志
logger1.critical('1111')
```

结合项目的使用，在log.py文件中的对象在导入模块时会是单例模式



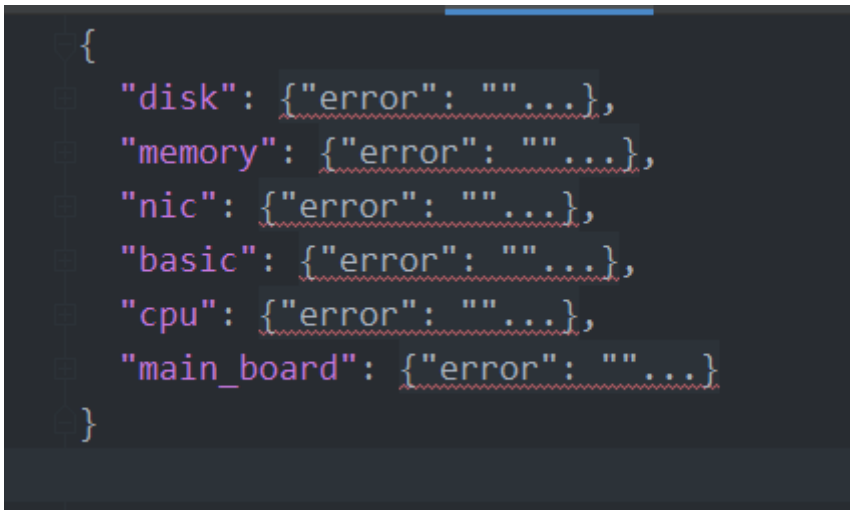
- 在utils/log.py代码如下：

```
import logging
from libs.conf import settings

class Logger:
    # 最好单例
    def __init__(self,name,log_file,level=logging.DEBUG):
        file_handler = logging.FileHandler(log_file, 'a', encoding='utf-8')
        fmt = logging.Formatter(fmt="%(asctime)s - %(name)s - %(levelname)s
-%(module)s:  %(message)s")
        file_handler.setFormatter(fmt)

        self.logger = logging.Logger(name, level=level)
        self.logger.addHandler(file_handler)

    def info(self,msg):
```

```
            self.logger.info(msg)

        def debug(self,msg):
            self.logger.debug(msg)

        def error(self, msg):
            self.logger.error(msg)

    logger = Logger(settings.LOG_NAME,settings.LOG_FILE_PATH)
```

结合log.py在disk中使用

```
...
from libs.utils.log import logger

class Disk(BasePlugin):

    def linux(self, handler, hostname=None):
        # 将数据封装成一个对象中，调用__dict__返回
        response = BaseResponse()
        try:
            ...
        except Exception:
            error = traceback.format_exc()
            response.status = False
            response.error  = error
            logger.debug(error)

        return response.dict
```



唯一标识问题

当设计到数据库有数据后，我们要去修改更新数据，那么就存在唯一标识问题

- 物理机
  - sn号（*SN*:Serial Number,产品序列号）
- 物理机 + 虚拟机
  - 方案一：

```
1、物理机sn号
2、虚拟机  通过虚拟技术的接口
```

- 方案二：

```
主机名

大致流程：
    1、在cert文件中保存上主机名
    新的机器，如果没有文件 --> 告诉API  新增的操作 ,响应正常后，保存主机名到cert文件中

    2、老的机器
    读取文件 拿到老的主机名，使用老主机名和新采集的主机名对比：
        1、相等。没有修改主机名，告知api 更新资产信息
        2、不相等，修改主机名，告知API 更新资产信息 + 更新主机名
```

总的来说，就是在客户端添加一个认证证书，用于记录当前运行的主机名，相当于唯一标识，那分为两种情况：

- 新机器，没有认证证书，将会又API回复，而自动写入
- 老机器，有且未发生改变，那么资源要更新数据即可；如果不等，则修改主机名+资产信息
- 代码如下：angent.py

```
class AgentHandler(BaseHandler):

    def cmd(self,command,hostname=None):
        import subprocess
        ret = subprocess.getoutput(command)
        return ret

    def handler(self):
        """
        收集硬件信息 汇报给API
        :return:
        """
        info = get_sever_info(self)

        # 客户端没有认证文件，表示新的主机
        if not os.path.exists(settings.CERT_PATH):
            info['action'] = 'create'  # 携带action
        else:
            # 老的主机名，认证文件已经存在
            with open(settings.CERT_PATH,'r',encoding='utf-8') as f:
                old_hostname = f.read()
            hostname = info['basic']['data']['hostname']
            if hostname == old_hostname:
                # 相当于主机名没有更改，告诉API只更新资产信息
                info['action'] = 'update'
            else:
                # 修改主机名，告知API更新资产信息 + 主机名
                info['action'] = 'update_host'
                info['old_hostname'] = old_hostname

