决赛附件-赛题1决赛开发环境-TPT平台自定义算法编写规范及运行说明

1. 概述

1.1.编写说明

TPT平台内置多种常用的数据分析和机器学习算法，满足通用性需求，同时也支持自定义算法的上传运行，满足不同场景下的个性化使用需求。TPT平台支持的自定义算法为Python语言编写，本文档是针对使用Python语言编写的py文件，在平台上的使用说明。

1.2.术语说明

文档中涉及的术语说明如下表所示：

序号	名称	说明
1	算法文件	使用Python语言编写的算法逻辑格式文件,后缀为.py。
2	租户id（tenent_id）	TPT产品为支持多租户设计，每一次算法的调用都必须知道是哪个租户调用，该租户的唯一标识就是租户id。
3	日志id（log_id）	从发起调用算法，再导算法的运行，每一次调用都会通过一个唯一id完成全链路的调用监控，即log_id，通过log_id可以查询本次算法执行过程的详细日志信息。
4	算法返回结果码code	算法返回结构中，对算法运行结果表征的特征码，运行成功必须为"000",运行异常或错误自定义算法可以通过10000以上的数字字符串分别自己定义表示，例如：10001、10002等。

自定义算法编写完成后，在TPT平台上需通过任务管理服务、算法管理服务、算法运行时服务和Ray分布式计算框架等管理和运行，若涉及对在线数据的读取和算法运行结果的回写保存，还需要数据中心服务支持。各服务的说明如下表所示：

序号	服务名称	说明
1	任务管理schedule-manager	在任务管理配置需要定时执行的算法任务，通过任务的单次执行或间隔的频率执行，运行自定义算法。
2	算法管理	对自定义算法上传和编辑，负责维护算法本身的一些

	algorithm-manager	元数据，比如算法入参、出参，作者等，对发布算法还需要进行发布操作，方可在平台调用运行。
3	算法运行时 runtime-python	作为 ray 的一个客户端，负责和 ray 交互，负责快速将算法文件发布成服务，或者提交到 ray 上触发算法训练。属于一个前期算法文件规范的校验层和中转层。（注：算法开发人员无需关注）
4	分布式计算框架 ray	一个 Python 代码运行的分布式计算框架，上述所说的算法发布和算法训练本质都是在 ray 上运行。 （注：算法开发人员无需关注）
5	数据管理 data-hub	TPT 平台实时数据的采集和存储服务，需要配置完成连接的数据源、数据源中需要用到的位号配置等。自定义算法可通过内部 API 读取配置的位号实时值、历史值，也可以将算法结果回写到位号中（一次位号需配置为可回写或者使用虚位号）。

1.3. 运行环境要求

TPT 平台的 Python 运行环境的版本为 Python 3.10.16，环境中的常用各依赖包及版本如下表所示：

序号	依赖包名	版本
1	pandas	1.5.3
2	numpy	1.24.4
3	matplotlib	3.10.0
4	matplotlib-inline	0.1.7
5	tune-sklearn	0.4.6
6	pytorch-forecasting	1.0.0
7	pytorch-lightning	2.5.0.post0
8	pytorch-metric-learning	2.9.0
9	pytorch_optimizer	2.12.0
10	reformer-pytorch	1.4.4
11	segmentation_models_pytorch	0.5.0

12	torch	2.1.0
13	torchdata	0.11.0
14	torchmetrics	1.6.1
15	torchsummary	1.5.1
16	torchvision	0.16.0
17	joblib	1.4.2
18	scikit-base	0.8.3
19	scikit-fuzzy	0.5.0
20	scikit-image	0.24.0
21	scikit-learn	1.5.2
22	scikit-opt	0.6.6
23	scikit-optimize	0.9.0
24	scipy	1.15.3

自定义算法使用的依赖包相同的，需要保持与运行环境中的版本一致，同等功能的依赖包优先使用环境中已有的。

注：如需完整依赖包清单，可在中控杯决赛群联系组委会肖老师，或发邮件至 xiaojun@supcon.com 咨询。

2. 算法编写规范说明

2.1.总体规范

2.1.1. 算法文件命名规范

py 文件名确保不要出现 state.py，os.py 等常用的名称，此类名称容易在 Python 解释器里面产生重复，导致导入 py 文件时产生冲突，无法加载到解释器里。py 文件的名字尽量带点业务含义，减少冲突的发生。

