

工具环境以及依赖报：

开发语言：Python2.7

配置文件类型：yaml

数据文件类型：yaml

Python 依赖包：yaml, Mongo, Redis, Mysql, paramiko

工具版本 1.0

开发者：文普琨

工具背景：

随着接口数量增多，接口逻辑越来越多样化，之前针对某一个服务或者某个接口，单独封装一套脚本，一套配置文件和一套参数这种场景的维护成本越来越高。

在目前的工作中我面对了以下一些主要问题：

1. 在执行回归某一个服务下的某一个接口的时候，需要提前复习之前写的脚本，数据文件，配置文件字段含义，以及代码的逻辑流程，耗费时间
2. 新接口管理总是要融合到一个已经存在的脚本中，或者新开发一套脚本中，很可能导致之前的脚本不能正常运行，同时要准备大量的对应新街口的新字段或者数据，对后续再次复习该脚本早成困惑
3. 当上线或者有线上问题需要旧接口验证的时候，如果没有当前场景的特定数据存在，在复习脚本的代码逻辑，参数，以及配置文件的同时，还要修改脚本内部参数数据
4. 如果有部分数据写死到了程序中，那么后续在触发多场景数据测试的时候，就会每执行一个用例修改一次脚本代码，或者写多个方法的方式把数据封装到内部，
 - a) 前一种会让我们经常的修改代码源文件，会导致一些意想不到的问题出现，
 - b) 后一种会导致程序的长度过长而且执行的都是相似的功能。
5. 有些接口在请求之后，之前的数据就不能在此使用了，比如说注册，改密码，改用户名，关注等类似的接口，这就导致在使用完脚本后之前的数据就完全失效的情况，还要花费很多时间再次准备数据，而在此准备的数据依旧用完就不能再使用了
6. 接口之前存在依赖关系的时候，总是要把其他接口封装到本接口的脚本中，同时还要小心接口依赖时候的数据同步关系，因为接口存在依赖关系，那表示数据也存在这一定的依赖，如果数据不匹配总会报错误。在这种场景下，很难实现接口和接口之前的解耦不解耦那么接口在被其他程序调用的重复利用率就很低
7. 一旦接口逻辑需要修改，可能导致整个脚本要修改的地方非常多，而且很可能是一个一个修改，成本非常高，而且修改完成后，还要大量的时间去调试验证，如果修改过程中涉及到了其他服务的接口变动，那么将是无休止的修改工作。。。。
8. 接口测试完成之后，还需要准备一套性能测试的脚本。某一个性能测试的框架很难满足这么多的接口，那就总是需要我到框架里面去简单的改一改，改一改，忘一忘，改一改，忘一忘。。呵呵！～

在此种背景之下，我开发了一套解决上述问题的工具，该工具具有如下特点：

1. 配置文件统一，数据文件统一
2. 采用 python exec 和 eval 两个函数进行动态调度
3. 封装了 sql, mongo, redis 的实例，用于数据恢复和查询
4. 在动态调度的基础上，实现了所有接口代码一致，

5. 在动态调度的基础上解决了，接口依赖，数据依赖的问题
6. 在代码一致的基础上，较少了代码复习，配置文件复习，数据文件复习的成本
7. 在代码一致的基础上，减少了新街口管理的成本
8. 工具本身具备数据初始化功能，只要填写特定的配置文件，执行一个定义好的 shell 脚本，新街口的所有文件，数据，以及配置文件，执行文件，全部一键配置完成，直接可以运行
9. 接口的运行方式：
 - a) 单接口单数据 用于快速确认接口的状态，并且后续被多数据调用，作为依赖被调用，后续会讲
什么意思：我想快速的验证一下某一个接口是不是正常的。给它运行一个正常的数据进去，看看能不能正常返回
 - b) 单接口多数据 用于执行一个接口下多组数据
什么意思：我想验证一个接口在很多数据场景下，能不能给我对应的返回结果
 - c) 服务下一类接口多数据
什么意思：我想运行用户服务下，和 token 相关的所有接口，多数据的场景，每个接口能不能给我对应的返回结果
 - d) 服务下所有接口多数据
什么意思：上面的部分接口换成了所有接口呗，就这个意思
10. 日志级别，正常，DEBUG, DETAIL，目前还不是很正规，后续会更新
11. 所有服务管理的接口，都可以在外部直接导入调用，不用关心路径问题。

不理解？:python 脚本的运行都是以当前脚本路径为准的，这给配置文件和数据文件的读取会造成一定的困难，因为你要考虑不同运行路径下，你的数据能不能找到正确的路径进行读取，否则是会报错的。。。如果还不懂，百度把。。

那到底是什么意思：意思是当前工具无论从哪里运行，都可以正常读取它自己的配置文件和数据文件!!!
12. 封装了 python +shell 的性能测试框架，其中参数和 url 的生成完全依赖当前工具
13. 为了弥补 python+shell 在性能测试中，速度慢，单节点灌不满的情况，工具中封装了 siege，存 C 语言开发，单节点灌满不是梦，post, get 都行

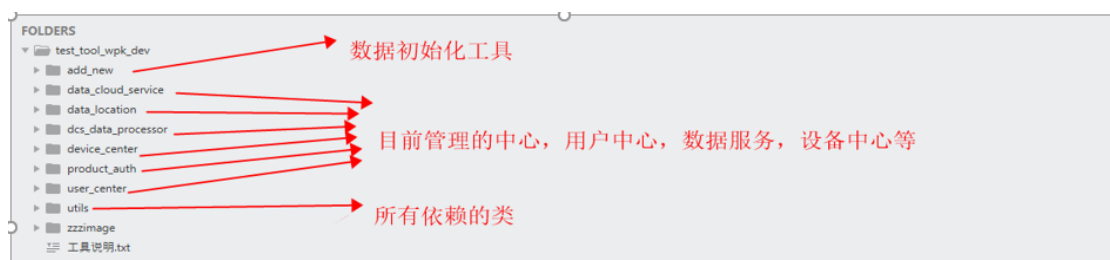
说了这么多，下面就用例子的方式，把上面的特点都演示一遍~

工具使用场景和举例

首先，先介绍一下工具的目录结构，以便后续能够找到想要执行某个操作的可执行文件在什么位置

根目录结构：

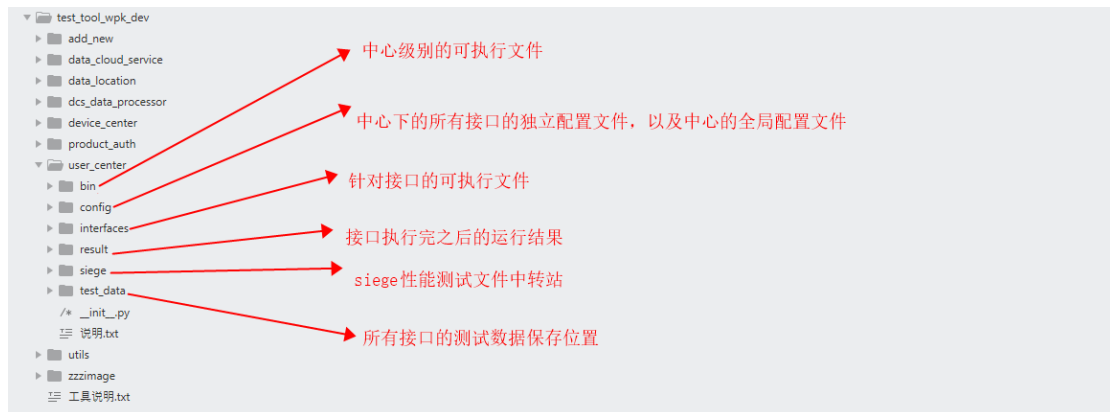
1. add_new—>数据初始化工具 会在后面新接口管理的时候介绍
2. 目前管理的中心，如果后续管理的中心越来越多，会将这些服务放到统一的目录中
3. utils 所有公共抽象的方法，都在这个下面，比如文件读取，签名定义，时间定义等等
4. zzzimage 文件暂时可以忽略，它是图片上传的图片保存路径，后续会移动到内部



所有中服务的目录结构都是一样，包括配置文件，数据文件，和代码逻辑全部都一样，下面用用户服务来举例说明一下

用户服务目录结构：

1. bin 是服务级别的可执行文件，前面说的服务下某一类接口多数据执行，以及服务下所有接口多数据执行都是在这下面后面会展开介绍
2. config 下是所有的配置文件，包括全局配置文件，以及每个接口的个性配置文件，配置文件中的内容后续会展开介绍
3. interfaces 是单个接口的可执行文件，对应上面说的，单接口单数据执行，单接口多数据执行,性能测试，siege 性能测试
4. result 就是每个接口单独执行完成后，会保存结果到这个下面，如果是服务运行，则保存所有结果到一个文件，
5. siege 是性能测试的中转文件夹，使用者不用关心
6. test_data 是上述所有测试读取数据的位置，后面展开说



了解了目录结构，下面开始用例子介绍怎么让工具工作，在介绍工具的同时讲解全局配置文件和接口配置文件以及数据文件字段的意义

场景 1 单接口单数据运行，公共依赖(时间，签名)

白话：我要快速验证一个服务下的一个接口，正向数据或者特定的 1 组数据的返回结果是不是正常，你需要干啥

公共依赖：用户服务的逻辑中，请求参数需要有签名时间戳，而且所有的接口都有，并且它不是死的，是活的。。。。，所以要定义一个类去单独生成它，所以它是用户中线的所有接口的公共依赖，我起的名字。

场景 2 单接口多数据运行，公共依赖(时间，签名)

白话：我要快速验证一个服务下的一个接口，多组数据下，返回结果是不是正确的返回结果，你需要干啥

场景 3 单接口单数据运行 公共依赖(时间，签名)+接口依赖

白话：前半句都一样，我还是想验证某一个接口的一组数据的返回结果，但是这个接口有一个参数，它的值需要我调用另一个接口来给我返回，这种场景下，你需要干啥

场景 4 单接口多数据运行 公共依赖(时间，签名)+接口依赖

白话：理解了 1，2，3 这个应该明白了，过！~

场景 5 服务下某一类接口多数据 公共依赖(时间，签名)+接口依赖

白话：我要测试一个服务下的某些接口的多组数据，比如我要测试用户服务下所有和 token 相关的接口的所有测试用例包括正常的，异常的，啥都要执行，那你需要干啥

场景 6 服务下所有接口多数据，公共依赖(时间，签名)+接口依赖

白话：这个就是服务下的全部接口呗。。过~~

场景 1 单接口单数据运行，公共依赖(时间，签名)

接口名字:	login 工具中的名字为 get_flush_token
接口的 url	rest/v2/user/login?
请求方式:	post
请求参数:	account password signature timestamp clientId subsystemId

需要依赖获取的参数: signature timestamp

请求环境: 内网环境

工具执行文件 user_center\interfaces\get_flush_token\interface_for_call\get_flush_token.py

=====

上面的这些信息代表, 我要以 post 的方式, 请求内网环境的获取 flush_token 的接口,同时由于用户服务的要求, 要获取当前时间戳, 并且在请求之前要对所有的参数进行签名, 这俩就是公共依赖。

当前场景, 我只要运行, 工具执行文件, 工具就能自动的帮我执行完成, 先看下效果

```
===== Detail log info =====
==>interface_name: [-->>> get_flush_token <---]
==>request_url:http://10.30.10.32:10080/rest/v2/user/login?
==>request_method:post
==>request_params:{"account": "zhenzhen", "timestamp": "1558407310", "clientId": "uc_py_test_2", "signature": "90D6F3F8A4E88A5A5EB74CE61A1E67F66D7F4C6A", "subsystemId": "5", "password": "96e79218965eb72c92a64dd5a330112"}
-----result-----
==>return_result:{"result":{"FlushToken":"uf_ghZ6ot0871fgGjxV53doMGeHEXrg7612fJ6UgtepGYXi72F1wF0a+sZXRaCa87HE7jTFiHJ/zUpFlupfubofmXDFZ4hxEl1","validTime":31536000},"returnCode":"uc_0000","costTime":"54","message":"操作成功"}
["result":{"FlushToken":"uf_ghZ6ot0871fgGjxV53doMGeHEXrg7612fJ6UgtepGYXi72F1wF0a+sZXRaCa87HE7jTFiHJ/zUpFlupfubofmXDFZ4hxEl1","validTime":31536000},"returnCode":"uc_0000","costTime":"54","message":"操作成功"}
[Finished in 0.4s]
```

