# 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，
   1. 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，
   2. 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在使用了
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

存在依赖关心，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

1. 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，那么将是无休无止的修改工作。。。。
2. 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evel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：
   1. 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

* 1. 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

* 1. 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

* 1. 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

1. 日志级别，正常，DEBUG, DETAIL，目前还不是很正规，后续会更新
2. 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！！！**

1. 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
2. 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

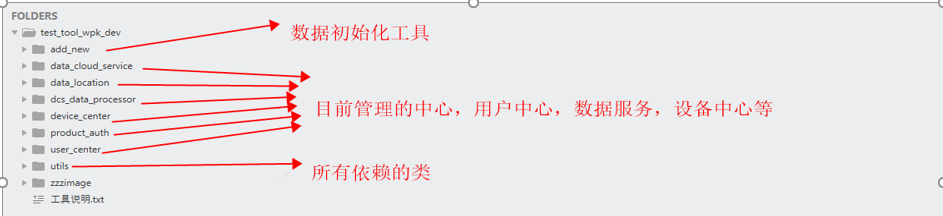
说了这么多，下面就用例子的方式，把上面的特点都演示一遍~

# 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. 目前管理的服务，如果后续管理的服务越来越多，会将这些服务放到统一的目录中
3. utils 所有公共抽离的方法，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



===============================================================================================

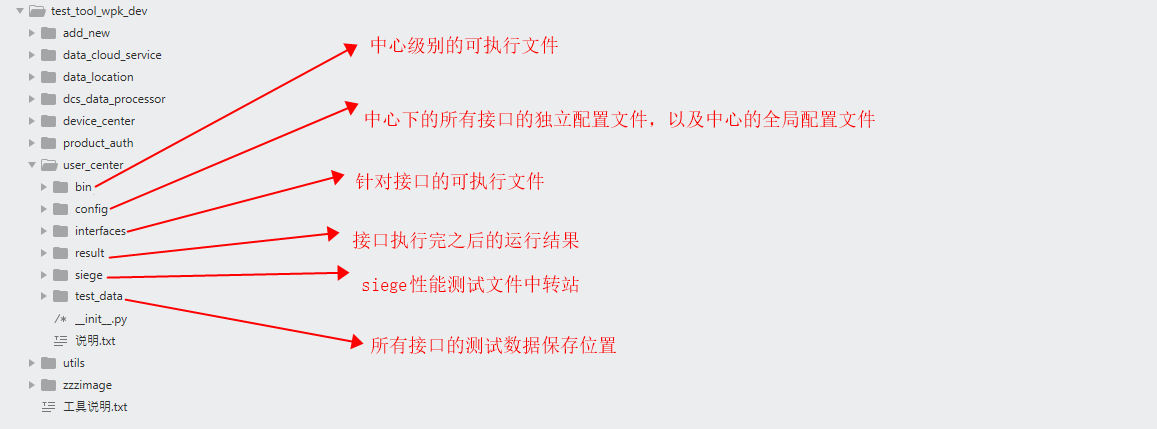
所有中服务的目录结构都是一样，包括配置文件，数据文件，和代码逻辑全部都一样，下面用用户服务来举例说明一下

**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

1. 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
2. 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
3. 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，
4. siege是性能测试的中转文件夹，使用者不用关心
5. 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

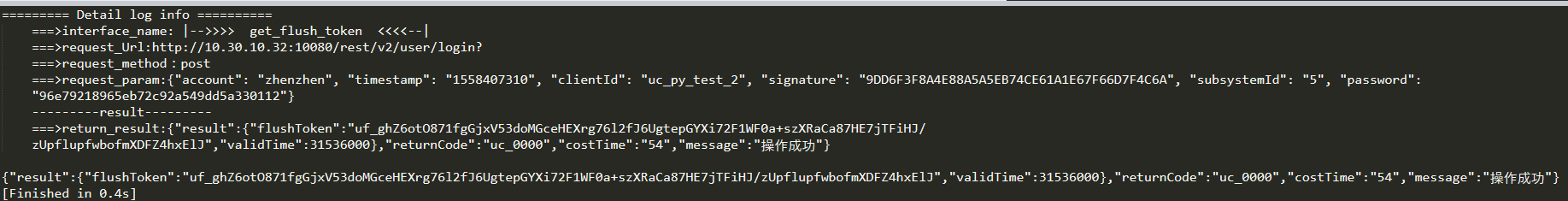
工具执行文件 user\_center\interfaces\get\_flush\_token\interface\_for\_call\get\_flush\_token.py

================================================================================

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flushtoken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。