2.1.2. 算法内主体规范

所有的 py 文件必须有 def main() 方法，算法管理上传会识别到 main 方法，入参需加上参数数据类型否则无法识别。如果多个 py 文件之间有依赖，需要把这些 py 文件打包成 zip 包的格式。zip 包名字和包含 main 方法的 py 文件名保持一致。

为规范算法在运行过程中的链路监控，全链路都需要有租户 id 和日志 id，算法文件编写时，需要按照特定写法，获取到租户 id 和日志 id。

2.1.3. 算法返回结构规范

算法的返回结构中必须按如下结构返回，包含 code、message、log_id、content 四个 key。

```
output = {
  "code":"000",  #算法成功还是失败的 code 码
  "message": "算法成功执行", #算法本身执行成功与否的详细说明，如位号未找到。
  "log_id": log_id, #记录日志的全局唯一标识
  "content": alg_result  #算法执行的最终结果
}
return output
```

其中："code"返回算法执行状态的标识，执行成功为"000"。其它状态外部自定义算法参考 10000 以上的五位数字字符串，具体可自己定义如"10001"、"10020"等，此类通常在捕获异常时返回。

"message"定义算法返回的信息提示，通常 code 为"000"时为"执行成功"，其它为配合 code 对应的异常信息。

"log_id"为使用公共模块工具获取的日志 id 信息。

"content"为算法执行成功的最终结果，若其它状态可为""。

2.2. 公共模块使用规范

公共模块为算法运行环境中提供的一些工具和方法，为算法调用中间件、算法外部参数获取及传递提供通用的方法，以便算法适应不同的环境配置可正常运行。

2.2.1. 算法内部获取租户 id

注意事项：方法调用必须放在 main 函数内部调用。

- 算法内部获取租户 id

正确用法：

```
from cache import get_tenant_id
def main(XXXXXX):
    # 获取租户信息
    tenant_id = get_tenant_id()
    print(f"当前处理的租户：{tenant_id}")
```

错误用法：

```
from cache import get_tenant_id
# 获取租户信息
tenant_id = get_tenant_id()
def main(XXXXXX):
    print(f"当前处理的租户：{tenant_id}")
```

2.2.2. 日志使用方法

注：平台默认算法打印日志级别是 ERROR；alg_name 的参数名需要和包含 main 方法的文件名一致。

```
#使用示例
from custom_log import CustomLog
#CustomLog 接受 1 个参数：alg_name 算法名
#默认只打印 100*1024 字节的消息，多余的会截断
logger = CustomLog(alg_name="nsj_test.py")
def main(XXXXXX):
    logger.info(code='111',message="info")
    logger.error(code='777',message="error
/root/PycharmProjects/runtime-")
    logger.warn(code='333',message="warn
/root/PycharmProjects/runtime-")
    logger.debug(code='444',message="debug
/root/PycharmProjects/runtime-")
```

