一、算法开发说明文档

1. 开发服务器用户环境搭建

① 连接开发服务器： nas.fast-fun.cn:65389
② 在 D:\Develop\User 路径下建立个人的文件夹，例如: myself
③ 进入 D:\Develop\User\myself 路径，拉取 git 代码（需要提前获取 git 权限）。

git clone http://git.fast-fun.cn:92/lmstack/data_analyze_platform.git

④ 配置用户环境变量

PYTHONPATH = D:\Develop\User\myself\data_analyze_platform

2. 算法开发规则

① 算法文件结构建立。LIB/MIDDLE 下必须包含 算法类别/算法名/版本号/CoreAlgo/ 和算法类别/算法名/版本号/Main/ 的路径，路径不要包含中文和空格。

- LIB
  - MIDDLE
    - 算法类别（AlgoType)
      - 算法名 (AlgoName1)
        - 版本号 (V1_0_0)
          - CoreAlgo
            - core.py (核心算法文件，名字任意)
          - Main
            - config-dev.ini (开发环境配置文件，名字不要修改)
            - config-pro.ini (生产环境配置文件，名字不要修改)
            - config-test.ini (测试环境配置文件，名字不要修改)
            - Dockerfile(镜像打包所需要的文件，名字不要修改)
            - main.py (该算法运行的主函数，名字不要修改)

以下的示例文件，可从 data_analyze_platform 里面找到

② core.py 文件示例

a. 使用 LIB 的公共库文件时，必须采用 from LIB.BACKEND 的格式从 LIB 开始书写
b. 使用 CoreAlgo 下的其他文件时，也必须从 LIB 开始书写
c. 算法仅接收需要的原始数据和参数，并返回结果。
输入输出建议都采用 dataframe 数据类型

```python
from LIB.BACKEND import DBManager # 以相对路径的方式从 LIB 开始引入
import pandas as pd
class CalSor:
    def __init__(self):
```

```
            pass
        def calSor(self, sn,df_bms, celltype):
            i = len(df_bms)
            return pd.DataFrame({'data':[i],'sn':[sn]})
```