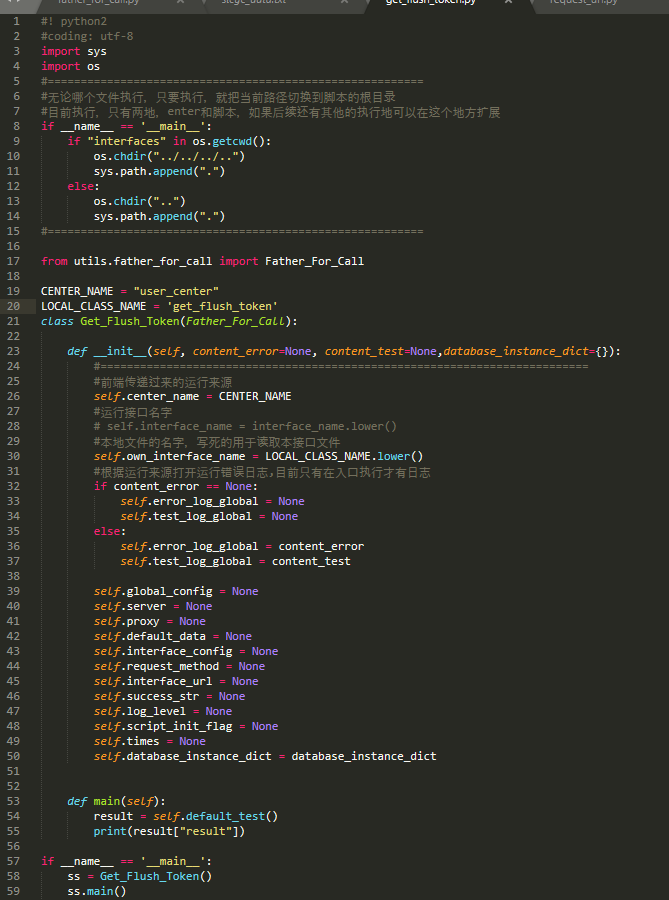
当前场景，我只要运行，工具执行文件，工具就能自动的帮我执行完成，先看下效果



**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。



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

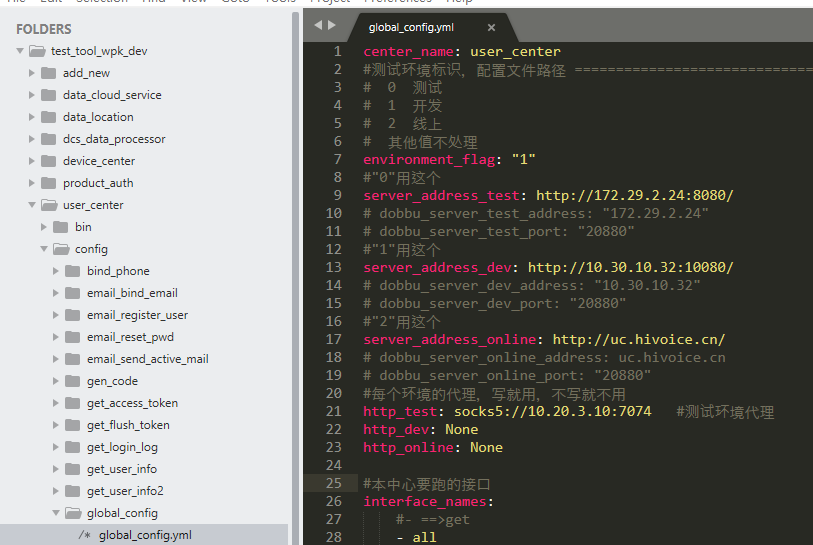
这个参数就已经确定了，所以把它放在了服务的全局配置文件中。

**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，如果我想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



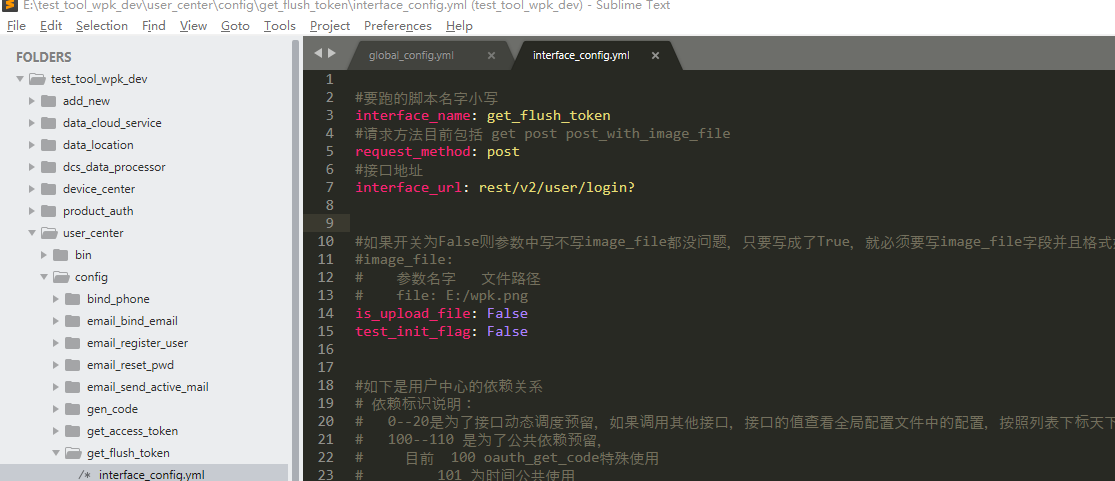
**确定请求方法，接口url、**

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



**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

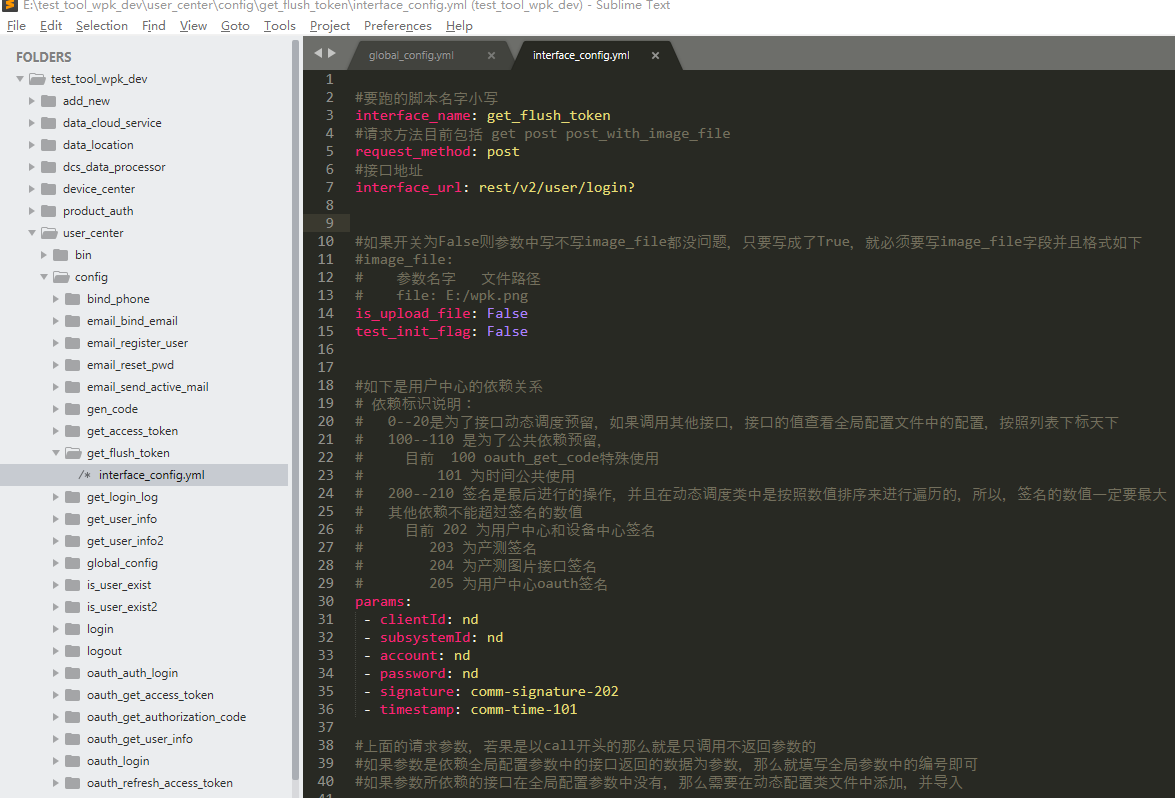
格式是固定的，前面的空格和 – 不能省略，后面跟参数名字，如果该参数不依赖任何东西，就直接写nd，not-dependency的意思

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



**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，后面会慢慢的介绍，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\_conmand: 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

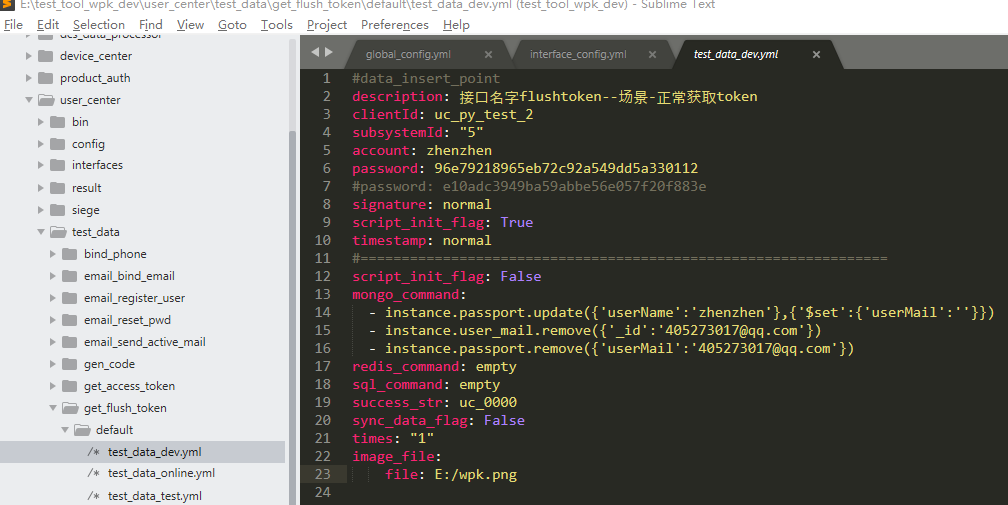
结果，也就是通过

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

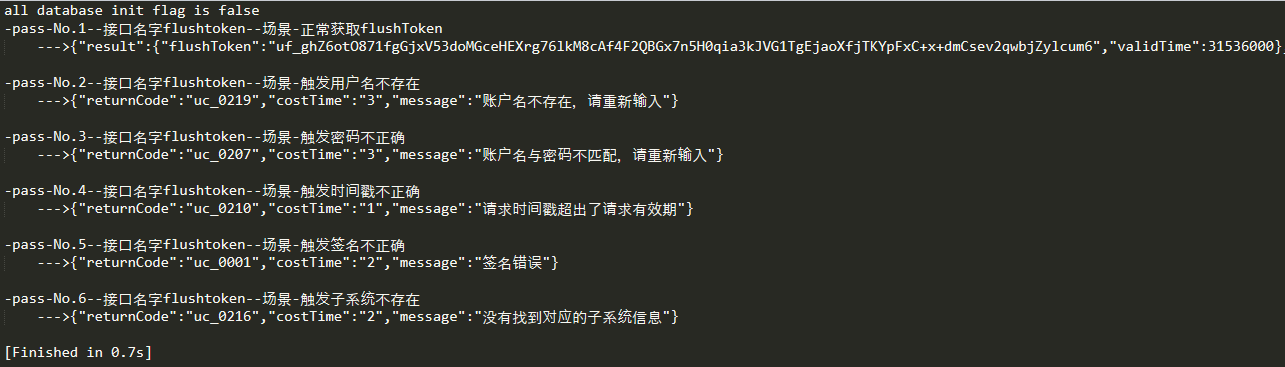
工具执行文件 user\_center\interfaces\get\_flush\_token\interface\_func\_test\get\_flush\_token\_func\_test.py