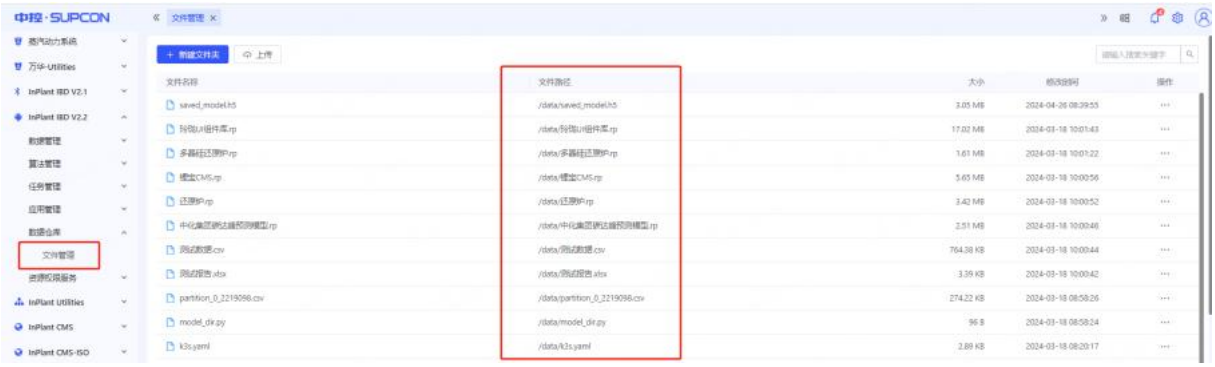
2.2.3. 文件操作使用规范

TPT 平台采用 MinIO 组件时间对模型文件、数据文件等各类文件数据的存档，MinIO 使用的是 s3 协议。

注：s3 是一种通用协议，MinIO 兼容此协议。s3fs 库是 s3 协议的一种实现，但是使用 s3fs api 操作 MinIO 时，有很小的概率会出现数据下载不下来的情况。

fs = s3fs.S3FileSystem() fs.open(f"/data/xxx.pkl", 'wb')中，fs.open 会从 MinIO 读取文件，每次执行都会重新从 MinIO 读取，文件较小时，效率影响不大。如果文件很大，并且需要重复读取的，建议先保存到服务本地目录，再从本地进行读取操作。

上传及查看文件的方式如下图所示：



使用工具获取文件的方法如下：

```
#数据仓库写
import s3fs
fs = s3fs.S3FileSystem()
with fs.open(f"/data/xxx.pkl", 'wb') as s:
       dill.dump(model, s)
```

2.2.4. 缓存工具使用规范

该缓存工具主要有以下功能：

(1)获取对应环境各个中间件及业务所需的配置，避免算法写死，导致在不同环境修改算法；

(2)从 MinIO 下载文件到服务器本地，然后算法本身去读取该文件或加载该文件；

(3)将算法本身生成的文件保存到 MinIO，做到持久化存储。（服务器本地目录并没有持久化，如果不上传到 MinIO，会丢失。）；

(4)算法内部获取租户 id 信息。

- 参数介绍

传入三个参数，依次是 redis_host:redis 所在服务器名，需要传入，但不做校验；redis_port:redis 服务的端口号，需要传入，但不做校验；model:在服务器上运行必须为"run"

备注：本地默认路径/root/app/model，MinIO 的默认路径是 alg/model/

- 在算法内部导入该模块

```
import cache
ca = cache.CacheTools(redis_host='localhost', redis_port='6379',
model="run")
```

初始化完成后，不同场景的使用示例：

```
#获取对应集群的中间件地址，所有获取的信息都是字符串，有些参数需要是整形，使用者自己转一下
#获取初始化后的 redis 客户端
#获取 cache 工具内部初始化的 redis 客户端
redis_client = ca.redis_client

#获取 MinIO 相关信息，可以使用 s3 协议自定义操作 MinIO 的文件
MinIO_key = ca.configs.get("biz", "MinIO_key")
MinIO_secret = ca.configs.get("biz", "MinIO_secret")
MinIO_endpoint_url = ca.configs.get("biz", "MinIO_endpoint_url")

#获取 timescaladb 相关信息
timescaladb_user = ca.configs.get("biz", "timescaladb_user")
timescaladb_password= ca.configs.get("biz",
"timescaladb_password")
timescaladb_host=ca.configs.get("biz", "timescaladb_host")
timescaladb_port=ca.configs.get("biz", "timescaladb_port")
#密码一般都包含特殊字符，需要编码一下
from urllib.parse import urlencode, urlunparse,quote
encoded_password = quote(timescaladb_password)
```

2.2.5. MinIO 相关操作说明

MinIO 上的默认路径是 alg/model/，本地默认路径/root/app/model/ ，type 默认是 dir。

算法编写用到的典型场景为从文件系统将大的模型文件下载到本地进行运行，减少每次从文件系统读大的文件的时间。

```
import cache
ca = cache.CacheTools(redis_host='localhost',  redis_port='6379',
  model="run")

def main():
    # 传入 bb.txt 文件名，同步 minio 上的 data/aa/bb.txt 文件到本地路径,返回本地路径/tmp/model/aa/bb.txt
    #下载模型文件到本地
   ca.sync_object("bb.txt", type="file",bucket_name="data",base_local_folder="/tmp/model/",middle_name="",sub_dir="aa")
```