③ main.py 文件示例

a. 使用 LIB 的公共库文件时，必须采用 from LIB.BACKEND 的格式从 LIB 开始

b. 调用核心算法文件时，也必须从 LIB 开始

c. main.py 不包含算法相关的任何逻辑。包含如下内容：

- ✧ 读取环境变量，判断当前是运行在开发、测试、还是正式环境
- ✧ 根据环境变量，读取对应的配置文件，从配置文件中获取需要的配置参数
- ✧ 配置日志生成的方式及路径
- ✧ 获取资产 sn 总表及相关的信息，如电池类型、厂家等
- ✧ 遍历 sn，获取对应 sn 的历史数据，并调用核心算法
- ✧ 将算法的结果入库
- ✧ 异常处理
- ✧ 数据库的连接及资源释放等

```
# -*- coding: UTF-8 -*-

import pandas as pd
import time
from sqlalchemy import create_engine
import os
import configparser
import pymysql
import traceback
import datetime
from LIB.BACKEND import DBManager,Log # 以相对路径的方式从 LIB 开始引入！！！！！
from LIB.MIDDLE.AlgoDemo.Demo.V_1_0_0.CoreAlgo import core # 以相对路径的方式从 LIB 开始引入！！！!!
from LIB.BACKEND.OPENAPI import OpenApi
```

```
if __name__ == '__main__':



    # 环境变量配置（通过环境变量确定当前程序运行在开发、测试、生产环境）
    env_dist = os.environ
    cur_env = env_dist.get("CURENV", 'dev') # 默认为开发环境

    # 读取配置文件
    cf = configparser.ConfigParser()
    if cur_env == 'dev':
        cf.read("config-dev.ini")
```

```
    elif cur_env == 'test':
        cf.read("config-test.ini")
    elif cur_env == 'pro':
        cf.read("config-pro.ini")

    host1 = cf.get("Mysql-1", 'host')
    port1 = int(cf.get("Mysql-1", 'port'))
    db1 = cf.get("Mysql-1", 'db')
    user1 = cf.get("Mysql-1", 'user')
    password1 = cf.get("Mysql-1", 'password')

    host2 = cf.get("Mysql-2", 'host')
    port2 = int(cf.get("Mysql-2", 'port'))
    db2 = cf.get("Mysql-2", 'db')
    user2 = cf.get("Mysql-2", 'user')
    password2 = cf.get("Mysql-2", 'password')

    # 日志配置（按照该配置，每次运行时可自动生成运行日期的文件夹，会在与 main 同级的）
    now_str = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()).replace(":","_")
    log_path = 'log/' + now_str
    if not os.path.exists(log_path):
        os.makedirs(log_path)
    log = Log.Mylog(log_name='cal_sor', log_level = 'info')
    log.set_file_hl(file_name='{}/info.log'.format(log_path), log_level='info', size=1024* 1024
* 100)
    log.set_file_hl(file_name='{}/error.log'.format(log_path), log_level='error', size=1024*
1024 * 100)
    log.set_stream_hl("error") # 打印错误日志
    logger = log.get_logger()
    logger.info("{}, 算法开始".format(str(os.getpid())))

    try:
        # 连接数据库的两种方式
        # 方式一：新 dataframe 写入数据库时，采用该方式可以不需要写 sql 语句；
        #     该方式无法对数据库进行修改；

        db_engine_1 = create_engine(
            "mysql+pymysql://{}:{}@{}:{}/{}?charset=utf8".format(
                user1, password1, host1, port1, db1
            ))

        # 方式二：该方式可以通过写 update SQL 语句，对数据库中的数据进行修改
        conn = pymysql.connect(host=host2, port=port2, user=user2, password=password2,
database=db2)
```

```
        cursor = conn.cursor()
```

```
        # 准备算法数据

        # 原始数据时间设置
        end_time=datetime.datetime.now()
        start_time=end_time-datetime.timedelta(seconds=300)
        start_time=start_time.strftime('%Y-%m-%d %H:%M:%S')
        end_time=end_time.strftime('%Y-%m-%d %H:%M:%S')

        # 从开放平台获取资产列表及相关信息
        openApi = OpenApi.OpenApi()
        df_all_sn = openApi.get_asset()
        calSor = core.CalSor(); # 算法初始化

        # 遍历资产列表，获取数据和参数，输入算法中
        for index in range(0, df_all_sn.index[-1]):

            try:# 解析得到厂家、sn、协议类型、电芯参数等输入数据
                factory = df_all_sn.loc[index,'factory']
                sn = df_all_sn.loc[index,'sn']
                imei = df_all_sn.loc[index,'imei']
                protocolType = df_all_sn.loc[index,'protocolType'] # 协议类型（3：32960，10：科易，
13：优旦）
                if imei[5:9] == 'N640':
                    celltype=1 #6040 三元电芯
                elif imei[5:9] == 'N440':
                    celltype=2 #4840 三元电芯
                elif imei[5:9] == 'L660':
                    celltype=99 # 6060 锂电芯
                elif imei[3:5] == 'LX' and imei[5:9] == 'N750':
                    celltype=3 #力信 50ah 三元电芯
                elif imei[3:5] == 'CL' and imei[5:9] == 'N750':
                    celltype=4 #CATL 50ah 三元电芯
                elif imei[1:3] == 'JM': # 金茂
                    celltype=100 #CATL 50ah 三元电芯
```

```
                # 获取电池历史数据
                logger.info("{} start".format(sn))
                dbManager = DBManager.DBManager()
                df_data = dbManager.get_data(sn=sn, start_time=start_time, end_time=end_time,
data_groups=['bms', 'gps', 'accum', 'system'])
                df_bms = df_data['bms']
```

```
                # 获取数据库原始数据
                df_ram = pd.read_sql("select * from test_tb where sn ='{}'".format(sn), db_engine_1)

                # 调用核心算法
                df_res = calSor.calSor(sn, df_bms,celltype);

                # 算法结果入库
                # 新增
                if (len(df_ram) == 0):
                    df_res.to_sql("test_tb",con=db_engine_1, if_exists="append",index=False)

                else:
                # 修改
                    sql = '''update test_tb set data={} where sn='{}'
'''.format(df_res['data'].values[0], (df_res['sn'].values[0]))
                    cursor.execute(sql)
                    conn.commit()
                logger.info("{} done".format(sn))
            except Exception as e :
                logger.error(traceback.format_exc)
                logger.error(str(e))
                logger.error(u"{} : {},{} 任务运行错误\n".format(sn,start_time,end_time),
exc_info=True)

    except Exception as e :
        logger.error(traceback.format_exc)
        logger.error(str(e))
        logger.error(u"任务运行错误 2\n", exc_info=True)

    # 释放数据库资源
    cursor.close()
    conn.close()
    db_engine_1.dispose()
```

④ config-dev.ini 示例

a. 该文件配置了两个 mysql 的连接参数。[]内部为参数组名，在 main.py 有读取该文件的示例

```
[Mysql-1]
host = localhost
port = 3306
db = test
user = root
password = Qx123456
```

```
[Mysql-2]
host = localhost
port = 3306
db = test
user = root
password = Qx123456
```

3. 算法打包并在测试环境发布

① 编写 Dockerfile 文件

a. 拷贝 AlgoDemo/Demo/V_1_0_0/Main 下面的 Dockerfile 到各自算法的 Main 文件夹下面

b. 修改以下几个参数

1) author

2) 用到的 python 第三方库和版本

3) ALGO_PATH 算法的路径，从 LIB 开始书写

```
# 基础镜像
FROM python:3.8

# 作者 （需要修改)
LABEL author="lm"
```

```
# 环境变量参数
ENV TZ="Asia/Shanghai"
ENV PYTHONPATH="/"
```

```
# 安装用到的 python 第三方库（需要修改)
RUN pip install -i https://pypi.tuna.tsinghua.edu.cn/simple \
    sqlalchemy \
    pandas==1.3.4 \
    pymysql apscheduler \
    scipy \
    cryptography \
    numpy==1.20.3 \
    requests
```

```
#RUN rm -r /usr/bin
#RUN rm -r /bin
```

```
# 配置算法路径（需要修改)
ARG ALGO_PATH=LIB/MIDDLE/AlgoTest/Algo1/V_1_0_0/
```

```
# 复制文件文件到 容器
ADD LIB/BACKEND/ /LIB/BACKEND
ADD ${ALGO_PATH}/ /${ALGO_PATH}/
WORKDIR ${ALGO_PATH}/Main/
```

```
CMD ["python", "main.py"]
```