        #发送数据给API，但是这是同步的，显得特别慢，因为有多个主机
```

```
        ret = requests.post(
            url=self.asset_url,
            data=json.dumps(info).encode('utf-8'), # 需要编码，否则过去解码不行
            headers = {'content-type': 'application/json'}  # 没有会抛415
        )
        res = ret.json()

        if res.get('status'):
            with open(settings.CERT_PATH,'w',encoding='utf-8') as f:
                f.write(res['hostname'])
```

当basic.py文件中的，hostname发生改变后，响应的证书也会变化，标识的唯一性满足

```
class Basic(BasePlugin):
    ...
    def linux(self, handler, hostname=None):
        response = BaseResponse()
        try:
            if self.debug:
                ret = {
                    'os_platform': 'linux',
                    'os_version': '6.5',
                    'hostname': 'c1.com'  # 发生改变
                }
        ....
```

响应的后端服务器提供的API，视图逻辑

```
class Asset(APIView):

    def get(self,request):
        host_list = ['192.168.109.128','192.168.109.129']*10
        return Response(host_list)

    def post(self,request):
        print(request.data)
        info = request.data
        action = info.get('action')
        hostname = info['basic']['data']['hostname']

        result = {
            'status':True,
            'hostname':hostname
        }

        if action == 'create':
            print('新增资产')
        elif action == 'update':
            print('只更新资产信息')

        elif action == 'update_host':
            print('修改资产信息+ 主机名')

        return Response(result)
```

后端接口操作数据库

主要时根据 提交的数据，对数据库已有的数据进行增删改查，这里使用到了集合的概念：重点！

- views.py文件代码如下

```
from rest_framework.views import APIView
from rest_framework.response import Response
from repository import models
from .service import process_basic, process_disk, process_memory,
process_nic

class Asset(APIView):

    def get(self,request):
        host_list = ['192.168.109.128','192.168.109.129']*10
        return Response(host_list)

    def post(self,request):
        info = request.data
        action = info.get('action')
        hostname = info['basic']['data']['hostname']

        result = {
            'status':True,
            'hostname':hostname
        }

        if action == 'create':
            # 新增资产信息，新增server、disk、memory
            server_info = {}
            basic = info['basic']['data']
            main_board = info['main_board']['data']
            cpu = info['cpu']['data']
            server_info.update(basic)
            server_info.update(main_board)
            server_info.update(cpu)
            server = models.Server.objects.create(**server_info)

            # 新增disk
            disk_info = info['disk']['data']
            disk_obj_list = []
            for disk in disk_info.values():
                # 通过字段赋值的方式创建
                disk_obj_list.append(models.Disk(**disk, server=server))

            if disk_obj_list:
                models.Disk.objects.bulk_create(disk_obj_list) # 批量插入

            # 新增memory
            memory_info = info['memory']['data']
            memory_obj_list = []
            for memory in memory_info.values():
                memory_obj_list.append(models.Memory(**memory,
server=server))
            if memory_obj_list:
                models.Memory.objects.bulk_create(memory_obj_list)
```

```
            # 新增nic
            nic_info = info['nic']['data']
            nic_obj_list = []
            for name, nic in nic_info.items():
                nic_obj_list.append(models.NIC(**nic, name=name,
server=server))
            if nic_obj_list:
                models.NIC.objects.bulk_create(nic_obj_list)

        elif action == 'update' or action=='update_host':
            # 只更新资产信息
            # 更新主机表
            server = process_basic(info)
            process_disk(info, server)
            process_memory(info, server)
            process_nic(info, server)

        return Response(result)
```

响应的service.py文件，主要解决对数据的情况进行判断，是增/删/改

```
from repository import models

def process_basic(info):
    server_info = {}

    basic = info['basic']['data']
    main_board = info['main_board']['data']
    cpu = info['cpu']['data']
    server_info.update(basic)
    server_info.update(main_board)
    server_info.update(cpu)

    hostname = info['basic']['data']['hostname']  # 新的hostname
    old_hostname = info.get('old_hostname')  # 老的hostname

    server_list = models.Server.objects.filter(hostname=old_hostname if
old_hostname else hostname)
    server_list.update(**server_info)
    server = models.Server.objects.filter(hostname=hostname).first()
    return server


def process_disk(info, server):
    disk_info = info['disk']['data']  # 新提交的数据

    disk_slot_set = set(disk_info)
    disk_slot__db_set = {i.slot for i in
models.Disk.objects.filter(server=server)}