2.3. 算法 zip 包编写规范

如果多个 py 文件之间有依赖，需要把这些 py 文件打包成 zip 包的格式。zi 包名字和包含 main 方法的 py 文件名保持一致。

如含有 main()方法的 data_eval.py 文件依赖于 read_minio.py 这个文件，则把这两个文件打成 zip 包，打的 zip 包重命名为 data_eval.zip。zip 包打开的第一个层级就是这两个 py 文件。

py 文件导入注意事项

若在 main()方法所在的 py 文件中需要导入自定义模块或 py 文件,需要加入以下两行代码，注意将其添加在使用 import 导入自定义的模块之前。

```
sys.path.append("/root/app/resource/")
sys.path.append(os.path.dirname(os.path.abspath(__file__)))
```

示例：

```
import os
import sys
sys.path.append("/root/app/resource/")
sys.path.append(os.path.dirname(os.path.abspath(__file__)))

#以下是自定义 py 文件导入
import test1
from preprocess import Preprocess
```

3. 完整示例演示

若用户已有训练好的模型文件，TPT 平台支持用户已有训练好的模型上传平台使用，请参照 2.2.3、2.2.5 两个章节说明，将模型文件上传到平台，并在算法内部对模型文件读取。

本示例将演示简单的推理算法编写实例，此类通常执行时间较短，上传平台后需要发布方可运行。

3.1. 应用场景

分别计算两个位号过去一分钟平均值，再与第三个位号的实时值相加，并将结果回写到第四个位号保存,四个位号分别为 1_TT_21011.PV、1_TT_21402.PV、1_TT_21008.PV、test_Tag_Write1。

在 TPT 的系统管理->数据管理->位号管理中确保上述 4 个位号存在，并且前面数据采集正常，位号 test_Tag_Write1 可写。

3.2. 算法示例

算法文件命名为 test_customize_alg.py，示例代码如下：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import os
import sys
import requests
import json
import pandas as pd
from pandas import DataFrame as pd_DataFrame
from numpy import ndarray
import numpy as np
```

```
    from datetime import datetime, timedelta

    # 导入 cache 模块（请确保 cache.py 存在且包含 CacheTools 类）
    import cache
    from cache import get_tenant_id,get_log_id
    from custom_log import CustomLog
    #CustomLog 接受 1 个参数：alg_name 算法名
    #默认只打印 100*1024 字节的消息，多余的会截断。发布算法是常驻内存，所以 CustomLog 初始化放在全局
    logger = CustomLog(alg_name="test_customize_alg.py")

    ca = cache.CacheTools(redis_host='localhost', redis_port='6379', redis_db=1, model="run")

    def main() -> dict:
    try:
            current_time = datetime.now()
            current_time_str =
    str(current_time.strftime('%Y-%m-%d %H:%M:%S'))
             print('+++++', current_time_str, '+++++')
    # 获取租户 id
            tenant_id = '0' if get_tenant_id() == '' else get_tenant_id()
    # 获取日志 id
            log_id = 'test_customize_alg_'+current_time_str if get_log_id() ==
'default_log_id' else get_log_id()
            tag_sum = 0
            error_code = "19999"
    #data-hub 服务在 k8s 的内部域名
            data_hub_baseurl = 'http://data-hub.obp:8080'
    #获取历史值
            request_params = {
                "data": {
                    "begTime": (current_time - pd.Timedelta(1,
                unit='minutes')).strftime('%Y-%m-%d %H:%M:%S'), #开始时间
                    "endTime": current_time_str, #结束时间
```