可以看到成功返回了结果, 请求参数中包括时间戳和签名---这个返回结果应该能看懂吧

先看下 user_center\interfaces\get_flush_token\interface_for_call\get_flush_token.py 下的代码是什么样的, 是不是把请求数据, url 啥的都写进了程序里面了。

```
1  #! python2
2  #coding: utf-8
3  import sys
4  import os
5
6  #无论哪个文件执行, 只要执行, 就把当前路径切换到脚本的根目录
7  #目前执行, 只有两地方, enter和脚本, 如果后续还有其他的执行地可以在这个地方扩展
8  if __name__ == '__main__':
9      if "interfaces" in os.getcwd():
10         os.chdir("../..")
11         sys.path.append(".")
12     else:
13         os.chdir("..")
14         sys.path.append(".")
15
16
17 from utils.father_for_call import Father_For_Call
18
19 CENTER_NAME = "user_center"
20 LOCAL_CLASS_NAME = 'get_flush_token'
21 class Get_Flush-Token(Father_For_Call):
22
23     def __init__(self, content_error=None, content_test=None, database_instance_dict={}):
24         #=====
25         #前端传递过来的运行来源
26         self.center_name = CENTER_NAME
27         #运行接口名字
28         self.interface_name = interface_name.lower()
29         #本地文件的名字, 写死的用于读取本接口文件
30         self.own_interface_name = LOCAL_CLASS_NAME.lower()
31         #根据运行来源打开运行错误日志, 目前只有在入口执行才有日志
32         if content_error == None:
33             self.error_log_global = None
34             self.test_log_global = None
35         else:
36             self.error_log_global = content_error
37             self.test_log_global = content_test
38
39         self.global_config = None
40         self.server = None
41         self.proxy = None
42         self.default_data = None
43         self.interface_config = None
44         self.request_method = None
45         self.interface_url = None
46         self.success_str = None
47         self.log_level = None
48         self.script_init_flag = None
49         self.times = None
50         self.database_instance_dict = database_instance_dict
51
52
53     def main(self):
54         result = self.default_test()
55         print(result["result"])
56
57 if __name__ == '__main__':
58     ss = Get_Flush-Token()
59     ss.main()
```

就这些内容，只是定义了一下全局属性而已，真正的逻辑全部抽取到了父类的 Father_For_Call 中，

Father_For_Call 的位置：`utils\father_for_call.py`

如果现在打开 Father_For_Call 你会发现，在它下面也没有写入任何的数据，都是从外部读取的。有人可能会问，那你那些东西都是哪来的？

下面针对，需要读取的内容介绍文件读取的位置

首先确定请求环境，

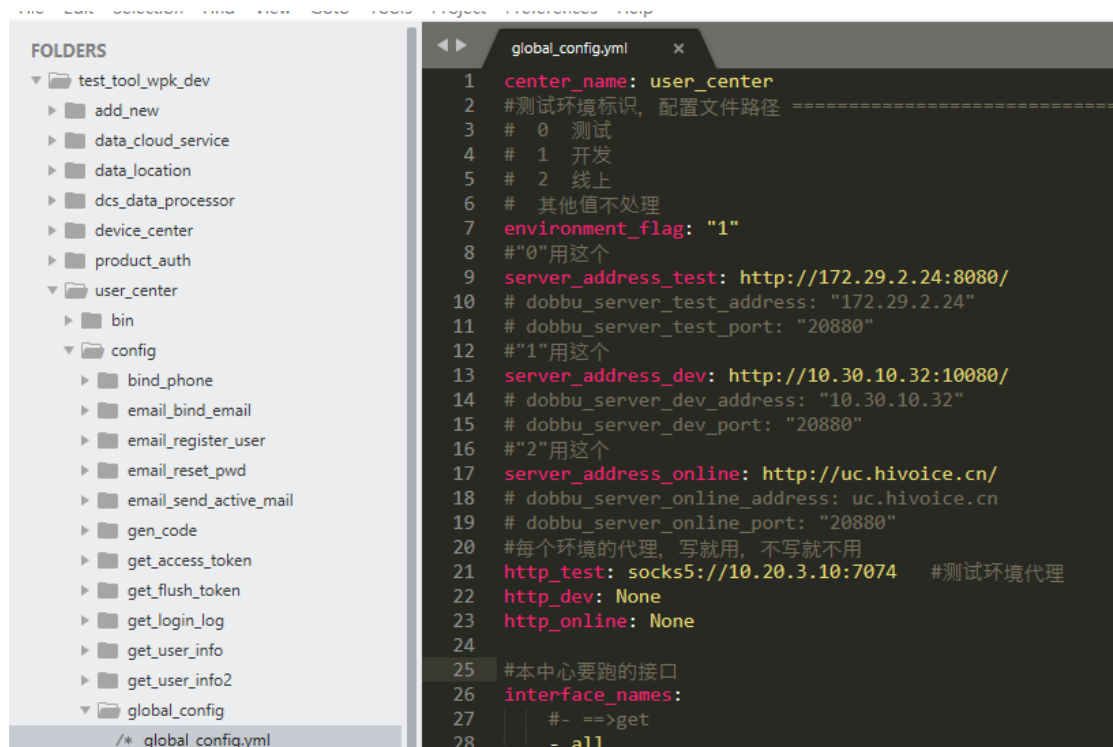
我们的请求环境分为，内网，线上测试和线上，我们希望在运行接口的时候，选择一个环境去运行，通常是运行之前

这个参数就已经确定了，所以把它放在了服务的全局配置文件中。

`user_center/config/global_config/global_config.yml`

下图中，`server_address_*`代表了不同的测试环境的地址，如注释说，0 为线上测试，1 为内网，2 为线上

也就是说在运行 `get_flush_token` 脚本之前，如果我要选择开发环境那么，只要把对应的地址填写到对应的位置，把标识填写为 1，则完成请求环境的配置



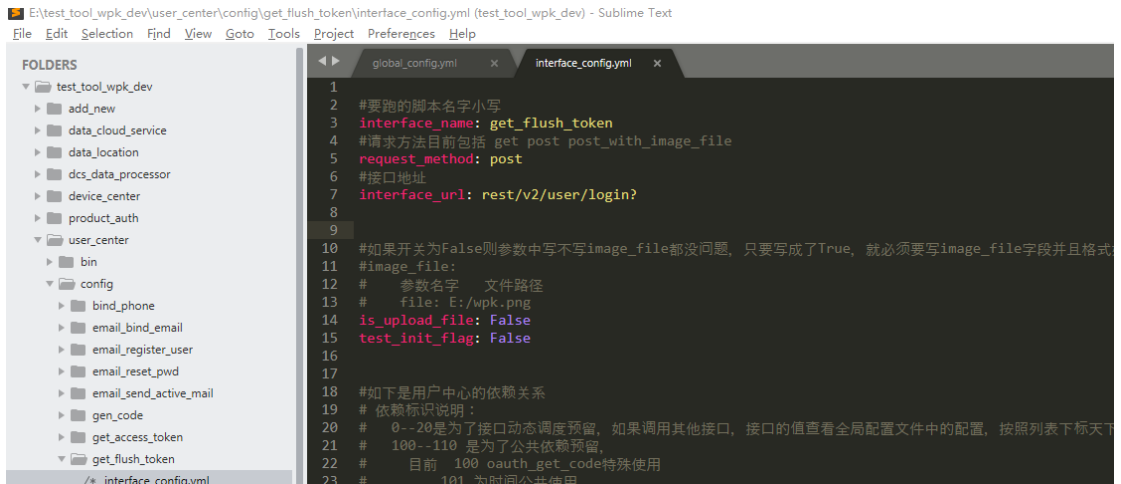
确定请求方法，接口 url、

这两个参数很明显是每个接口都不一样的，所以一定是跟着接口的配置文件走的，接口配置文件位置

`user_center/config/get_flush_token/interface_config.yml`

可以看到请求的接口名字，接口请求方法以及接口请求的 url 都是接口的配置文件中。

也就是说在调用 `get_flush_token` 之前，我需要把这三个必要的参数按照接口文档填写到当前的配置文件中



```
1
2 #要跑的脚本名字小写
3 interface_name: get_flush_token
4 #请求方法目前包括 get post post_with_image_file
5 request_method: post
6 #接口地址
7 interface_url: rest/v2/user/login?
8
9
10 #如果开关为False则参数中写不写image_file都没问题，只要写成了True，就必须写image_file字段并且格式
11 #image_file:
12 # 参数名字 文件路径
13 # file: E:/wpk.png
14 is_upload_file: False
15 test_init_flag: False
16
17
18 #如下是用户中心的依赖关系
19 # 依赖标识说明：
20 # 0--20是为了接口动态调度预留，如果调用其他接口，接口的值查看全局配置文件中的配置，按照列表下标天下
21 # 100--110 是为了公共依赖预留，
22 # 目前 100 oauth_get_code特殊使用
23 # 101 为时间公共使用
```

之后确定，依赖关系

当前接口根据用户服务的要求，需要生成签名，和当前的时间戳，因为这两个参数是公共的，用户服务的其他所有接口所依赖的都是同一个方法，所以把这两个参数抽离出来做成依赖，放在 utils\gen_signature, utils\get_time，那么作为我们在请求接口之前要做什么？我们只要在接口的配置文件中，告诉工具我依赖的是什么就可以了，语法见下图：

可以看到，还是在接口的配置文件中，下面的 params 对应的就是参数和依赖关系

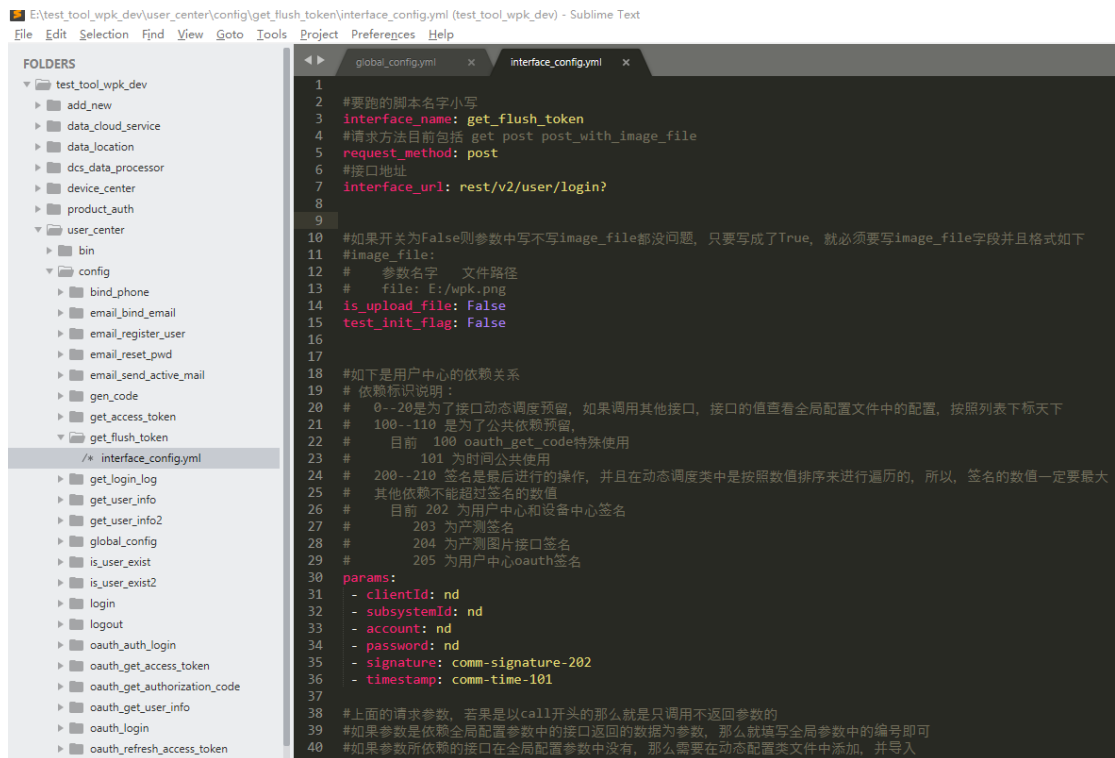
格式是固定的，前面的空格和 - 不能省略，后面跟参数名字，如果该参数不依赖任何东西，就直接写 nd，not-dependency 的意思