② 上传代码并 push 到远程服务器。

③ 利用 xshell 登录 测试环境的 linux 服务器

ip: 192.168.0.43

User: qx

Password: Qx123456

④ 在终端执行如下命令

algo_build.sh 算法类型名 算法名 版本

```
qx@qx-db:~/algo_test/project$ algo_build.sh AlgoDemo Demo V_1_0_0
```

等待出现如下界面，则表示镜像打包成功，并已经上传到了仓库。

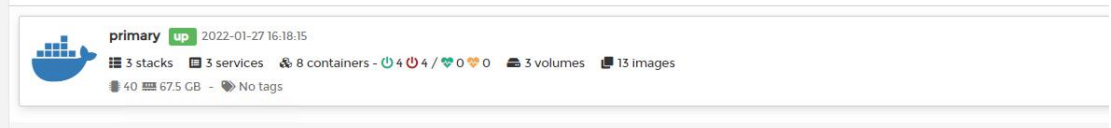
并复制下红圈中[]内部的内容。

```
Successfully built fde7300ea7cc
Successfully tagged algodemo_demo:V_1_0_0
The push refers to repository [nas.fast-fun.cn:5000/zhl/algo/algodemo_demo]
acab931e2769: Pushed
08e16148ec00: Pushed
ee94822c5c5a: Mounted from zhl/algo/algotest_algo1
d09836256ed7: Mounted from zhl/algo/algotest_algo1
984d2bf694d9: Mounted from zhl/algo/algotest_algo1
5e2d6061e5d9: Mounted from zhl/algo/algotest_algo1
aedcb370b058: Mounted from zhl/algo/algotest_algo1
c3a0d593ed24: Mounted from zhl/algo/algotest_algo1
26a504e63be4: Mounted from zhl/algo/algotest_algo1
8bf42db0de72: Mounted from zhl/algo/algotest_algo1
31892cc314cb: Mounted from zhl/algo/algotest_algo1
11936051f93b: Mounted from zhl/algo/algotest_algo1
V_1_0_0: digest: sha256:2a397f0770169b8a02f12281b8f57b7638ff5df7d1ecbb2635fe2e083f7708f5 size: 2849
```

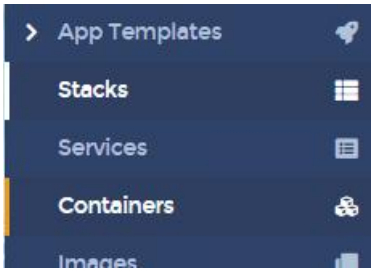
⑤ 开发服务器上用浏览器打开 https://192.168.0.43:9000,

⑥ 用账号 qx 密码 Qx123456 登录

⑦ 点击 primary

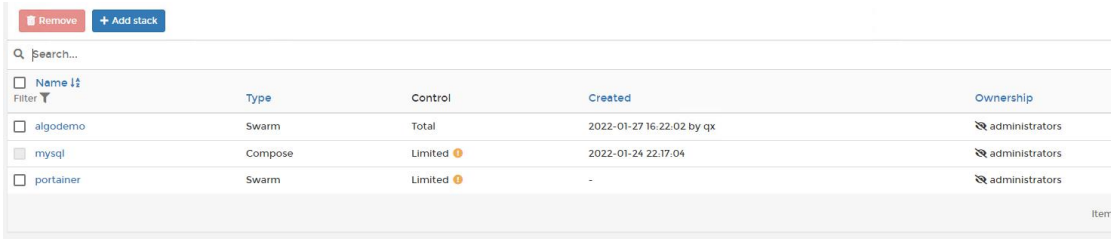


⑧ 点击左侧的 Stacks



⑨ 如果要部署的算法是一个新的算法类型，则点击 Add stack； 否则直接在列表中点

击对应的算法类型



⑩ Add stack 后，Name 填写算法的类型名；editor 里填入如下内容，{}内按照实际情况替换

```
version: "3.7"
services:
  {算法名}:
    image: {第 4 步中，复制下来的红色方框中的内容}
    restart: always
    environment:
      - CURENV=test
    volumes:
      - {算法名}:{log 的生成位置。例： /LIB/MIDDLE/AlgoDemo/Demo/V_1_0_0/Main/log}
volumes:
  {算法名}:
```

⑪ 点击最下方的 Deploy the stack 即可在测试环境中运行起该镜像