```
                    "option": 2, #默认向前取数补点
                    "isAsc": True, #排序
                    "tagNames": ["1_TT_21011.PV", "1_TT_21402.PV"], #位号名列表
                    "interval": 10 #时间间隔为 10 秒
                },
                "requestBase": {
                    "page": "1-500000",
                    "sort": "-appTime"
                }
            }
        response = requests.post(
            url= data_hub_baseurl + "/api/tag-value/getHistoryValueFromDB",
            data=json.dumps(request_params, ensure_ascii=False),
            headers={
    "Connection": "keep-alive",
    'Content-Type': 'application/json',
    'tenant-id': tenant_id,
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/117.0.0.0 Safari/537.36'
    })
    if response.status_code == 200:
            result = json.loads(response.content.decode())['content']
    try:
    for key, value in result.items():
    # 提取所有 tagValue
                    tag_values = []
    for item in value.get('list', []):
                        tag_value = item.get('tagValue')
    if tag_value is not None:
                            tag_value = 0 if tag_value == "" else float(tag_value)
                            tag_values.append(tag_value)
    # 计算平均值
                    avg_value = sum(tag_values) / len(tag_values) if tag_values else
0
```

```
    # 累加平均值
                    tag_sum += avg_value
    except Exception as e:
    raise ValueError(f"{e}")
    else:
            error_code = "10001"
    raise ValueError("获取历史值失败！")

    #获取实时值
        request_params_rt = {
            "data": {
                "tagNames": ["1_TT_21008.PV"], #位号名列表
            },
            "requestBase": {
                "page": "0-0",
            }
         }
        response_rt = requests.post(
            url= data_hub_baseurl + "/api/tag-value/getRTValue",
            data=json.dumps(request_params_rt, ensure_ascii=False),
            headers={
    "Connection": "keep-alive",
    'Content-Type': 'application/json',
    'tenant-id': tenant_id,
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/117.0.0.0 Safari/537.36'
    })
    if response_rt.status_code == 200:
            result = json.loads(response_rt.content.decode())['content']
    try:
                item = next((x for x in result if x['tagName'] == '1_TT_21008.PV'),
None)
    if item and 'tagValue' in item and item['tagValue'] is not None:
    if item['tagValue'] =="":
                        tag_rtvalue = 0
```

```
    else:
                        tag_rtvalue  = float(item['tagValue'])
    else:
                    tag_rtvalue = 0
                tag_sum += tag_rtvalue
    except Exception as e:
    raise ValueError(f"{e}")
    else:
            error_code = "10002"
    raise ValueError("获取实时值失败！")
    #调用回写接口将计算结果回写对应位号
        request_params_write = {
             "data": {
                 "values": {
                     "test_Tag_Write1": tag_sum
                 }
             },
             "requestBase": {
                 "page": "0-0",
             }
         }
        response_rt = requests.post(
            url= data_hub_baseurl + "/api/tag-value/writeTagValues",
            data=json.dumps(request_params_write, ensure_ascii=False),
            headers={
    "Connection": "keep-alive",
    'Content-Type': 'application/json',
    'tenant-id': tenant_id,
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/117.0.0.0 Safari/537.36'
    })
    if response_rt.status_code != 200:
            error_code = "10003"
    raise ValueError("回写计算结果失败！")
```