================================================================================

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

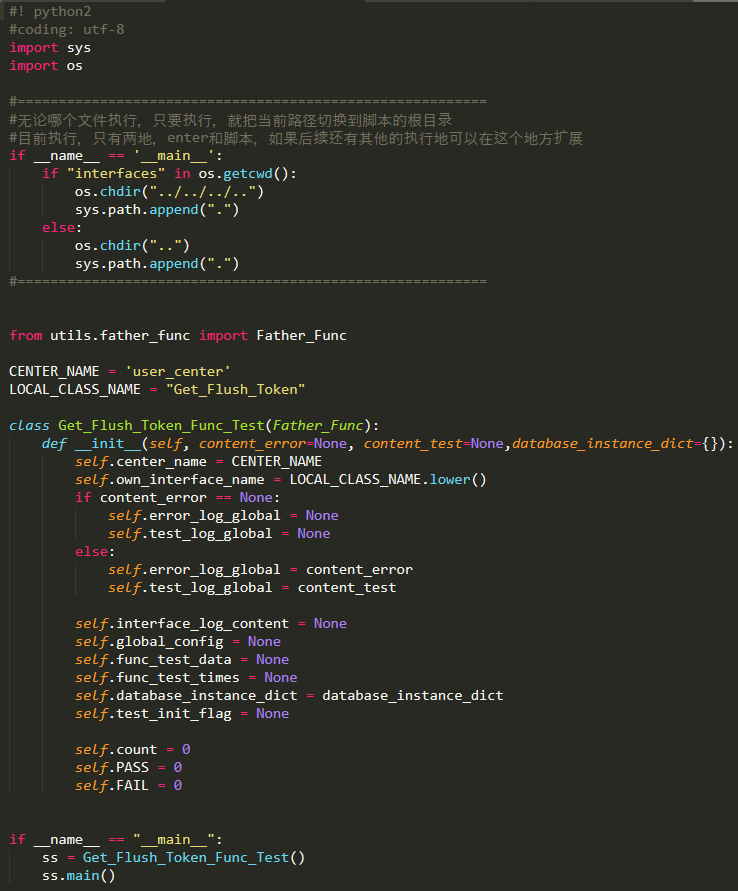


可以看到上面执行了6条用例。分别触发了不通的场景，并且测试全部都通过了，为啥异常场景也通过了，还记得全面说道的数据里面的

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



**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：

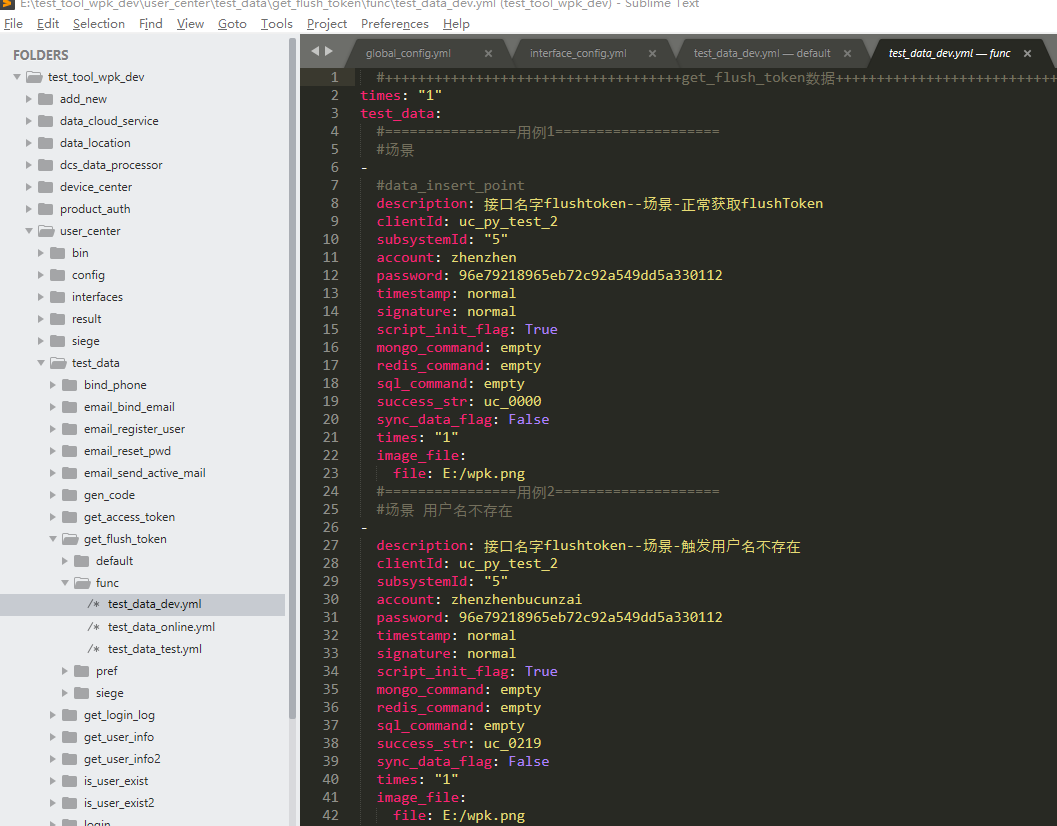
可以看到，当前文件的格式，在- 下面的数据结构，和default是一样的，也就是说，脚本会到当前文件读取所有的数据，形成一个 可迭代对象，然后对这个数据进行迭代，从而完成单接口多数据的场景测试

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，可以看第二条用例，为uc\_0219所以如果结果中有uc\_0219我也认为他通过了



**最后，如果你相触发更多的用例，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

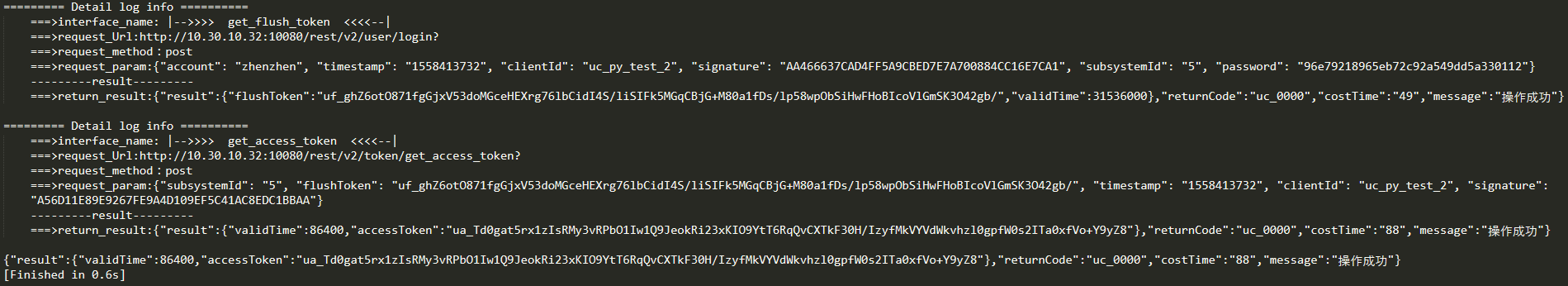
请求参数: flushToken signature timestamp clientId subsystemId

