目录

[1 整体概览 1](#_Toc203385077)

[1.1 项目概述 1](#_Toc203385078)

[1.1.1 项目背景和目标 1](#_Toc203385079)

[1.1.2 项目特点 1](#_Toc203385080)

[1.2 项目总体架构设计 2](#_Toc203385081)

[1.2.1 架构图 2](#_Toc203385082)

[1.2.2 技术栈 3](#_Toc203385083)

[1.2.3 项目结构说明 3](#_Toc203385084)

[1.3 用例设计与数据驱动实现 4](#_Toc203385085)

[1.3.1 数据驱动设计思想 4](#_Toc203385086)

[1.3.2 Pytest 数据驱动实现 4](#_Toc203385087)

[1.3.3 用例分层思想 4](#_Toc203385088)

[1.4 接口请求封装与执行引擎 5](#_Toc203385089)

[1.4.1 请求模块设计 5](#_Toc203385090)

[1.4.2 依赖注入与参数替换机制 6](#_Toc203385091)

[1.5 响应断言与提取机制 8](#_Toc203385092)

[1.5.1 响应断言逻辑 8](#_Toc203385093)

[1.5.2 提取变量机制 9](#_Toc203385094)

[1.6 日志系统设计 12](#_Toc203385095)

[1.6.1 日志封装实现 12](#_Toc203385096)

[1.6.2 如何将日志挂载到 Allure 12](#_Toc203385097)

[1.7 Allure 报告集成与优化 12](#_Toc203385098)

[1.7.1 使用方式 12](#_Toc203385099)

[1.7.2 报告结构与内容说明 12](#_Toc203385100)

[1.8 用例依赖与上下文变量机制 12](#_Toc203385101)

[1.8.1 为什么需要依赖机制？ 12](#_Toc203385102)

[1.8.2 当前的实现方案 13](#_Toc203385103)

[2 从零搭建接口测试框架 13](#_Toc203385104)

[2.1 测试入口与用例提取 13](#_Toc203385105)

[2.1.1 实现 13](#_Toc203385106)

[2.2 用例执行主逻辑 14](#_Toc203385107)

[2.2.1 执行函数 15](#_Toc203385108)

[2.3 用例上下文管理 17](#_Toc203385109)

[2.3.1 上下文保存 17](#_Toc203385110)

[2.3.2 上下文提取 17](#_Toc203385111)

[2.4 接口执行 19](#_Toc203385112)

[2.5 测试断言 19](#_Toc203385113)

[2.5.1 断言主逻辑 20](#_Toc203385114)

[2.5.2 状态码断言 21](#_Toc203385115)

[2.5.3 空值处理 21](#_Toc203385116)

[2.5.4 包含断言 21](#_Toc203385117)

[2.5.5 相等与不相等断言 23](#_Toc203385118)

[2.5.6 数据库断言 25](#_Toc203385119)

[2.6 日志与异常处理 28](#_Toc203385120)

[2.6.1 日志记录 28](#_Toc203385121)

[2.6.2 异常处理 31](#_Toc203385122)

[2.7 Allure报告集成 32](#_Toc203385123)

[2.7.1 Allure+Pytest原理 32](#_Toc203385124)

[2.7.2 Allure报告级别 32](#_Toc203385125)

[2.7.3 Allure报告记录 33](#_Toc203385126)

[2.7.4 Allure报告展示 33](#_Toc203385127)

[2.8 配置文件全局管理 35](#_Toc203385128)

[2.8.1 能够抽离的配置 35](#_Toc203385129)

[2.8.2 全局配置管理 35](#_Toc203385130)

[2.9 工作流程的多接口用例执行 36](#_Toc203385131)

[2.9.1 yaml： 37](#_Toc203385132)

[2.9.2 入口函数： 38](#_Toc203385133)

[2.9.3 可能的优化 38](#_Toc203385134)

[2.10 conftest钩子优化架构 38](#_Toc203385135)

[2.10.1 pytest\_sessionstart 39](#_Toc203385136)

[2.10.2 pytest\_generate\_tests 39](#_Toc203385137)

[2.10.3 pytest\_runtest\_setup 40](#_Toc203385138)

[2.10.4 pytest\_sessionfinish与pytest\_terminal\_summary 42](#_Toc203385139)

[2.11 分批次运行与报告合并（未完全解决） 43](#_Toc203385140)

[2.11.1 分批次运行 43](#_Toc203385141)

[2.11.2 报告合并 43](#_Toc203385142)

[2.12 Mock骨架实现 44](#_Toc203385143)

[3 项目问答 47](#_Toc203385144)

[3.1 框架设计与模块解耦 47](#_Toc203385145)

[3.1.1 你为什么选择用 Pytest，而不是 unittest 或 nose？它对你的框架有什么优势？ 47](#_Toc203385146)

[3.1.2 你的框架中是如何实现测试数据与逻辑解耦的？具体通过什么方式加载 YAML？ 47](#_Toc203385147)

[3.1.3 请你详细说说 pytest\_generate\_tests 在你框架中的作用，它在哪个阶段生效？为什么选它？ 48](#_Toc203385148)

[3.1.4 你有没有实现前置条件复用，比如登录、鉴权？是用 fixture、钩子，还是代码封装实现的？ 48](#_Toc203385149)