下面的 signature 和 timestamp 需要依赖，comm 代表公共依赖，signature 代表依赖的是谁，202 代表依赖的编号，在上面的注释

中已经定义好了依赖编号对应的是哪些依赖，并且为后续工具扩展预留了空间

在本接口请求过程中，时间和签名都是公共依赖，对应的编号也在其中，按照自己的需求填写即可，

注意依赖标识中间要以 - 分割，不可省略，也不可替换成其他



```
1
2 #要跑的脚本名字小写
3 interface_name: get_flush_token
4 #请求方法目前包括 get post post_with_image_file
5 request_method: post
6 #接口地址
7 interface_url: rest/v2/user/login?
8
9
10 #如果开关为False则参数中写不写image_file都没问题，只要写成了True，就必须写image_file字段并且格式如下
11 #image_file:
12 # 参数名字 文件路径
13 # file: E:/wpk.png
14 is_upload_file: False
15 test_init_flag: False
16
17
18 #如下是用户中心的依赖关系
19 # 依赖标识说明：
20 # 0--20是为了接口动态调度预留，如果调用其他接口，接口的值查看全局配置文件中的配置，按照列表下标天下
21 # 100--110 是为了公共依赖预留，
22 # 目前 100 oauth_get_code特殊使用
23 # 101 为时间公共使用
24 # 200--210 签名是最后进行的操作，并且在动态调度类中是按照数值排序来进行遍历的，所以，签名的数值一定要最大
25 # 其他依赖不能超过签名的数值
26 # 目前 202 为用户中心和设备中心签名
27 # 203 为产测签名
28 # 204 为产测图片接口签名
29 # 205 为用户中心oauth签名
30 params:
31 - clientId: nd
32 - subsystemId: nd
33 - account: nd
34 - password: nd
35 - signature: comm-signature-202
36 - timestamp: comm-time-101
37
38 #上面的请求参数，如果是call开头的那么就是只调用不返回参数的
39 #如果参数是依赖全局配置参数中的接口返回的数据为参数，那么就填写全局参数中的编号即可
40 #如果参数所依赖的接口在全局配置参数中没有，那么需要在动态配置类文件中添加，并导入
```

现在请求的 url，方法，请求环境，以及参数的依赖关系都已经确定，就需要读取测试数据了

测试数据在什么地方：

user_center\test_data\get_flush_token\default\test_data_dev.yml

你会发现，在 default 文件夹下，不只有 test_data_dev.yml 还有 test_data_test.yml 和 test_data_online.yml，没错.dev 就是针对内网环境的测试数据，其他两个不言而喻，内部的数据结构是完全一样的，

现在来看一下，内部的构造，我们需要的参数在什么地方

请求参数: account password signature timestamp clientId subsystemId

可以看到，我们需要的请求参数都在这个文件中，这里面还有一些不是我们参数的东西，后面会慢慢介绍，先简单说一下

先说 signature 和 timestamp，

在前面的接口配置文件中，设置了依赖关系，那现在参数的值是 normal 是什么意思，这个地方有点绕，**请注意**

前面我的确设置了依赖，但是如果我设置了依赖它就给我传一个正常值回来，那我想触发一个错误的时间戳，或者

错误的签名的场景怎么办，因为接口的配置文件是针对改接口下所有的测试用例的，如果我通过修改配置文件的话，那么

接口下的所有测试用例都会发生变化，这肯定是不行的，所以想要触发特殊场景一定的用例单位的行为，所以要加一个双重

条件，也就是 normal 的由来，

如果配置文件中设置了依赖，并且当前字段是 normal 那么才会传回一个真正正常的值，

如果我配置文件中设置了依赖，但是这里面不是 normal 而是其他值，那么这个参数就等于其他值，也就完成，异常场

景的触发，同时也不会影响其他真正想依赖的用例—bingo!~

descript 就是用例的描述，当前用例是一个什么场景，它会随着用例的执行完成随着结果输出，兼容中文

script_init_flag:是用例级别的数据库初始化标识

mongo_command: 是执行完用例之后，要执行的 mongo 操作

redis_command:同上

sql_command:同上

success_str: 当前用例可能是正常用例，也可能是异常用例，那么这个字符串只要出现在返回的结果中，我就认为他是一个正常的返
结果，也就是通过

sync_data_flag: 前面说过，本工具解决了接口依赖，在接口依赖的过程中，则一定会存在数据依赖，该开关如果打开，那么该接口
依赖的所有接口的请求数据，都会优先从这个用例下面的数据中读取

times: 该用例执行的次数

image_file: 兼容上传文件的接口

至此，单接口单数据的执行就完成了，脚本内部的实现可以先不用关心，等熟悉了使用以后再研究逻辑

场景 2 单接口多数据运行，公共依赖(时间，签名)

接口名字:	login 工具中的名字为 get_flush_token
接口的 url	rest/v2/user/login?
请求方式:	post
请求参数:	多组 正常或者异常 account password signature timestamp clientId subsystemId
需要依赖获取的参数:	signature timestamp
请求环境:	内网环境
工具执行文件	user_center\interfaces\get_flush_token\interface_func_test\get_flush_token_func_test.py
=====	

还是以 get_flush_token 为例, 注意上面红色的部分发生了变化

单接口多数据运行, 意思就是, 在不改变脚本的情况下, 以多组不同的数据来驱动脚本运行, 相当于, 逐个读取已经定义好的数据, 然后传入脚本中执行, 主要用来验证, 接口在正常场景, 或者异常场景下, 能否执行特定的逻辑, 是否会报错误

同样, 先看一下 效果

```
all database init flag is false
-pass-No.1--接口名字flushtoken--场景-正常获取flushToken
-->{"result":{"flushToken":"uf_ghZ6ot0871fgGjxv53doMGceHEXrg761kM8cAf4F2Q8Gx7n5H0qia3kJVG1TgEjaoXfjTKYpFxC+x+dmCsev2qwbjZy1cum6","validTime":31536000}}
-pass-No.2--接口名字flushtoken--场景-触发用户名不存在
-->{"returnCode":"uc_0219","costTime":"3","message":"账户名不存在, 请重新输入"}
-pass-No.3--接口名字flushtoken--场景-触发密码不正确
-->{"returnCode":"uc_0207","costTime":"3","message":"账户名与密码不匹配, 请重新输入"}
-pass-No.4--接口名字flushtoken--场景-触发时间戳不正确
-->{"returnCode":"uc_0210","costTime":"1","message":"请求时间戳超出了请求有效期"}
-pass-No.5--接口名字flushtoken--场景-触发签名不正确
-->{"returnCode":"uc_0001","costTime":"2","message":"签名错误"}
-pass-No.6--接口名字flushtoken--场景-触发子系统不存在
-->{"returnCode":"uc_0216","costTime":"2","message":"没有找到对应的子系统信息"}

[Finished in 0.7s]
```

可以看到上面执行了 6 条用例。分别触发了不通的场景, 并且测试全部都通过了, 为啥异常场景也通过了, 还记得全面说道的数据里面的 success_str 字段吗。后面会看到

看下 user_center\interfaces\get_flush_token\interface_func_test\get_flush_token_func_test.py 代码的样子

和 get_flush_token.py 类似, 当前文件只是定义了一些实例属性, 真正的代码逻辑都存在于 Father_Func 中


```

#!/ python2
#coding: utf-8
import sys
import os

#=====
#无论哪个文件执行，只要执行，就把当前路径切回到脚本的根目录
#目前执行，只有两地，enter和脚本，如果后续还有其他的执行地可以在这个地方扩展
if __name__ == '__main__':
    if "interfaces" in os.getcwd():
        os.chdir(".././.././../")
        sys.path.append('.')
    else:
        os.chdir("../")
        sys.path.append("../")
#=====

from utils.father_func import Father_Func

CENTER_NAME = 'user_center'
LOCAL_CLASS_NAME = "Get_Flush-Token"

class Get_Flush-Token_Func_Test(Father_Func):
    def __init__(self, content_error=None, content_test=None, database_instance_dict={}):
        self.center_name = CENTER_NAME
        self.own_interface_name = LOCAL_CLASS_NAME.lower()
        if content_error == None:
            self.error_log_global = None
            self.test_log_global = None
        else:
            self.error_log_global = content_error
            self.test_log_global = content_test

        self.interface_log_content = None
        self.global_config = None
        self.func_test_data = None
        self.func_test_times = None
        self.database_instance_dict = database_instance_dict
        self.test_init_flag = None

        self.count = 0
        self.PASS = 0
        self.FAIL = 0

if __name__ == '__main__':
    ss = Get_Flush-Token_Func_Test()
    ss.main()

```

下面针对，需要读取的内容介绍文件读取的位置

首先确定请求环境，

请求环境上面单接口单用例已经说过了，在 user_center\config\global_config\global_config.yml 中设置对应环境的标识就可以了

确定请求方法，接口 url

因为请求的还是 get_flush_token 接口，接口的请求方法，和 url 都没有发生变化

user_center\config\get_flush_token\interface_config.yml

确定，依赖关系

正如上面所说，因为配置文件针对的是当前接口下所有的用例，所以要在数据文件中加入双保险，来确保正常和异常都能触发

所以接口没有变化所以依赖关系也没有变化

现在请求的 url，方法，请求环境，以及参数的依赖关系都已经确定，就需要读取测试数据了

数据是唯一和单接口单数据变化的地方，首先找到数据位置

user_center\test_data\get_flush_token\func\test_data_dev.yml

同样的上面红色字体发生了变化，由 default 变成了 func

在 func 下也就有个文件对应 dev test online 3 个环境，当全局配置文件环境标识发生变化，读取当前位置的文件也会随着标识变化

文件内容：

可以看到，当前文件的格式，在- 下面的数据结构，和 default 是一样的，也就是说，脚本会到当前文件读取所有的数据，形成一个

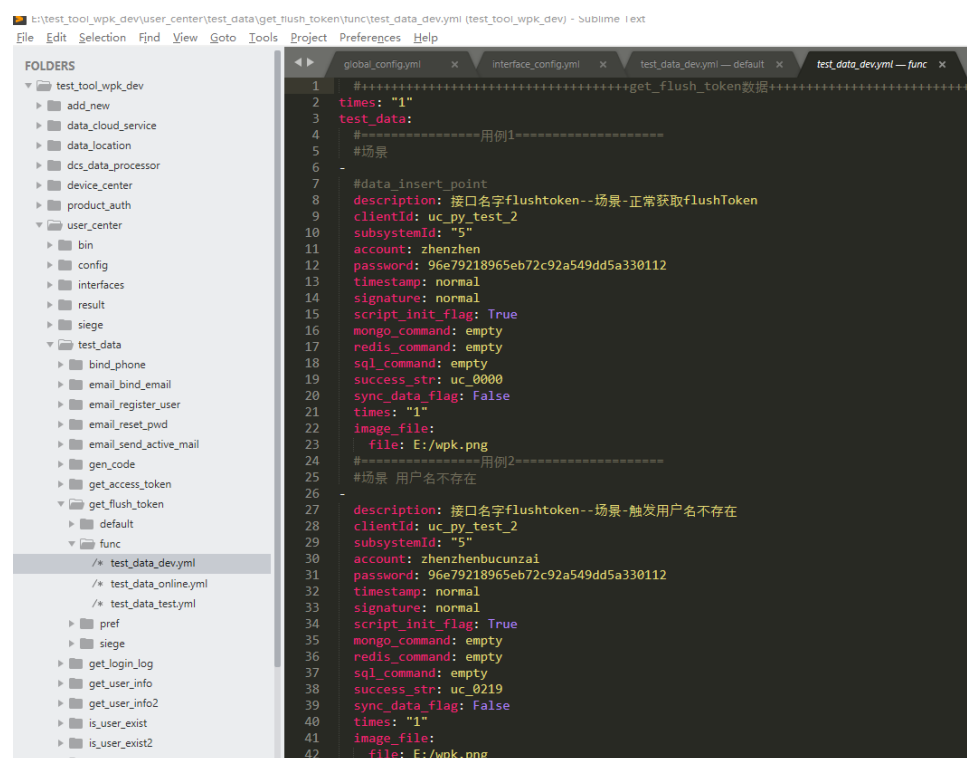
可迭代对象，然后对这个数据进行迭代，从而完成单接口多数据的场景测试

以 - 分割每一个 - 代表一条用例，逐条执行，请在回过头看下执行的效果就明白了

上面的 times 是当前文件执行多少次，也就是所有用例执行多少次

用例内部的 times 是单条用例执行多少次，都可以灵活配置、

在说一下 success_str 字段，可以看第二用例，为 uc_0219 所以如果结果中有 uc_0219 我也认为他通过了



最后，如果你想触发更多的用例，只要来修改这个文件就行了，按照格式要求，加一个“-”就一个用例，加一个“-”就是一个用例

脚本咋写跟你没关系啦！～

场景 3 单接口单数据运行 公共依赖(时间，签名)+接口依赖

接口名字:	获取 accessToken	工具中的名字为 get_access_token
接口的 url	rest/v2/token/get_access_token?	
请求方式:	post	
请求参数:	flushToken signature timestamp clientId subsystemId	
需要依赖获取的参数:	signature timestamp flushToken	
请求环境:	内网环境	
工具执行文件	user_center\interfaces\get_access_token\interface_for_call\get_access_token.py	