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



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



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

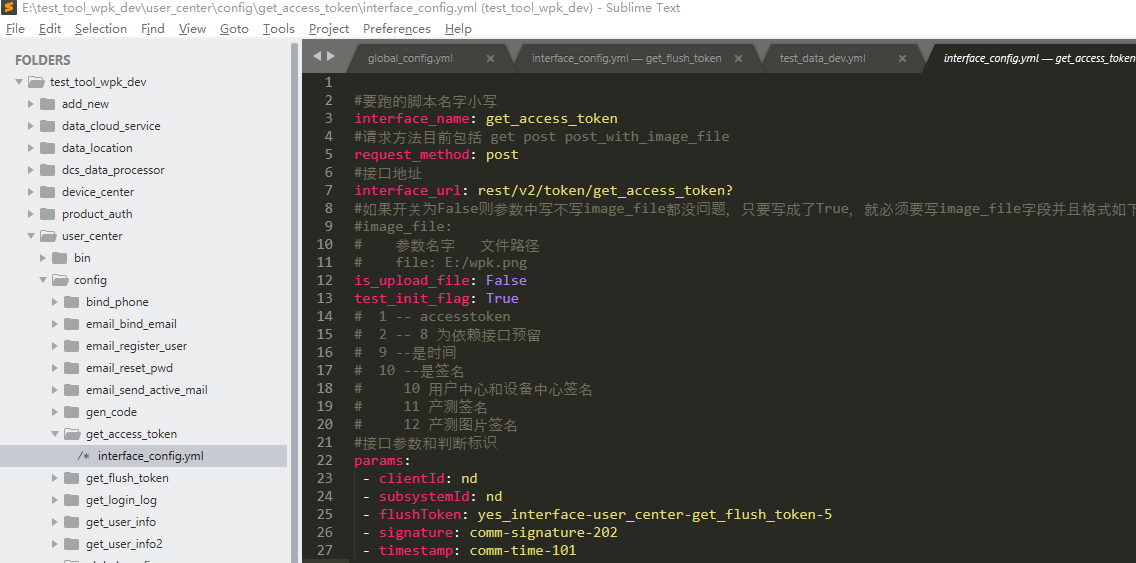
**注意**：这里要说明一点，可能当前的结构看起来不是很明确，可能有人会问为什么不用键值对的方式，去确定依赖关系，那样的话更

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

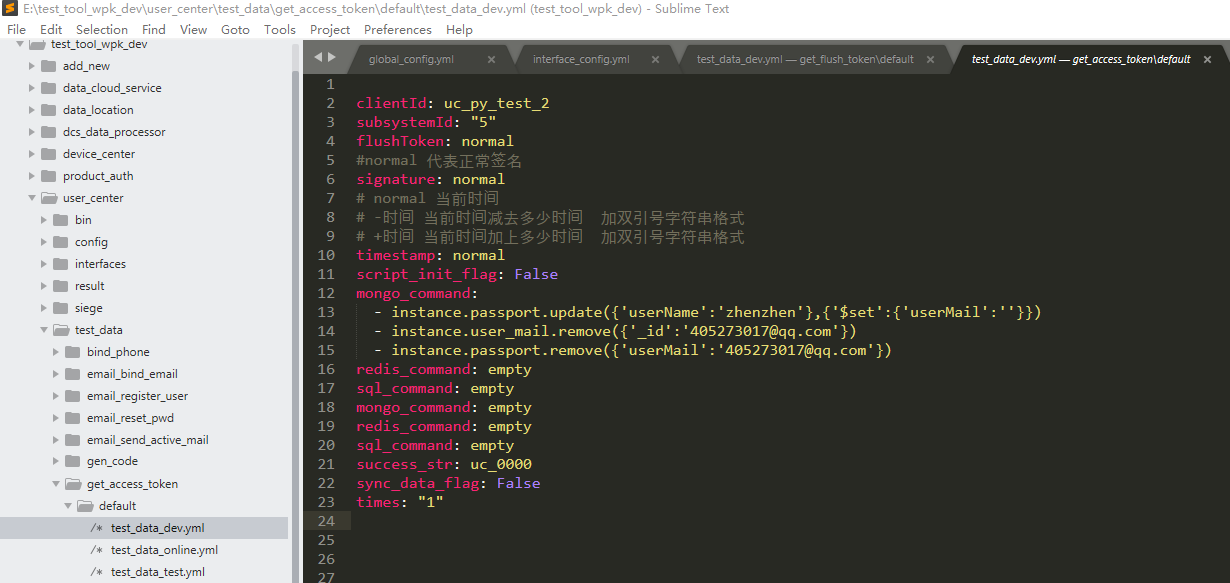
接口的返回值中，所获取的结果的位置或者深度都是不固定的，这样兼容性不是很好，以下标的方式，反而是兼容性更好的选择，因

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

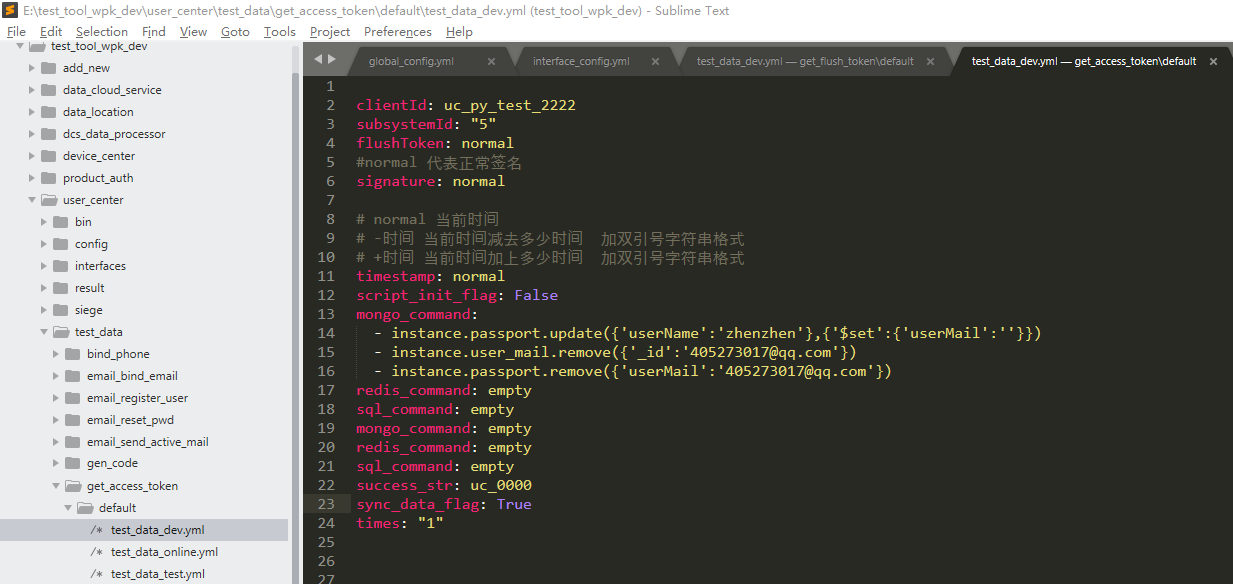
现在我想让accesstoken 在调用依赖flush\_token的时候clientId参数用我的给它定义的，uc\_py\_test\_2222，那我需要怎么做

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



## 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

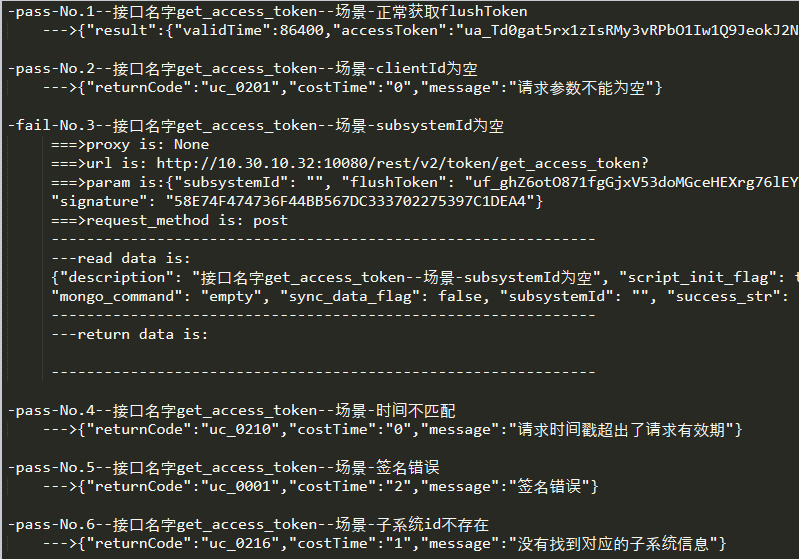
需要依赖获取的参数: signature timestamp flushToken