```
        alg_result={
"test_Tag_Write1":tag_sum
}
        output = {
            "code":"000", #算法成功还是失败的 code 码
            "message": "算法成功执行", #算法本身执行成功与否的详细说明，如位号未找
        到。
            "log_id": log_id, #记录日志的全局唯一标识
            "content": alg_result  #算法执行的最终结果
        }
return output
except Exception as e:
   output = {
        "code":error_code,
        "message": f"算法执行失败{e}",
        "log_id": log_id,
        "content": ""
    }
return output
```

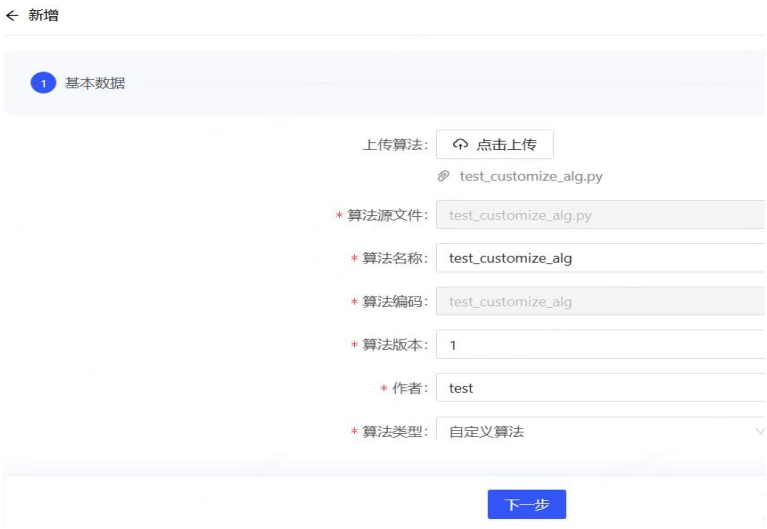
3.3. 算法上传

完成算法编写后，通过访问 https://tpt.supcon.com/tpt-admin，输入账号密码后进入 TPT 后台管理界面（若提示用户名密码不对也可先登录 TPT 平台后，再次访问），在系统管理->算法和模型管理管理->算法管理界面进行算法的上传和编辑。

3.3.1. 新增算法

进入管理界面；

点击新增按钮，跳转至基本数据填写界面，点击上传算法，上传符合算法编写规范的本地算法文件，输入算法名称、算法版本、作者，选择算法类型，选择是否被禁用，点击 下一步；



跳转至参数配置填写界面，若上传的算法符合算法编写规范，平台将自动识别入参和出参的参数列表信息，对于算法入参/出参，可以按个人需求进行参数配置的修改，若无需修改，点击确定。

创建成功之后会有反馈信息，并新增一行算法列表。

3.3.2. 算法发布

在算法管理的列表中找到需要发布的算法，在算法的操作栏中，点击发布按钮，在弹出的发布窗口中选择计算资源 CPU/GPU，输入核数，副本数，默认为 CPU，1 个核数，1 个副本数，点击确定。

注：算法发布成功后，若需要修改算法需先取消发布再进行编辑。

3.3.3. 算法取消发布

在算法管理的列表中找到需要取消发布的算法，在算法的操作栏中，点击取消发布按钮。

3.3.4. 删除算法

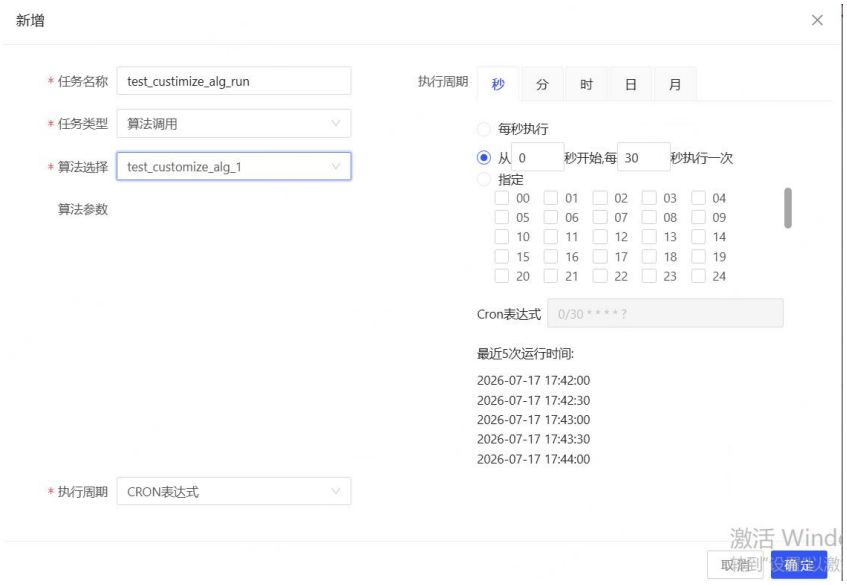
在算法管理的列表中找到需要删除的算法，在算法的操作栏中，点击删除按钮进行删除。注：算法发布成功后，若需要删除算法需先取消发布再进行删除，否则提示报错。

3.4. 算法执行

TPT 平台可以通过任务执行的方式驱动自定义算法的运行，配置完成算法执行任务

后，通过单次执行，运行一次算法，查看算法的运行情况并进行错误排查，待运行成功后，可通过启动任务进行周期性执行。

在 TPT 的后台管理系统重，点击系统管理->任务管理->任务管理界面进行任务的新建和编辑。



对应任务的单次运行和任务启动：

任务执行情况的查看：