同样先来看下效果

```
===== Detail log info =====
==>Interface name: |--> get_flush_token <---|
==>request_url:http://10.30.10.32:10080/rest/v2/user/login?
==>request_method:post
==>request_params:{"account": "zhenzhen", "timestamp": "1558413732", "clientId": "uc_py_test_2", "signature": "AA466637CAD4FF5A9CBED7E7A700884CC16E7CA1", "subsystemId": "5", "password": "96e79218965eb72c92a549dd5a330112"}
==>result:-----
==>return_result:{"result":{"flushToken":"uf_ghZ6ot0871fgGjv53doMceHEXrg761bCidI45/1151fk5MGqCBjG+M80a1fDs/lp58up0b5iHwFHo8IcoVlGmSK3042gb/","validTime":31536000},"returnCode":"uc_0000","costTime":"49","message":"操作成功"}

===== Detail log info =====
==>Interface name: |--> get_access_token <---|
==>request_url:http://10.30.10.32:10080/rest/v2/token/get_access_token?
==>request_method:post
==>request_params:{"subsystemId": "5", "flushToken": "uf_ghZ6ot0871fgGjv53doMceHEXrg761bCidI45/1151fk5MGqCBjG+M80a1fDs/lp58up0b5iHwFHo8IcoVlGmSK3042gb/", "timestamp": "1558413732", "clientId": "uc_py_test_2", "signature": "AA466637CAD4FF5A9CBED7E7A700884CC16E7CA1"}
==>result:-----
==>return_result:{"result":{"validTime":86400,"accessToken":"ua_Td0gat5rx1zIsRMy3vRPh011wI093eokR123xKI09YtT6RqQcXTkf30H/IzyfMkVYVdMkvzh10gpFm0s2ITa0xFVoY9yZ8"},"returnCode":"uc_0000","costTime":"88","message":"操作成功"}

[Result:{"validTime":86400,"accessToken":"ua_Td0gat5rx1zIsRMy3vRPh011wI093eokR123xKI09YtT6RqQcXTkf30H/IzyfMkVYVdMkvzh10gpFm0s2ITa0xFVoY9yZ8"},"returnCode":"uc_0000","costTime":"88","message":"操作成功"}
[Finished in 0.6s]
```

可以看到上面的结果，accessToken 调用成功了，并且它先调用了 get_flush_token 的接口获取了 flushToken

那要完成这样的接口依赖操作，我们需要干什么呢，是不是要在 get_access_token 的内部导入 get_flush_token 的接口并调用呢

我可以告诉你根本不用， 本工具就是来解决这个问题的，要不怎么做到代码统一

先看下，get_access_token.py 的代码逻辑，你可以对比一下 get_flush_token.py 的代码逻辑，他们内部除了类名字不一样之外，其他

完全一样，都是 Father_For_Call 的孩子.而且都是调用的是父类的 default_test 方法，完全一样的执行流程，但是结果却 accessToken

调用的 flushToken 这才是工具真正具有威力的地方，那我们到底需要干什么呢，往下看

```
!_access_token\interface_for_call\get_access_token.py (test_tool_wpk_dev) - Sublime Text
Project  Preferences  Help

global_config.yml  x  interface_config.yml  x  test_data_dev.yml  x  get_access_token.py  x

1  #! python2
2  #coding: utf-8
3  import sys
4  import os
5  #=====
6  #无论哪个文件执行，只要执行，就把当前路径切换到脚本的根目录
7  #目前执行，只有两地，enter和脚本，如果后续还有其他的执行地可以在这个地方扩展
8  if __name__ == '__main__':
9      if "interfaces" in os.getcwd():
10         os.chdir("../..")
11         sys.path.append(".")
12     else:
13         os.chdir("../")
14         sys.path.append(".")
15 #=====
16
17 from utils.father_for_call import Father_For_Call
18
19 CENTER_NAME = "user_center"
20 LOCAL_CLASS_NAME = 'get_access_token'
21 class Get_Access_Token(Father_For_Call):
22
23     def __init__(self, content_error=None, content_test=None, database_instance_dict={}):
24         #=====
25         #前端传递过来的运行来源
26         self.center_name = CENTER_NAME
27         #运行接口名字
28         # self.interface_name = interface_name.lower()
29         #本地文件的名字，写死的用于读取本接口文件
30         self.own_interface_name = LOCAL_CLASS_NAME.lower()
31         #根据运行来源打开运行错误日志，目前只有在入口执行才有日志
32         if content_error == None:
33             self.error_log_global = None
34             self.test_log_global = None
35         else:
36             self.error_log_global = content_error
37             self.test_log_global = content_test
38
39         self.global_config = None
40         self.server = None
41         self.proxy = None
42         self.default_data = None
43         self.interface_config = None
44         self.request_method = None
45         self.interface_url = None
46         self.success_str = None
47         self.log_level = None
48         self.script_init_flag = None
49         self.times = None
50         self.database_instance_dict = database_instance_dict
51
52
53     def main(self):
54         result = self.default_test()
55         print(result["result"])
56
57 if __name__ == '__main__':
58     ss = Get_Access_Token()
59     ss.main()
```

和 get_flush_token 单接口单数据类似

接口执行位置 user_center\interfaces\get_access_token\interface_forcall\get_access_token.py

全局文件中确定请求环境

接口配置文件中配置请求方法，以及接口 url

但是该接口除了公共依赖之外，还依赖的 get_flush_token 既然不是公共依赖而是接口依赖，那么接口配置文件中的依赖关系应该如何写？

可以看到，在下图中，

其他参数都没有什么变化，只有 flushToken 参数比较特殊，下面来解释它是什么意思

Yes_interface-user_center-get_flush_token-5 就是它了，就是这么一句话就实现上面 accesstoken 调用 flushtoken

首先上面的数据是用 - 分割的

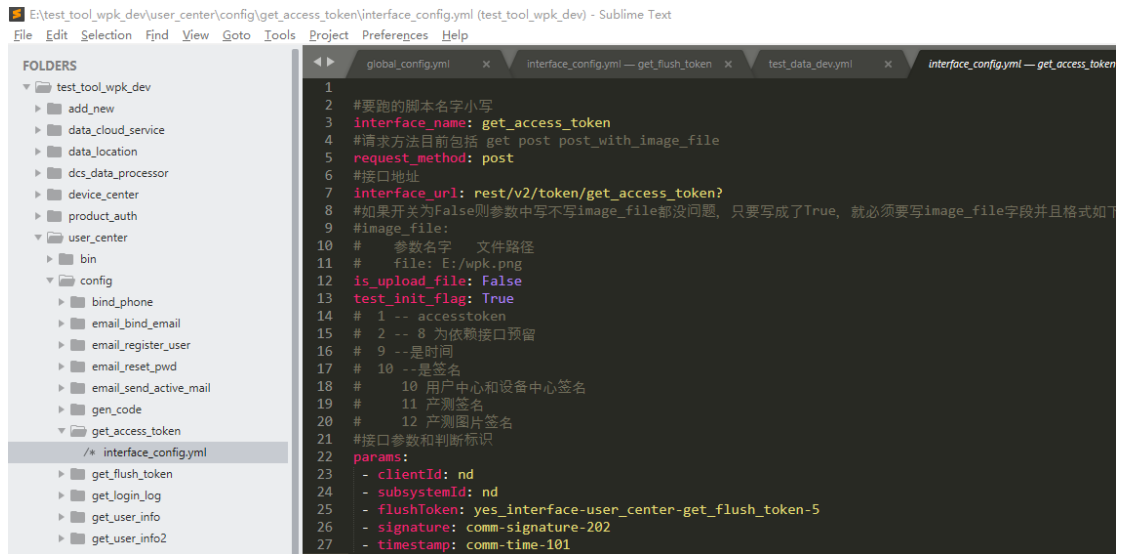
yes_interface 代表依赖的是接口而不是公共依赖

user_center 代表依赖的是用户服务

get_flush_token 代表依赖的是 get_flush_token 接口

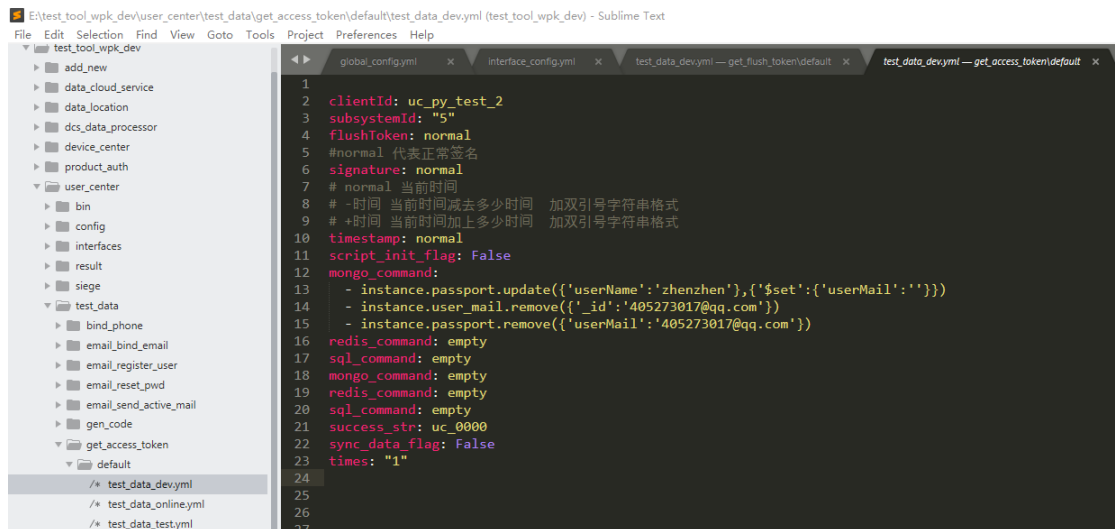
5 代表你要 get_flush_token 给你返回的结果中，用引号分割后，你要获取参数所在的下标位置

注意：这里要说明一点，可能当前的结构看起来不是很明确，可能有人会问为什么不用键值对的方式，去确定依赖关系，那样的话更精确，没错的确是更精确，但是，如果一个接口依赖了很多个接口的话，那会导致这个配置文件 params 下的字段非常多，反而没有这样的可读性高了，还有最后的下标 5，有人会问，为什么不用 json 的方式提取，更精确，没错确实 json 更精确，但是考虑到所有接口的返回值中，所获取的结果的位置或者深度都是不固定的，这样兼容性不是很好，以下标的方式，反而是兼容性更好的选择，因为本工具要考虑兼容很多接口，而不是某一个服务。So。。。。



解决了依赖关系--接下来看数据在哪

可以看到 accesstoken 的数据和 flushtoken 的数据完全一样，只是多了一个 flushToken 的参数，用 normal 的方式进行双保险



综上，我只是在 get_access_token 接口的配置文件中，设置了 Yes_interface-user_center-get_flush_token-5 就完成了接口的依赖过程，