请求环境: 内网环境

工具执行文件 user\_center\interfaces\get\_access\_token\interface\_funcl\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，



**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**

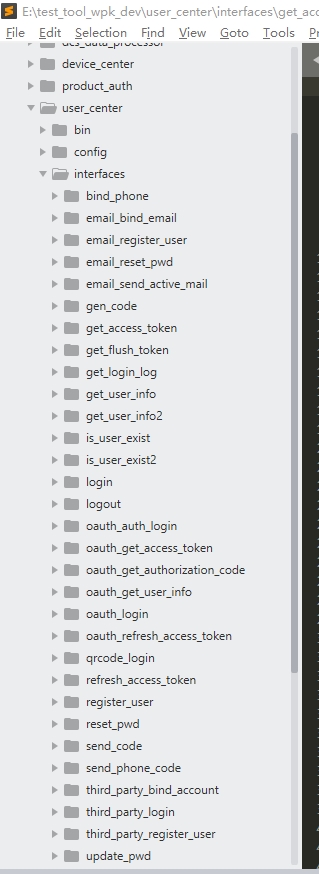
**user\_center\test\_data\ get\_access\_token\func\test\_data\_dev.yml**

## 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，是有关系的，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**

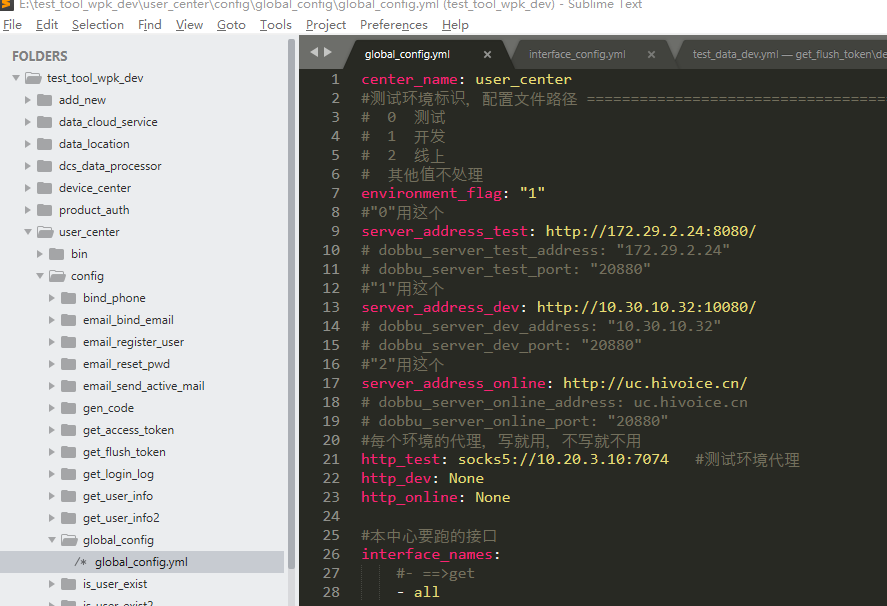
**我要跑所有接口**

**– 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，这里面的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

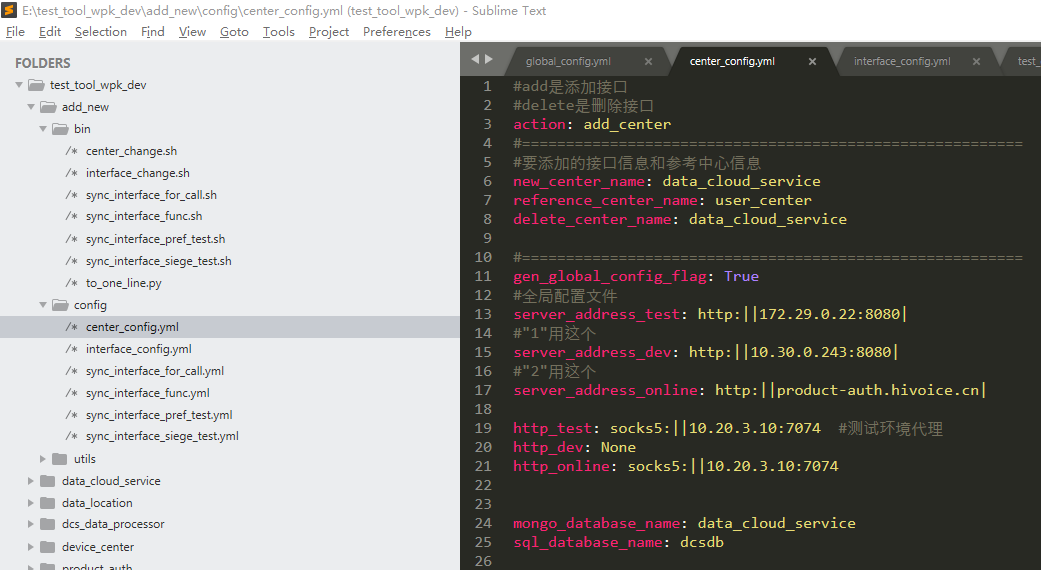


## 场景1 增加服务

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

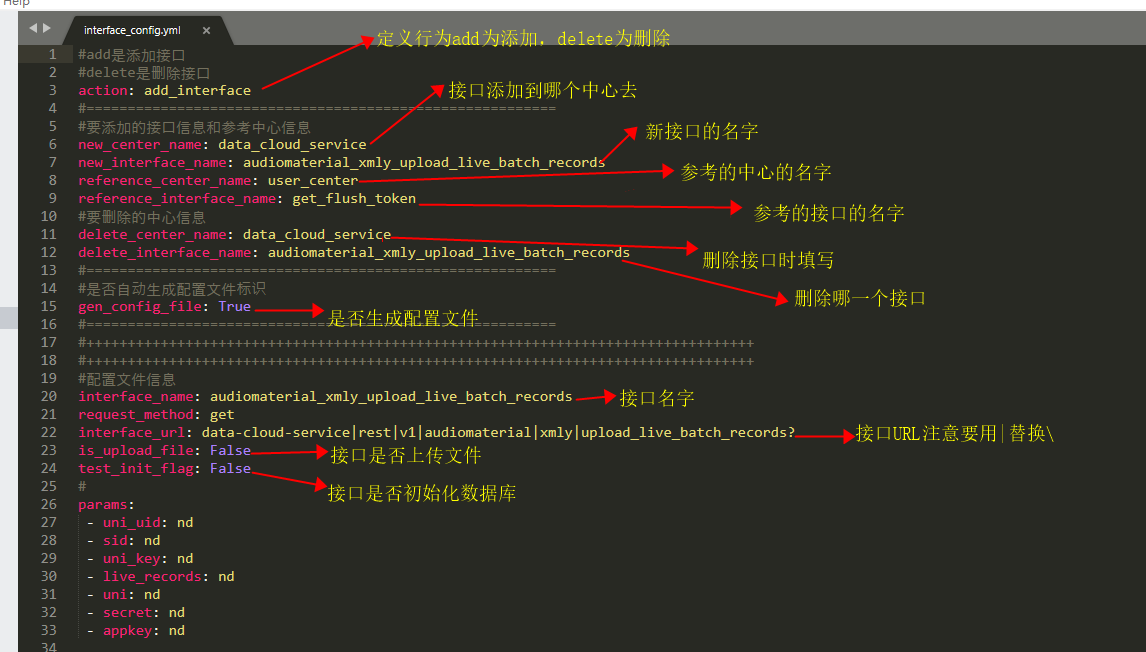
在delete\_center\_name: data\_cloud\_service 填写要删除的服务就可以