    # 新增  删除  更新
    add_slot_set = disk_slot_set - disk_slot__db_set  # 新增的槽位
    del_slot_set = disk_slot__db_set - disk_slot_set  # 删除的槽位
    update_slot_set = disk_slot__db_set & disk_slot_set  # 更新的槽位

    # 新增硬盘
    add_disk_lit = []
```

```
    for slot in add_slot_set:
        disk = disk_info.get(slot)
        add_disk_lit.append(models.Disk(**disk, server=server))

    if add_disk_lit:
        models.Disk.objects.bulk_create(add_disk_lit)

    # 删除硬盘
    if del_slot_set:
        models.Disk.objects.filter(server=server,
slot__in=del_slot_set).delete()

    # 更新硬盘
    for slot in update_slot_set:
        disk = disk_info.get(slot)
        models.Disk.objects.filter(server=server, slot=slot).update(**disk)

def process_memory(info, server):
    # 更新内存
    memory_info = info['memory']['data']  # 新提交的数据

    memory_slot_set = set(memory_info)
    memory_slot__db_set = {i.slot for i in
models.Memory.objects.filter(server=server)}

    # 新增  删除  更新
    add_slot_set = memory_slot_set - memory_slot__db_set  # 新增的槽位
    del_slot_set = memory_slot__db_set - memory_slot_set  # 删除的槽位
    update_slot_set = memory_slot__db_set & memory_slot_set  # 更新的槽位

    # 新增内存
    add_memory_lit = []
    for slot in add_slot_set:
        memory = memory_info.get(slot)
        add_memory_lit.append(models.Memory(**memory, server=server))

    if add_memory_lit:
        models.Memory.objects.bulk_create(add_memory_lit)

    # 删除内存
    if del_slot_set:
        models.Memory.objects.filter(server=server,
slot__in=del_slot_set).delete()

    # 更新内存
    for slot in update_slot_set:
        memory = memory_info.get(slot)
        models.Memory.objects.filter(server=server,
slot=slot).update(**memory)


def process_nic(info, server):
    nic_info = info['nic']['data']  # 新提交的数据

    nic_name_set = set(nic_info)
    nic_name__db_set = {i.name for i in
models.NIC.objects.filter(server=server)}
```

```
    # 新增   删除   更新
    add_name_set = nic_name_set - nic_name__db_set  # 新增的槽位
    del_name_set = nic_name__db_set - nic_name_set  # 删除的槽位
    update_name_set = nic_name__db_set & nic_name_set  # 更新的槽位

    # 新增网卡
    add_nic_lit = []
    for name in add_name_set:
        nic = nic_info.get(name)
        nic['name'] = name
        add_nic_lit.append(models.NIC(**nic, server=server))

    if add_nic_lit:
        models.NIC.objects.bulk_create(add_nic_lit)

    # 删除网卡
    if del_name_set:
        models.NIC.objects.filter(server=server,
name__in=del_name_set).delete()

    # 更新网卡
    for name in update_name_set:
        nic = nic_info.get(name)
        nic['name'] = name
        models.NIC.objects.filter(server=server, name=name).update(**nic)
```

后端API验证

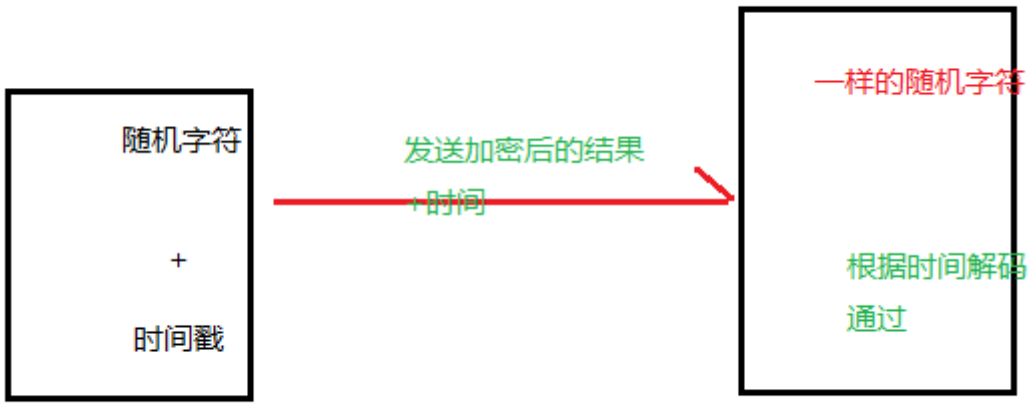
- 方式一：

  后端和客户端同时携带同一个随机字符串

  缺点：明文不安全、不动态

- 方式二：

  ```
  key + time + md5加密
  ```



后端根据数据库数据完成增删改查