其他配置文件，数据文件，以及代码逻辑，完全一样，

等等，还差一件事情没说，接口依赖解决完成了，那数据依赖呢，

细心的你可能发现了，在 accesstoken 参数中定义的都是自己的参数，那我依赖的 flushtoken 的参数从哪来的

没错，这个也是框架帮你解决的问题，注意下面有点绕

在接口依赖的过程中，在当前场景下，accesstoken 依赖 flushtoken，动态调度的逻辑是这样的，因为所有接口的逻辑

是一样的，也就是说所有接口都具备以下特性

任何人调用我(接口)，你都可以进行 3 种方式调用

1. 不传参数
2. 传一部分参数
3. 传全部参数

一个一个说，

-不传参数的场景下，那么接口会自动读取，test_data\default\对应环境测试文件下数据，作为本次被依赖请求的参数

-传一部分参数的场景下，接口会同自身的配置文件相比对，如果你传的参数存在于我的配置文件中，那么就用，否则不处理

-传全部参数，也就是说调用接口的时候，传入了在接口配置文件中定义的所有参数，那么全部采用

现在清楚了，那我只需要在 accesstoken 调用 flushtoken 的时候给他传递参数就行了呗，但是代码逻辑都一样怎么实现传参数？

这个不是你考虑的问题，是工具解决的问题，你要做的如下

先对比一下两个接口的参数，

flush token 参数： account password signature timestamp clientId subsystemId

access_token 参数： flushToken signature timestamp clientId subsystemId

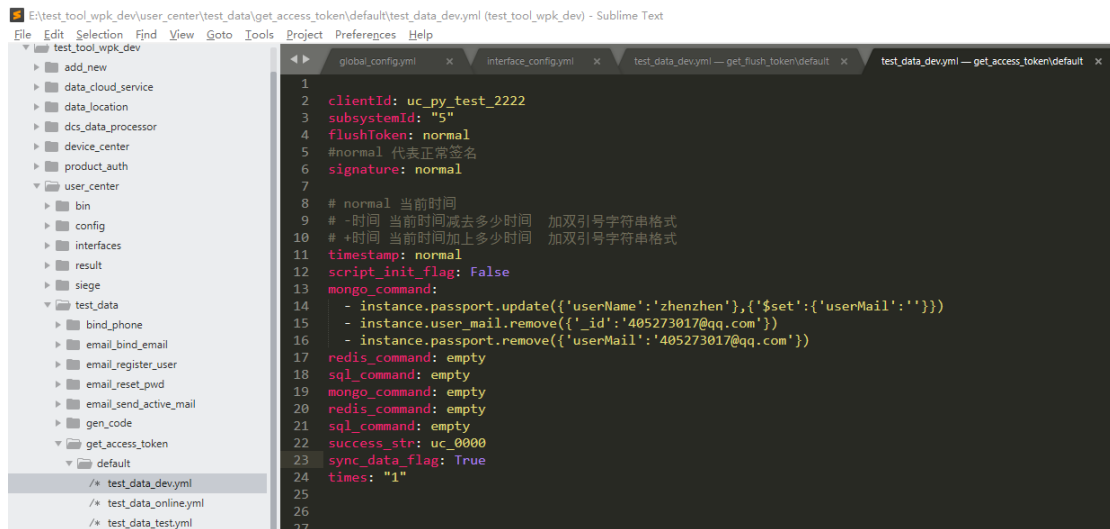
Check 一下 flushtoken 的参数 发现 clientId:uc_py_test_2

而当前 accesstoken 的参数，clientId:uc_py_test_2222

现在我想让 accesstoken 在调用依赖 flush_token 的时候 clientId 参数用我的给它定义的，uc_py_test_2222， 那我需要怎么做
我只需要把 accesstoken 的参数中的 sync_data_flag 改成 True，那么就相当于，flushtoken 用的参数要优先从我这个地方取，如果没有找到
在到 flush_token 自己的 test_data\default\对应环境测试文件下数据下去拿参数

也就是，在当前配置我执行 get_access_token.py 脚本，get_flush_token 脚本的请求的参数中，clientId 应该是 uc_py_test_2222

看下结果下图 2



果然变成了 uc_py_test_2222

那用户名和密码我没给他定义，他哪来的，前面说过了，如果没有就到它自己的 default 下去拿

换句话说，如果我想传一个用户名密码给他，那就 accesstoken 数据文件中定义跟被调用接口配置文件中定义的 key 一样的字段就可以了

那要是依赖多个接口呢，没错，写你想要同步的参数就行，其他不用管

```
===== Detail log info =====
==>interface_name: |-->>> get_flush_token <---|
==>request_url:http://10.30.10.32:10080/rest/v2/user/login?
==>request_method: post
==>request_param:{"account": "zhenzhen", "timestamp": "1558416837", "clientId": "uc_py_test_2222", "signature": "537280AFFA948C5E255396489C9750257E3E330A", "subsystemId": "5", "password": "96e79218965eb72c92a549dd5a338112"}
-----result-----
==>return_result:{"result": {"FlushToken": "uf_ghZ6ot0871fgGjxV53dMBLw//tvy88SeBie9um9xjHkSpYkbEiA31FcX0btrmiVunUYCeXeiXWkIcaQrEhCbtQFuo51rp7u", "validTime": 31536000, "returnCode": "uc_0000", "costTime": "64", "message": "操作成功"}}

===== Detail log info =====
==>interface_name: |-->>> get_access_token <---|
==>request_url:http://10.30.10.32:10080/rest/v2/token/get_access_token?
==>request_method: post
==>request_param:{"subsystemId": "5", "FlushToken": "uf_ghZ6ot0871fgGjxV53dMBLw//tvy88SeBie9um9xjHkSpYkbEiA31FcX0btrmiVunUYCeXeiXWkIcaQrEhCbtQFuo51rp7u", "timestamp": "1558416837", "clientId": "uc_py_test_2222", "signature": "178077E826E076761AC987824860ACE0B96318CE"}
-----result-----
==>return_result:{"result": {"validTime": 86400, "accessToken": "ua_Td0gat5rx1zIsRMy3vRpb3vWnhRgHtK8Z8WRYXp1/Cta7tkvWVf71c/r21HMCASJfFseYckI1d8+U772MjQCqbwTDTcYs", "returnCode": "uc_0000", "costTime": "105", "message": "操作成功"}, "result": {"validTime": 86400, "accessToken": "ua_Td0gat5rx1zIsRMy3vRpb3vWnhRgHtK8Z8WRYXp1/Cta7tkvWVf71c/r21HMCASJfFseYckI1d8+U772MjQCqbwTDTcYs", "returnCode": "uc_0000", "costTime": "105", "message": "操作成功"}}
[Finished in 0.6s]
```

场景 4 单接口多数据运行 公共依赖(时间，签名)+接口依赖

接口名字: accessToken 工具中的名字为 get_access_token

接口的 url rest/v2/token/get_access_token?

请求方式:	post
请求参数:	多组 正常或者异常 flushToken signature timestamp clientId subsystemId
需要依赖获取的参数:	signature timestamp flushToken
请求环境:	内网环境
工具执行文件	user_center\interfaces\get_access_token\interface_func\get_access_token_func_test.py

get_access_token 多用例运行, 先看看效果

跟 get_flush_token 多用例场景类似,

```
-pass-No.1--接口名字get_access_token--场景-正常获取flushToken
-->{"result":{"validTime":86400,"accessToken":"ua_Id0gat5rx1zIsRMy3vRPb01lwIQ9jeokJ2N

-pass-No.2--接口名字get_access_token--场景-clientId为空
-->{"returnCode":"uc_0201","costTime":"0","message":"请求参数不能为空"}

-fail-No.3--接口名字get_access_token--场景-subsystemId为空
==>proxy is: None
==>url is: http://10.30.10.32:10080/rest/v2/token/get_access_token?
==>param is: {"subsystemId": "", "flushToken": "uf_ghZ6ot0871fgGjxV53doMgcHExrg761EY
"signature": "58E74F474736F448B567DC333702275397C1DEA4"}
==>request_method is: post
-----
---read data is:
{"description": "接口名字get_access_token--场景-subsystemId为空", "script_init_flag": "
"mongo_command": "empty", "sync_data_flag": false, "subsystemId": "", "success_str":
-----
---return data is:
-----

-pass-No.4--接口名字get_access_token--场景-时间不匹配
-->{"returnCode":"uc_0210","costTime":"0","message":"请求时间超出了请求有效期"}

-pass-No.5--接口名字get_access_token--场景-签名错误
-->{"returnCode":"uc_0001","costTime":"2","message":"签名错误"}

-pass-No.6--接口名字get_access_token--场景-子系统id不存在
-->{"returnCode":"uc_0216","costTime":"1","message":"没有找到对应的子系统信息")
```

下面针对, 需要读取的内容介绍文件读取的位置

首先确定请求环境,

请求环境在 user_center\config\global_config\global_config.yml 中设置对应环境的标识

确定请求方法, 接口 url

user_center\config\get_access_token\interface_config.yml

确定, 依赖关系

user_center\config\get_access_token\interface_config.yml

测试数据

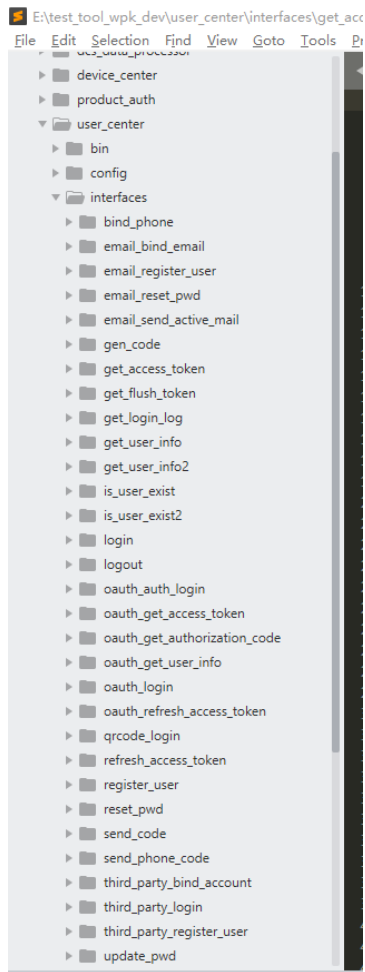
user_center\test_data\get_access_token\func\test_data_dev.yml

场景 5 服务下某一类接口多数据 公共依赖(时间, 签名)+接口依赖

说了上面那么多, 有人会问, 嗯, 现在你能执行, 单个接口快速验证, 单个接口多用例多场景执行

解决了代码的统一, 数据的统一, 配置文件的统一问题, 那么怎么运行一个服务下的所有接口呢, 或者一部分接口呢

下面开始说, 服务级别的运行, 某一类接口, 先看个图



可以看到通常一个服务下的接口名字，是有关联的，也就是开头大体相同。

那么如果我想跑用户服务，下所有以 get 开头的用例，可以吗，可以的

方法：

在用户服务全局配置文件下

字段 interface_names:

我要跑 get_flush_token 一个接口

- get_flush_token

我要跑以 get 开头的接口

- ==>get

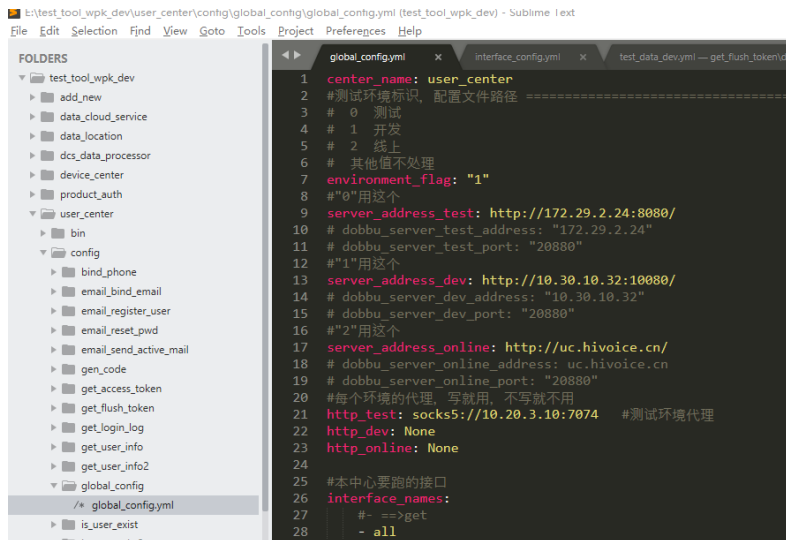
我要跑所有接口

- all

注意 - ==> 和 - all 必须放在第一个位置，才会生效

然后运行

user_center\bin\func\user_center_enter.py



场景 6 服务下所有接口多数据，公共依赖(时间，签名)+接口依赖

场景 5 说完了场景 6 也说完了，按照下面就行了

interface_names:

- all

然后运行

user_center\bin\func\user_center_enter.py

另外服务运行级别下的错误请求，全部都会记录在，user_center\result\error_log.txt 下

其他依赖场景

说完了所有场景，在说说依赖关系，我们之前说的依赖关系，都是要求，依赖某一个接口，请求这个接口，并且把对应结果返回给我当前的参数

params:

- clientId: nd
- subsystemId: nd
- flushToken: yes_interface-user_center-get_flush_token-5
- signature: comm-signature-202
- timestamp: comm-time-101

那么还有其他的依赖场景么。当然

当前我遇到的依赖场景有如下几个：

1. 执行某一个接口之前，只想先调用一下其他接口，而不需要任何的返回值
2. 执行一个接口之前，我要依赖某一个接口的不只一个参数，2 个甚至多个
3. 依赖接口被调用之后，返回的参数并不是直接给我返回的，而是回调传给了其他服务器

以上场景该工具都已经解决语法如下


```
- call: yes_interface-user_center-oauth_login-0

- transId: yes_param-udid-15

- code: comm-code-100
```

以上 3 个依赖语法对应，上面 3 个场景

```
- call: yes_interface-user_center-oauth_login-0
```

以 call 开头代表只是调用一下，某个服务下的某一个接口，不需要返回值，可以使用数据同步调用

```
- transId: yes_param-udid-15
```

yes_param 代表依赖于前面某一个参数的返回结果，这里的 udid 就是接口配置文件中的一个参数，简单说，

udid 依赖外部接口

transId，依赖 udid 返回值

```
- code: comm-code-100
```

comm-code-100 这个是用 paramiko 单独封装的脚本，用来到 linux 上取特定的东西

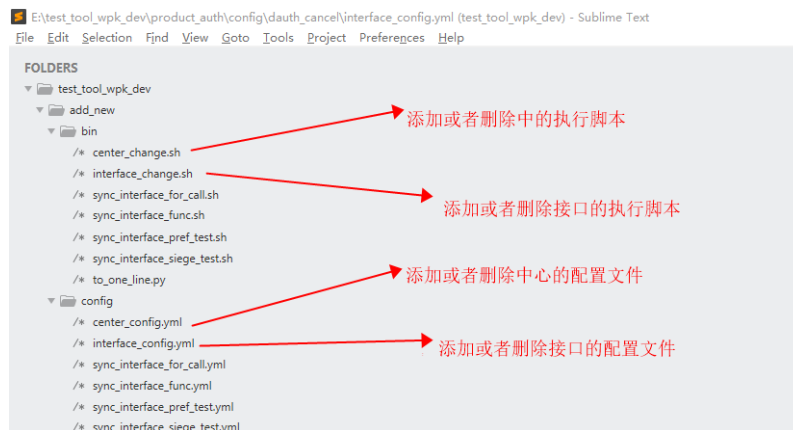
当然，接口的依赖调用，还有很多种场景，只不过目前我遇见的接口依赖关系，当前工具已经全部兼容，后续如果有更复杂的接口，完全可以扩展

接口快速管理工具

通过上述的讲解发现，针对某一个接口的定义还是相对复杂的，有配置文件，有接口执行文件，还有数据文件，难道我要把一个新接口，加入到这个工具中还要手动建立所有的文件，并且填写所有的数据么，当然不是，那不累死了

由于本工具代码，配置文件，数据文件都统一，所以也为快速管理初始化接口打下了非常好的基础，

所以工具本身就提供了接口快速管理工具，add_new 看图

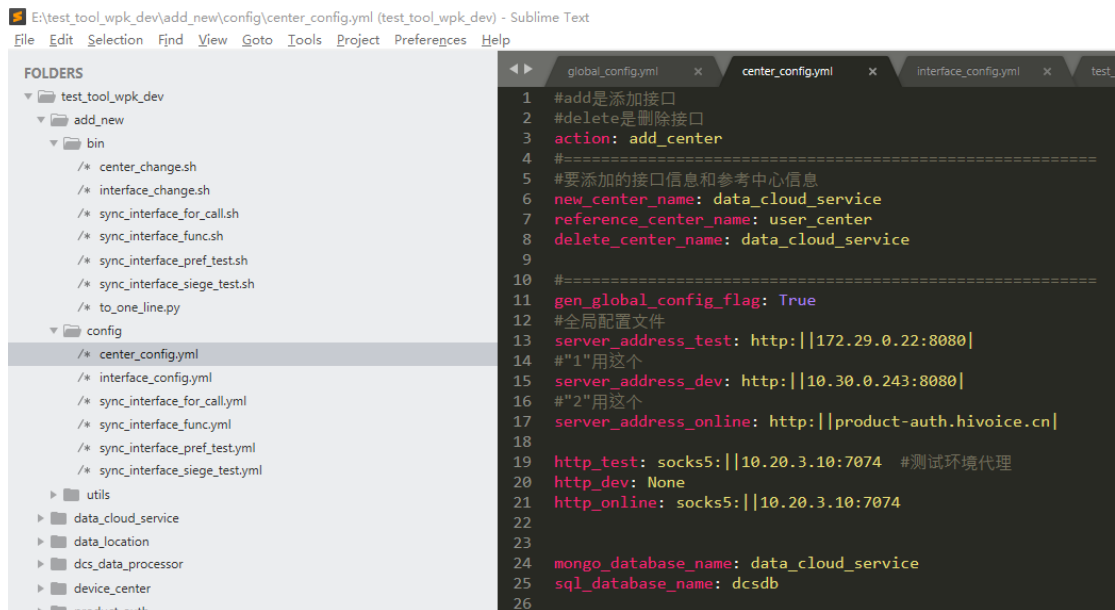


场景 1 增加服务

打开服务的配置文件，action 选成 add_center，字段都很好理解，按照要求填写就可以

如果要生成全局配置文件 就把 gen_global_config_flag: True 否则为 False

其中 reference_center_name: user_center 是参考服务，也就是脚本会按照一个已经存在的服务样式去创建另一个服务



用 shell 执行 center_change 脚本，新服务添加完成

执行环境，windows git_bash

Linux 也行，但是配置麻烦，建议在 windows

场景 2 删除服务

只要把 action 改成 delete_center

在 delete_center_name: data_cloud_service 填写要删除的服务就可以

注意删除服务的时候，做好吧编辑器关闭掉，， 否则 python 脚本可能会因为文件打开而删除不掉

用 shell 执行 center_change 脚本，新服务删除完成

场景 3 添加接口

添加接口操作的执行，必须在服务已经存在的情况下，才能执行，不能添加接口道一个不存在的服务中去，

```
interface_config.yml x
1 #add是添加接口
2 #delete是删除接口
3 action: add_interface
4 #=====
5 #要添加的接口信息和参考中心信息
6 new_center_name: data_cloud_service
7 new_interface_name: audiomaterial_xmly_upload_live_batch_records
8 reference_center_name: user_center
9 reference_interface_name: get_flush_token
10 #要删除的中心信息
11 delete_center_name: data_cloud_service
12 delete_interface_name: audiomaterial_xmly_upload_live_batch_records
13 #=====
14 #是否自动生成配置文件标识
15 gen_config_file: True
16 #=====
17 #+++++
18 #+++++
19 #配置文件信息
20 interface_name: audiomaterial_xmly_upload_live_batch_records
21 request_method: get
22 interface_url: data-cloud-service|rest|v1|audiomaterial|xmly|upload_live_batch_records?
23 is_upload_file: False
24 test_init_flag: False
25 #
26 params:
27   - uni_uid: nd
28   - sid: nd
29   - uni_key: nd
30   - live_records: nd
31   - uni: nd
32   - secret: nd
33   - appkey: nd
34
```

定义行为add为添加，delete为删除

接口添加到哪个中心去

新接口的名字

参考的中心的名字

参考的接口的名字

删除接口时填写

删除哪一个接口

是否生成配置文件

接口名字

接口URL注意要用|替换\

接口是否上传文件

接口是否初始化数据库

```
#配置文件信息结尾
#+++++
#+++++
#是否自动生成数据标识
gen_default_data_file: True
#=====
#内网默认数据
dev__ description: "--data_cloud_service-点播资源--批量上传直播播放数据--正常数据"
dev__ sid: sid
dev__ uni_key: uni_key
dev__ uni_uid: uni_uid
dev__ live_records: ""
dev__ uni: ""
dev__ secret: ""
dev__ appkey: ""

#=====
dev__ script_init_flag: False
dev__ mongo_command: empty
dev__ redis_command: empty
dev__ sql_command: empty
dev__ success_str: success
dev__ sync_data_flag: False
dev__ times: "1"
dev__ image_file:
dev__ file:
```

是否自动生成数据

数据结构和之前的数据结构是一样的
只不过在前面加了dev__来区分，除了dev
还有test__，online__只要按照要求来配
所有数据的文件和文件夹都会按照当前
配置全部生成完毕

用 shell 执行 center_change 脚本，新接口添加完成

执行环境 windows git_bash

接口添加执行完成后，会自动生成，default 和 func 下的所有测试数据，不过只生成一条数据，多条数据的情况需要后续手动添加

场景 4 删除接口

把 action 的 add 改成 delete,在删除接口的时候退出编辑器，否则可能会报错

数据恢复以及数据库操作

针对背景中提到的，数据恢复的问题，本工具专门封装，sql,redis 和 mongo，用于数据查询和恢复，使用方法也很简单

首选数据的查询和恢复一定是用例级别的，所以，数据库的相关操作语句都存在于用例当中，相信在前面的截图中，大家应该已经看到了不少

要用数据库那么就先要初始化数据库，要初始化数据库，那么就需要连接数据库，要链接数据库那就需要数据库的地址和参数，以及数据库的名字

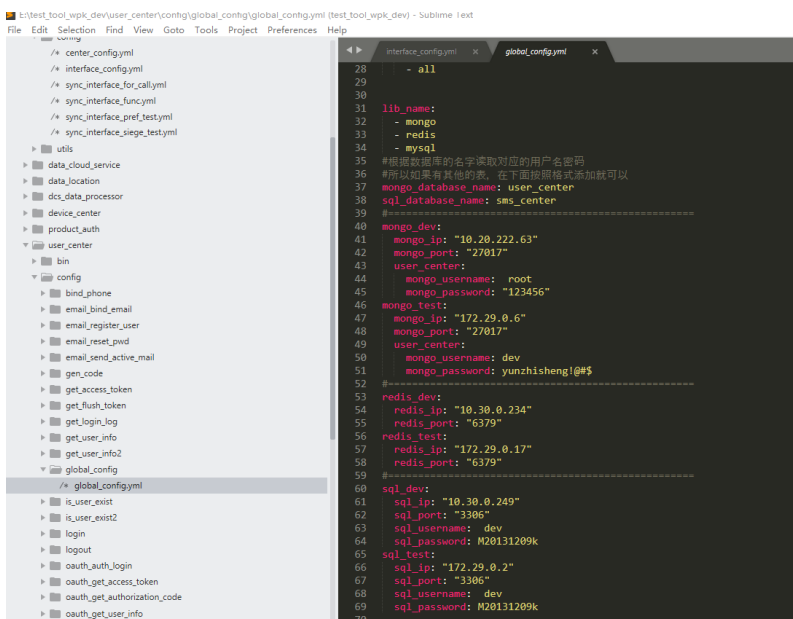
一个一个来，先看数据库在哪里

首先数据库一定是服务级别的所有，一定在全局配置文件中

如下图：

Lib_name 代表了本服务，需要初始化的数据库类型，按照真实情况填写，如果没有就注释就可以了

下面是数据库的地址信息，对应着不同的环境，按照要求填写就好了



现在已经有了链接数据库的地址，那么下面就是要初始化数据库了，初始化数据库有一些要注意的地方，首先前面说过，数据库的操作是用例级别的，但是我不能每个用例都初始化一次数据库，如果以接口为单位初始化数据可也不合适，因为还有服务运行级别，那就意味着，每个接口都初始化一次，也影响效率，但是我还有单个接口单个数据运行的场景，需要恢复数据，这可怎么办。。。。