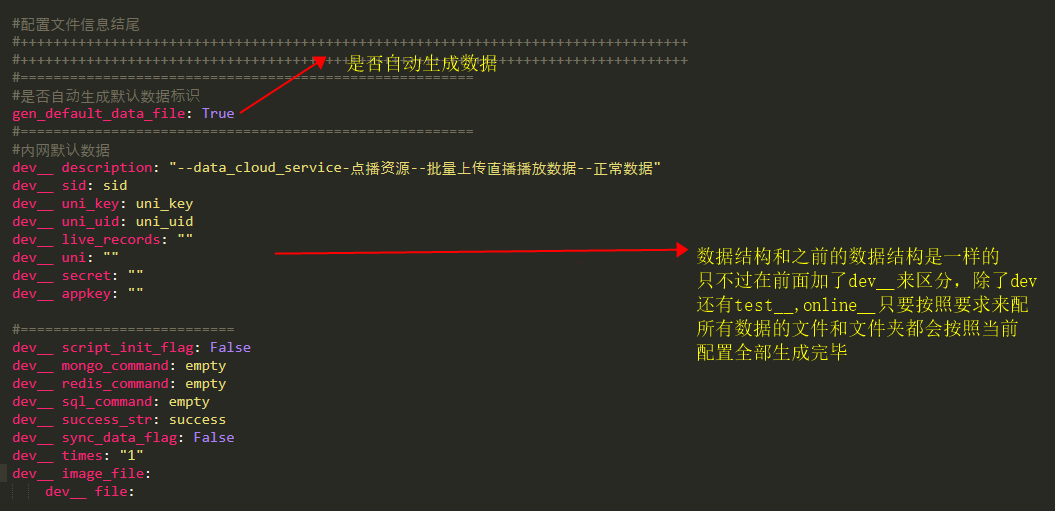
注意删除服务的时候，做好吧编辑器关闭掉，，否则python脚本可能会因为文件打开而删除不掉

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，





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

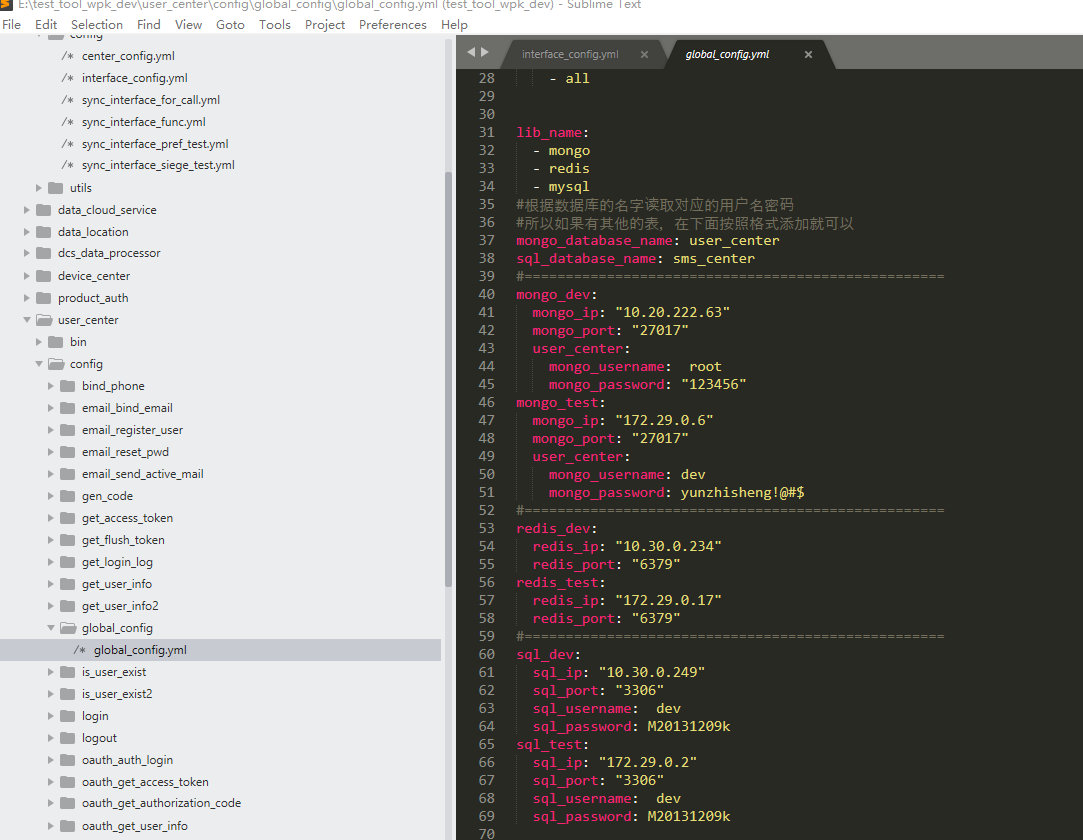
一个一个来，先看数据库在哪里

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，但是我不可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

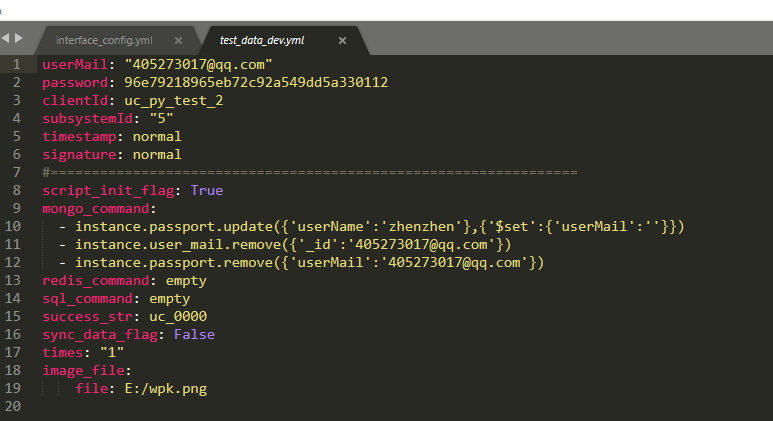
**其中mongo中的instance代表了命令行中的db**

**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**





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

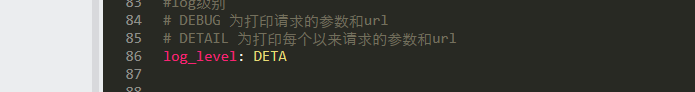
**另外，由于网路原因，在本地无法连接到线上测试环境的数据库，所以在执行之前，先把脚本拷贝到和线上测试环境网络通畅的测试机上**

# 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



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

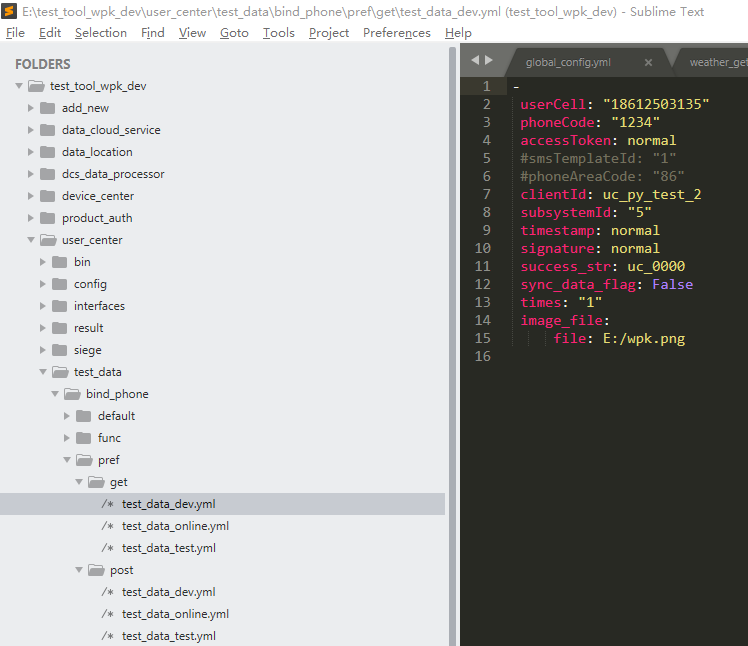
user\_center\test\_data\bind\_phone\pref\post\test\_data\_test.yml

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

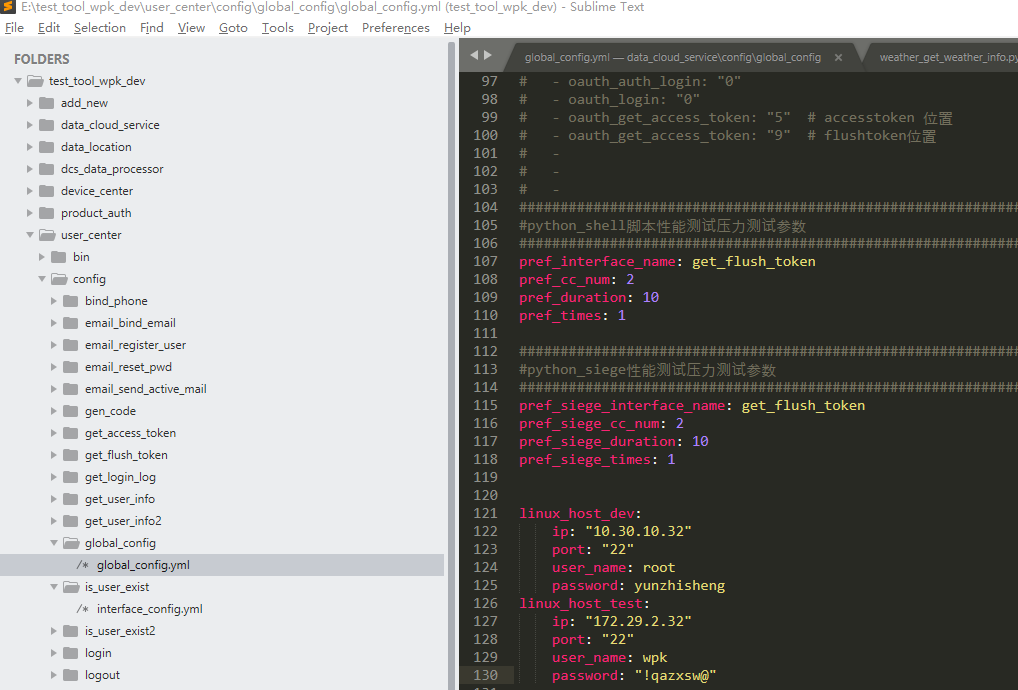
那这三个参数怎么设置呢

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

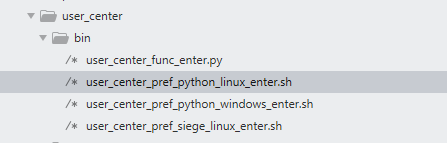
然后运行 user\_center\bin\ user\_center\_pref\_python\_linux\_enter.sh

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



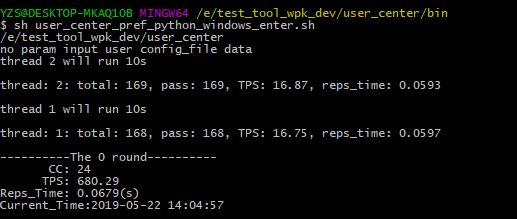
而真正的性能测试需要把脚本拷贝linux系统中去触发真正的多进程

### 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**

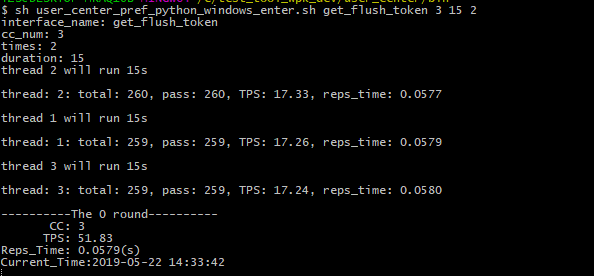


#### 服务级别运行加命令行参数

**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， 进程数， 持续时间 次数



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

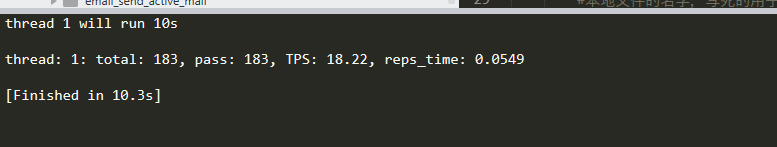


#### python脚本直接运行

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

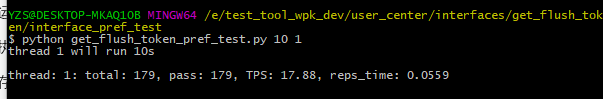
看下结果 当然这个执行文件的存在并不是为了真正的性能测试，而是验证一下，性能测试的脚本是否通，如果这个通了，那么服务级别的 调用的就是它，那一定也没问题了，这个也在一定程度上降低了程序调试的成本。



#### 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



**综上相信你已经掌握了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

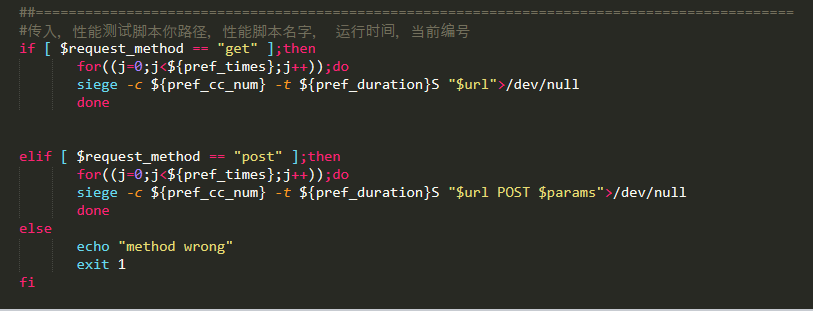
## Siege工具

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



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

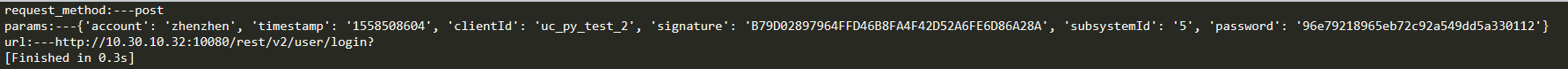
### 运行方式:

接口可执行文件运行，也就是执行

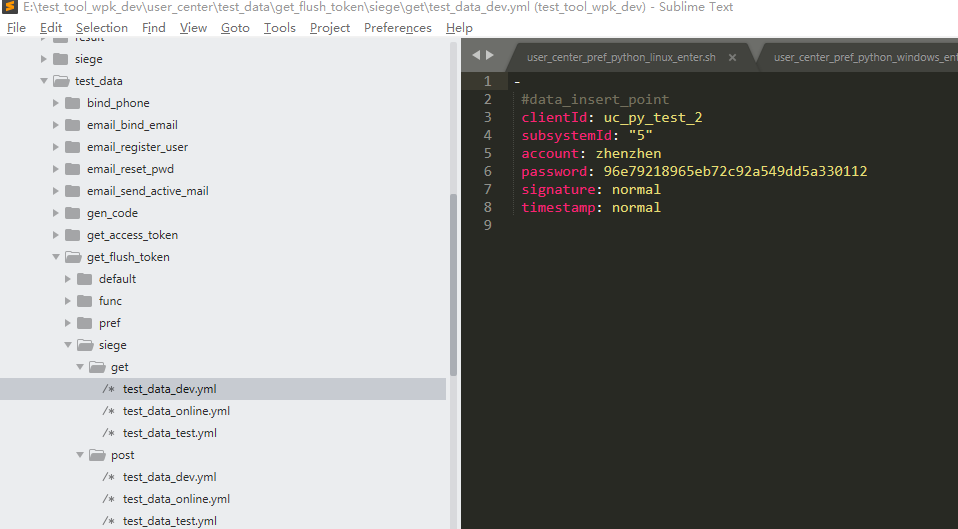
user\_center\interfaces\get\_flush\_token\interface\_siege\_test\get\_flush\_token\_siege\_test.py