注意这个地方有点绕：

我设立了三个数据库初始化的开关，分别在服务级别，接口级别，用例级别

判断的策略是这样的，

开关和数据库初始化实例字典一同判断

1. 如果服务级别开关为打开，并且初始化字典为空，那么就初始化数据库同时把生成的保存有实例的字典传入到接口级别
2. 接口级别判断，如果初始化字典为空并且接口级别开关为打开，那么就初始化数据库，很显然，如果运行级别在服务的话并且中的初始化开关为打开，那么势必会传入一个不为空的初始化字典，那么也就不会执行初始化了
3. 用例级别判断，如果初始化字典为空，并且用例中初始化开关为真，那么就初始化数据库

如此保证了不论从哪个级别运行，数据库都能够正确的初始化，并且最少次数的初始化

数据库初始化之后，就可以对数据库进行操作了，那么 python 封装的数据库自然有 python 的语法，

下面列举了操作数据库的方法

其中 mongo 中的 instance 代表了命令行中的 db

redis 中的 instance 和 mongo 中的同样的效果

sql 中的 cur 代表游标, conn 代表链接实例

使用之前需要熟悉一下 python 操作数据库的一些基本命令在使用

```
1 description: "--自动化授权回调, --正常数据"
2 #自动化授权回调, 会将当前的udid自动的写入到当前customercode关联的第一个规则组中去,
3 #无论是黑是白, 相当于在数据库中生成数据, 要想数据在mongo中真的生效, 还是要调用授权二次
4 #确认接口
5 customerCode: wpk_test #该组关联的是黑名单
6 ruleId: ""
7 udid: wpk_call_back01
8 signature: normal
9 script_init_flag: True
10 mongo_command: empty
11 redis_command: empty
12 sql_command: #删除回调之后数据库生成的数据, 注意回调的时候要调用commit才能生效
13     - cur.execute("delete from activate_rule_value where ruleValue = 'wpk_call_back01'")
14     #- cur.execute("select * from activate_rule_value where ruleValue = 'wpk_call_back01'")
15     #- cur.fetchmany(1)
16     - conn.commit()
17 success_str: dc_0000
18 sync_data_flag: False
19 times: "1"
20 image_file:
21     file:
```

```
1 userMail: "405273017@qq.com"
2 password: 96e79218965eb72c92a549dd5a330112
3 clientId: uc_py_test_2
4 subsystemId: "5"
5 timestamp: normal
6 signature: normal
7
8 script_init_flag: True
9 mongo_command:
10     - instance.passport.update({'userName': 'zhenzhen'}, {'$set': {'userMail': ''}})
11     - instance.user_mail.remove({'_id': '405273017@qq.com'})
12     - instance.passport.remove({'userMail': '405273017@qq.com'})
13 redis_command: empty
14 sql_command: empty
15 success_str: uc_0000
16 sync_data_flag: False
17 times: "1"
18 image_file:
19     file: E:/wpk.png
20
```

由于线上环境, 的数据库我们不能操作, 所以请保证线上数据中的数据库初始化开关的关闭状态

另外, 由于网络原因, 在本地无法连接到线上测试环境的数据库, 所以在执行之前, 先把脚本拷贝到和线上测试环境网络通畅的测试机上

工具的 log:

级别

目前 log 级别不是很正规分为了 3 个级别

设置位置 user_center\config\global_config\global_config.yml

```
83 #log级别
84 # DEBUG 为打印请求的参数和url
85 # DETAIL 为打印每个以来请求的参数和url
86 log_level: DETA
87
88
```

空	填写任意值	只打印请求结果
DEBUG	精确匹配	会把请求接口的请求数据和结果全部打出
DETAIL	精确匹配	会把请求接口的和依赖接口的请求数据全部打出

结果保存

另外服务运行级别下的错误请求，全部都会记录在，`user_center\result\error_log.txt` 下

性能测试和压力测试

本工具能够针对服务下所有接口进行多进程的性能测试，性能和压力测试分为两种，

1. Python 结合 shell
2. Siege

在动态调度的基础上，参数动态生成，也就使测试者不用在关心接口依赖，数据依赖

也不用在花时间在为了测试某一个接口而在去写性能测试脚本了，只要执行本服务的性能测试入口脚本，就能进行测试

下面细说使用方法和原理

python 结合 shell 方式

先说一下在性能测试之前都需要准备什么。

执行环境

接口 url

请求方式

参数

时间

进程数

次数

统计结果

一个一个说：

执行环境：在全局配置文件中已经定义好了

接口 url：在接口的配置文件中已经定义好了

请求方式：在接口的配置文件中已经定义好了

参数：这个要说一下，先看图

数据位置: `user_center\test_data\bind_phone\pref\对应方法\对应文件`

从目录结构就能看出，

如果我要执行某一个 get 请求接口的内网环境的性能测试，那么我就需要修改

`user_center\test_data\bind_phone\pref\get\test_data_dev.yml`

如果我要执行某一个 post 请求接口的线上测试环境的性能测试，那么我就需要修改

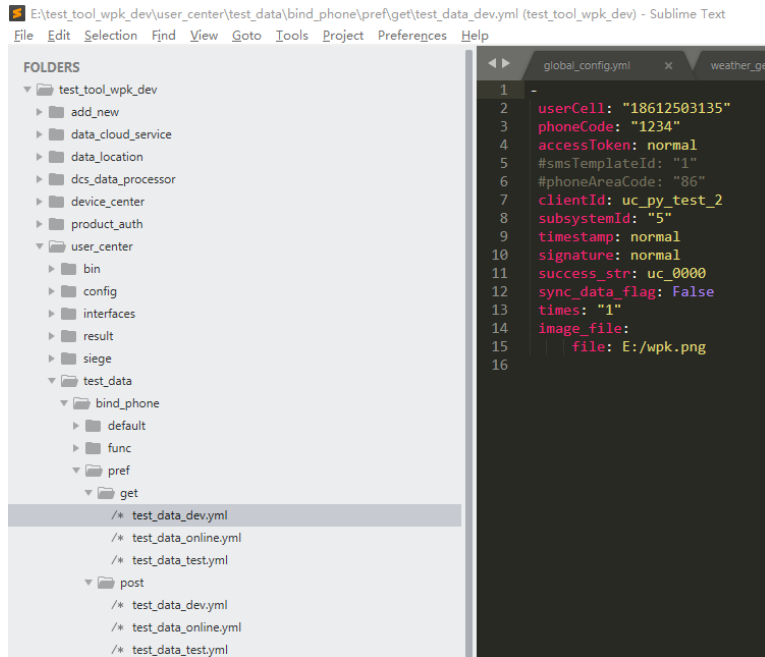
`user_center\test_data\bind_phone\pref\post\test_data_test.yml`

现在明白了执行的位置了，那么在看下数据结构

当前数据中也是用 - 分隔的，如果你传入了多个数据，那么工具会在创建进程的时候随机从多条数据中选择一条

如果传入一条，那么工具就只用当前数据创建所有进行

你可以发现，数据中依旧解决了接口依赖和数据依赖，在使用的时候完全可以像接口功能测试一样，愉快的使用 normal 和数据同步



时间，进程数，次数：放在一起说，其实具备了上面的几个条件，对一个接口请求的基本条件就全部具备了，这 3 个参数实际上就是在上面的基础上进行流程控制的

比如：

我要在当前执行环境下，以正常数据请求 get_flush_token 接口，启动 30 个进程，每个进程执行 5 分钟，执行 30 次

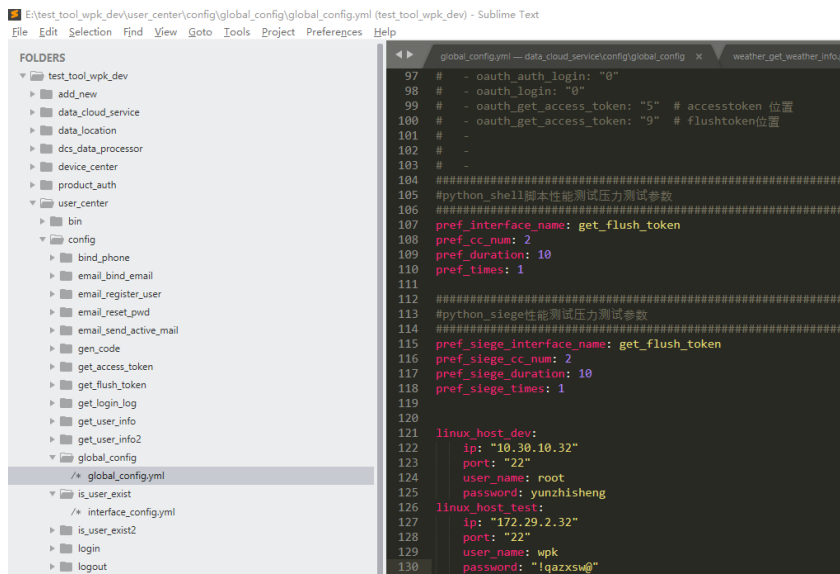
那这三个参数怎么设置呢

看图，在全局配置文件中，设置当前要跑的接口名字为 get_flush_token，设置进程数，设置执行时间，设置次数

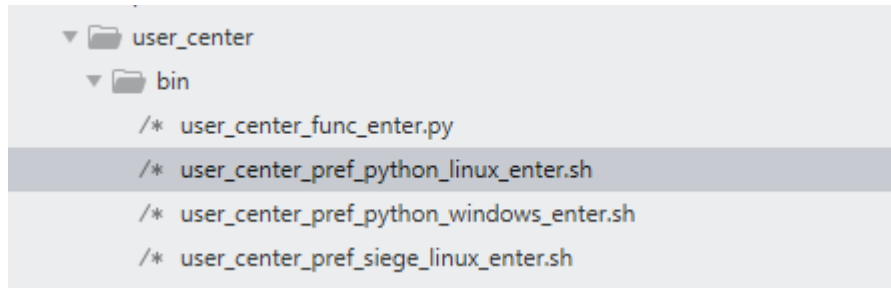
然后运行 user_center\bin\ user_center_pref_python_linux_enter.sh

细心的你应该发现了，是 shell 脚本，没错，因为 windows 下的 shell 无法触发多进程，所以，在运行之前，要把脚本拷贝到 linux

测试机上，然后用 sh user_center\bin\ user_center_pref_python_linux_enter.sh 的方式运行，工具就会按照当前的方式运行



服务的 bin 目录下提供两个环境的运行脚本，分别对应，linux 和 window 环境，其中 windows 主要是用 git_bash 在本地调试脚本使用的



而真正的性能测试需要把脚本拷贝到 linux 系统中去触发真正的多进程

运行方式

服务级别运行不加命令行参数

```
sh user_center_pref_python_windows_enter.sh
```

当前没有输入任何命令行参数，那它就会到全局配置文件中读取，要执行测试的接口名字，线程数，持续时间和次数

```
YZS@DESKTOP-MKAQ10B MINGW64 /e/test_tool_wpk_dev/user_center/bin
$ sh user_center_pref_python_windows_enter.sh
/e/test_tool_wpk_dev/user_center
no param input user config file data
thread 2 will run 10s
thread: 2: total: 169, pass: 169, TPS: 16.87, reps_time: 0.0593
thread 1 will run 10s
thread: 1: total: 168, pass: 168, TPS: 16.75, reps_time: 0.0597
-----The 0 round-----
CC: 24
TPS: 680.29
Reps_Time: 0.0679(s)
Current_Time:2019-05-22 14:04:57
```

服务级别运行加命令行参数

```
sh user_center_pref_python_windows_enter.sh 3 15 2 get_flush_token
```