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

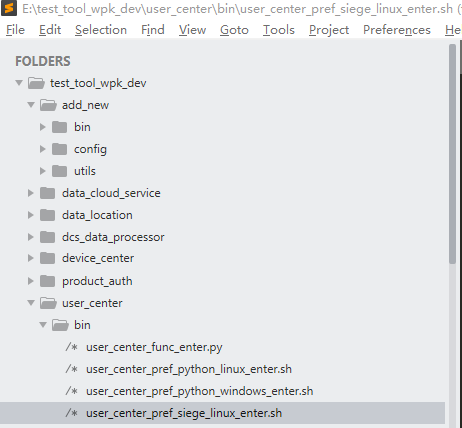
嗯 数据文件还得介绍一下，看下图2把



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

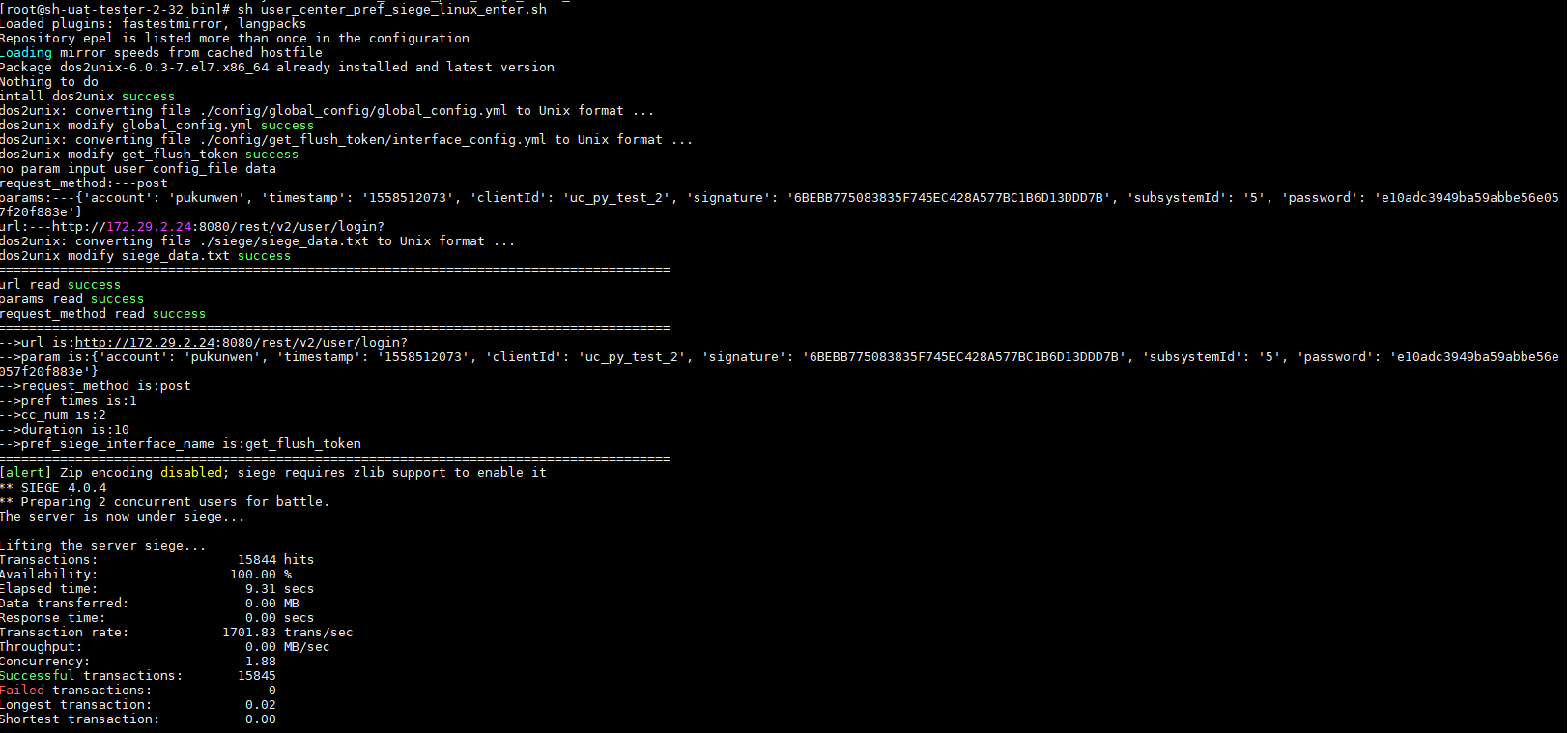
看到sh就知道了，这又是个shell脚本，

必须要到linux上去使用，因为git\_bash没有siege工具。。。。。

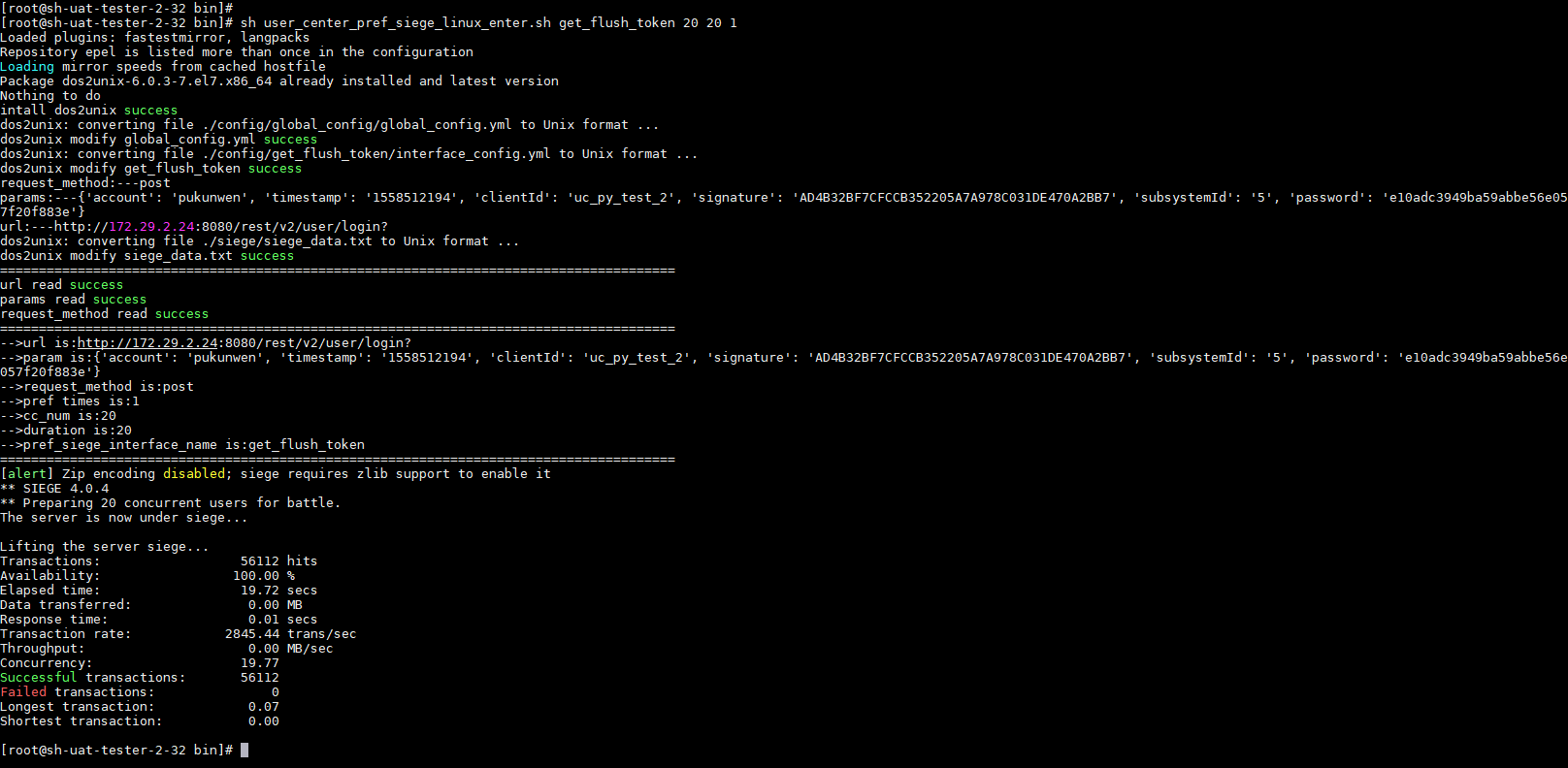
下面是真正的运行方式：

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



sh user\_center\_pref\_siege\_linux\_enter.sh get flush token 20 20 1



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