看命令就知道了 可以输入命令行参数自定义，4个参数的含义分别是

接口名字， 线程数， 持续时间 次数

```
$ sh user_center_pref_python_windows_enter.sh get_flush_token 3 15 2
interface_name: get_flush_token
cc_num: 3
times: 2
duration: 15
thread 2 will run 15s
thread: 2: total: 260, pass: 260, TPS: 17.33, reps_time: 0.0577
thread 1 will run 15s
thread: 1: total: 259, pass: 259, TPS: 17.26, reps_time: 0.0579
thread 3 will run 15s
thread: 3: total: 259, pass: 259, TPS: 17.24, reps_time: 0.0580
-----The 0 round-----
CC: 3
TPS: 51.83
Reps_Time: 0.0579(s)
Current_Time:2019-05-22 14:33:42
```

如果忘记了命令的输入参数怎么办

随便输入一个参数，然后它就会告诉你 应该输入什么，以什么顺序

```
YZS@DESKTOP-MKAQ10B MINGW64 /e/test_tool_wpk_dev/user_center/bin
$ sh user_center_pref_python_windows_enter.sh get_flush_token 3
echo param: pref_interface_name|cc_num|duration|times|
```


python 脚本直接运行

还有一种运行方式，那就是直接运行 user_center\interfaces\get_flush_token\interface_pref_test\get_flush_token_pref_test.py

就相当于创建了一个单进程的，执行时间为全局配置文件中执行时间的程序

看下结果 当然这个执行文件的存在并不是为了真正的性能测试，而是验证一下，性能测试的脚本是否通，如果这个通了，那么服务级别的调用的就是它，那一定也没问题了，这个也在一定程度上降低了程序调试的成本。

```
thread 1 will run 10s

thread: 1: total: 183, pass: 183, TPS: 18.22, reps_time: 0.0549

[Finished in 10.3s]
```

Python 命令行

Python 命令行 也就是 python 加脚本路径的方式，因为这个方式和 3 的方式是相同的，但是在 linux 中调试程序的时候可能会借助这种方法，在命令行运行的时候可以输入两个参数，也可以不输入，两个参数为，持续时间和进程号，效果是一样的

```
YZS@DESKTOP-MKAQ10B MINGW64 /e/test_tool_wpk_dev/user_center/interfaces/get_flush_tok
en/interface_pref_test
$ python get_flush_token_pref_test.py 10 1
thread 1 will run 10s

thread: 1: total: 179, pass: 179, TPS: 17.88, reps_time: 0.0559
```

综上所述你已经掌握了 python 结合 shell 的性能以及压力测试工具的使用，需要注意的是

1. Python 和 shell 的方式可能会单节点无法灌满服务器，所以做好是多节点请求
2. 它有它的好处，它能够自定义判断结果，也就是能够统计请求过程中的成功和失败的场景
3. 它可以做稳定性测试，通过请求时间和请求次数进行配合，完成长时间的稳定性测试
4. 建议在测试的时候打开 nmon 进行监控，以便更好的分析测试结果，后续会将 nmon 封装到本工具中

Siege 工具

为了弥补 python+shell 工具在性能测试上单节点表现不足的情况，本工具封装了 siege 工具，进行快速请求，单节点灌满不是梦

注意 siege 只是针对短时间的性能测试，无法进行长时间的稳定性测试

讲解 siege

从核心代码介绍会比较简单

```
##=====
#传入，性能测试脚本你路径，性能脚本名字， 运行时间，当前编号
if [ $request_method == "get" ];then
    for((j=0;j<${pref_times};j++));do
        siege -c ${pref_cc_num} -t ${pref_duration}S "$url">/dev/null
    done

elif [ $request_method == "post" ];then
    for((j=0;j<${pref_times};j++));do
        siege -c ${pref_cc_num} -t ${pref_duration}S "$url POST $params">/dev/null
    done

else
    echo "method wrong"
    exit 1
fi
```

看下 siege 命令的发送格式就知道，我们需要拿到的东西实际上就是

请求方法，请求次数，进程个数，持续时间，url 和参数

我们的工具能干啥，我们的工具专门就是动态生成参数和 url 的，这对于我们的框架来说太简单了。

终于写道最后一个接口的可执行文件了，它就是

user_center\interfaces\get_flush_token\interface_siege_test\get_flush_token_siege_test.py

运行方式:

接口可执行文件运行，也就是执行

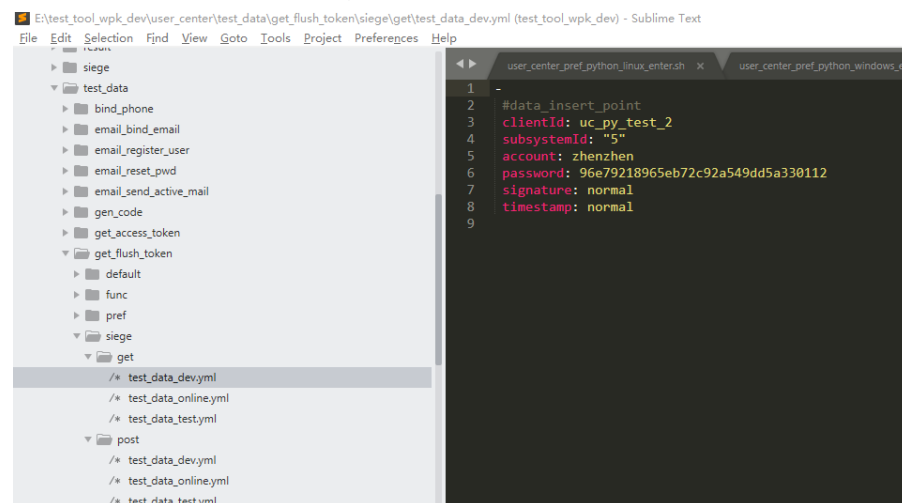
user_center\interfaces\get_flush_token\interface_siege_test\get_flush_token_siege_test.py

非常简单，它不做任何请求，他就是根据当前的数据文件生成了请求参数和 url

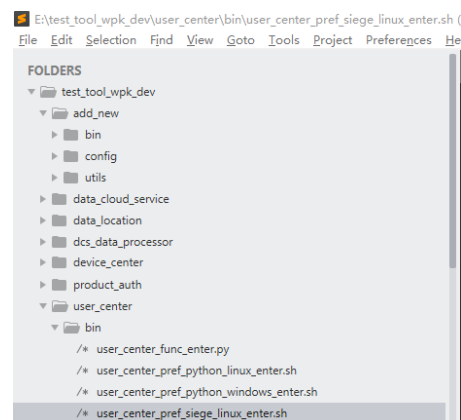
嗯 数据文件还得介绍一下，看下图 2 把

```
request_method: post
params: [{"account": "zhenzhen", "timestamp": "1558508604", "clientId": "uc_py_test_2", "signature": "B79D02897964FFD4688FA4F42D52A6FE6D86A28A", "subsystemId": "5", "password": "96e79218965eb72c92a549dd5a330112"}]
url: http://10.30.10.32:18080/rest/v2/user/login?
[Finished in 0.3s]
```

一看到这个我想你应该明白了，它的数据和 pref 的是完全一样的，找对应方法的对应的执行环境就可以了，这里就不在继续说了



那要真正的执行 siege 到底该怎么办呢——见下



user_center\bin\user_center_pref_siege_linux_enter.sh

看到 sh 就知道了，这又是个 shell 脚本，

必须要到 linux 上去使用，因为 git_bash 没有 siege 工具。。。。。

下面是真正的运行方式：

Linux 下入口运行 get_flush_token 从配置文件读取数据命令格式

sh user_center_pref_siege_linux_enter.sh

```
root@sh-uat-tester-2-32 bin# sh user_center_pref_siege_linux_enter.sh
loaded plugins: fastestmirror, langpacks
Repository epel is listed more than once in the configuration
loading mirror speeds from cached hostfile
Package dos2unix-6.0.3-7.el7.x86_64 already installed and latest version
Nothing to do
Install dos2unix success
dos2unix: converting file ./config/global_config/global_config.yml to Unix format ...
dos2unix: converting file ./config/global_config/global_config.yml to Unix format ...
dos2unix: converting file ./config/get_flush_token/interface_config.yml to Unix format ...
dos2unix: converting file ./config/get_flush_token/interface_config.yml to Unix format ...
dos2unix: modifying get_flush_token success
no param input user config file data
request_method:--post
params:--{'account': 'pukunwen', 'timestamp': '1558512073', 'clientId': 'uc_py_test_2', 'signature': '6BE8B775083835F745EC428A5778C1B6D130007B', 'subsystemId': '5', 'password': 'e10adc3949ba59abbe56e057f20f883e'}
url:--http://172.20.2.24:8080/rest/v2/user/login?
dos2unix: converting file ./siege/siege_data.txt to Unix format ...
dos2unix: modifying siege_data.txt success
=====
url read success
params read success
request_method read success
=====
--url is:http://172.20.2.24:8080/rest/v2/user/login?
--param is:{"account": "pukunwen", "timestamp": "1558512073", "clientId": "uc_py_test_2", "signature": "6BE8B775083835F745EC428A5778C1B6D130007B", "subsystemId": "5", "password": "e10adc3949ba59abbe56e057f20f883e"}
--request_method is:post
--pref times is:1
--cc_num is:2
--duration is:10
--pref_siege_interface_name is:get_flush_token
=====
[alert] Zip encoding disabled; siege requires zlib support to enable it
** SIEGE 4.0.4
** Preparing 2 concurrent users for battle.
The server is now under siege...
Lifting the server siege...
Transactions:      15844 hits
Availability:      100.00 %
Elapsed time:      9.31 secs
Data transferred:  0.00 MB
Response time:     0.00 secs
Transaction rate:  1701.83 trans/sec
Throughput:        0.00 MB/sec
Concurrency:       1.88
Successful transactions: 15845
Failed transactions: 0
Longest transaction: 0.02
Shortest transaction: 0.00
```

sh user_center_pref_siege_linux_enter.sh get flush token 20 20 1

```
root@sh-uat-tester-2-32 bin#
root@sh-uat-tester-2-32 bin# sh user_center_pref_siege_linux_enter.sh get_flush_token 20 20 1
loaded plugins: fastestmirror, langpacks
Repository epel is listed more than once in the configuration
loading mirror speeds from cached hostfile
Package dos2unix-6.0.3-7.el7.x86_64 already installed and latest version
Nothing to do
Install dos2unix success
dos2unix: converting file ./config/global_config/global_config.yml to Unix format ...
dos2unix: converting file ./config/global_config/global_config.yml to Unix format ...
dos2unix: converting file ./config/get_flush_token/interface_config.yml to Unix format ...
dos2unix: converting file ./config/get_flush_token/interface_config.yml to Unix format ...
dos2unix: modifying get_flush_token success
request_method:--post
params:--{'account': 'pukunwen', 'timestamp': '1558512194', 'clientId': 'uc_py_test_2', 'signature': 'AD4832BF7CFCB352205A7A978C0310E470A28B7', 'subsystemId': '5', 'password': 'e10adc3949ba59abbe56e057f20f883e'}
url:--http://172.20.2.24:8080/rest/v2/user/login?
dos2unix: converting file ./siege/siege_data.txt to Unix format ...
dos2unix: modifying siege_data.txt success
=====
url read success
params read success
request_method read success
=====
--url is:http://172.20.2.24:8080/rest/v2/user/login?
--param is:{"account": "pukunwen", "timestamp": "1558512194", "clientId": "uc_py_test_2", "signature": "AD4832BF7CFCB352205A7A978C0310E470A28B7", "subsystemId": "5", "password": "e10adc3949ba59abbe56e057f20f883e"}
--request_method is:post
--pref times is:1
--cc_num is:20
--duration is:20
--pref_siege_interface_name is:get_flush_token
=====
[alert] Zip encoding disabled; siege requires zlib support to enable it
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
Lifting the server siege...
Transactions:      56112 hits
Availability:      100.00 %
Elapsed time:      19.72 secs
Data transferred:  0.00 MB
Response time:     0.01 secs
Transaction rate:  2845.44 trans/sec
Throughput:        0.00 MB/sec
Concurrency:       19.77
Successful transactions: 56112
Failed transactions: 0
Longest transaction: 0.07
Shortest transaction: 0.00
root@sh-uat-tester-2-32 bin# █
```

Siege 的请求速度是非常快的，进程数量上去时候，单节点灌满一个服务，完全没有问题

但是 siege 不能定制的去判断返回的结果，这是它的缺点

代码内部逻辑：

关于代码的内部逻辑本文档暂时没有涉及，后续在补充吧