[3.1.5 你的请求封装模块中是否支持自定义请求头、超时、重试机制？设计上如何避免耦合？ 48](#_Toc203385150)

[3.2 用例设计与执行控制 49](#_Toc203385151)

[3.2.1 如何通过配置或参数控制执行不同环境（dev/test/prod）下的接口？ 49](#_Toc203385152)

[3.2.2 框架中是否支持用例失败重跑？如果有，是用插件实现还是自定义逻辑？ 49](#_Toc203385153)

[3.2.3 如果一个用例依赖另一个接口返回的 token 或 ID，你是如何管理这些上下文变量的？ 49](#_Toc203385154)

[3.2.4 你有没有处理接口依赖问题，比如某接口必须先执行其他接口才能测？如何保证用例的独立性？ 49](#_Toc203385155)

[3.2.5 如何控制测试用例的并发或批量分组运行？有没有做过分批执行和报告合并？ 49](#_Toc203385156)

[3.2.6 你是如何对测试用例进行分层管理的？是否区分了核心用例、冒烟用例、回归用例？ 50](#_Toc203385157)

[3.2.7 有没有设计用例标签机制，比如用 @pytest.mark.xxx 实现筛选？ 50](#_Toc203385158)

[3.2.8 如果某个接口在执行中频繁抛错影响其他用例，你会如何隔离这类不稳定用例？ 50](#_Toc203385159)

[3.2.9 如果断言失败，但实际是字段顺序变化导致，如何设计容错机制？ 50](#_Toc203385160)

[3.3 日志与报告系统 51](#_Toc203385161)

[3.3.1 你是如何设计日志系统的？日志的输出格式、级别、保存策略具体是如何配置的？ 51](#_Toc203385162)

[3.3.2 你的日志是否能精确到每个用例？在 Allure 报告中是否能展示该用例独立日志？ 51](#_Toc203385163)

[3.3.3 Allure 中用到了哪些动态内容？你在哪些位置使用了 allure.dynamic.xxx()？ 51](#_Toc203385164)

[3.3.4 你有没有遇到 Allure 报告合并冲突或 UUID 重复的问题？是如何解决的？ 51](#_Toc203385165)

[3.3.5 你是否自定义过 Allure 的附件或截图功能？用在哪些场景？attach 的 MIME type 是什么？ 51](#_Toc203385166)

[3.4 配置与扩展能力 52](#_Toc203385167)

[3.4.1 框架是否支持多环境配置管理？配置项是写在哪，如何被读取并生效？ 52](#_Toc203385168)

[3.4.2 你是否对框架进行了封装，比如统一入口 run.py 或 runner？支持命令行参数吗？ 52](#_Toc203385169)

[3.4.3 框架是否支持插件扩展，如：pytest-rerunfailures、pytest-xdist？是否实际使用过？ 52](#_Toc203385170)

[3.4.4 你有没有为这个框架写过 CI/CD 脚本或考虑在 Jenkins/GitLab Runner 中运行？ 52](#_Toc203385171)

[3.4.5 如何设计框架结构让新手也能快速上手使用和写用例？有没有自动生成用例模版的功能？ 53](#_Toc203385172)

[3.4.6 你是否支持参数化依赖，例如从 Excel / CSV 批量导入测试数据？ 53](#_Toc203385173)

[3.5 健壮性与边界处理 53](#_Toc203385174)

[3.5.1 如果接口返回非常大的数据或乱码，你的框架能否稳定解析或打印？ 53](#_Toc203385175)

[3.5.2 你是如何处理 YAML 数据格式错误或关键字段缺失导致的异常？ 53](#_Toc203385176)

[3.5.3 你的框架是否做了异常捕获机制？是全局捕获还是每步 try/except？ 54](#_Toc203385177)

[3.5.4 对接口响应做断言时，是不是支持多种方式（键值匹配、子集、正则等）？ 54](#_Toc203385178)

[3.5.5 有没有处理过 HTTP 非200状态码？怎么判断请求失败与业务失败的区别？ 54](#_Toc203385179)

[3.6 进阶与优化方向 54](#_Toc203385180)

[3.6.1 如果接口是异步的，比如发出请求需要轮询状态，你是怎么设计等待和断言逻辑的？ 54](#_Toc203385181)

[3.6.2 你有使用过 JSON Schema 来断言接口返回格式吗？是否封装成通用断言函数？ 55](#_Toc203385182)

[3.6.3 你考虑过对框架增加 Swagger 自动化生成用例功能吗？ 55](#_Toc203385183)

[3.6.4 你是否对报告做过二次封装，如将报告上传、推送到通知渠道（钉钉、企业微信）？ 55](#_Toc203385184)

[3.6.5 你是否思考过如何将这个框架迁移为更通用的服务化测试平台？支持 Web 界面管理和触发？ 55](#_Toc203385185)

[4 总结 55](#_Toc203385186)

[4.1 不足与经验 55](#_Toc203385187)

[4.1.1 单接口与多接口用例未统一 55](#_Toc203385188)

[4.1.2 分批次运行与报告合并的优化 56](#_Toc203385189)

[4.1.3 数据源支持的丰富性 56](#_Toc203385190)

[4.1.4 CI/CD的拓展 56](#_Toc203385191)

[4.1.5 测试环境的切换 56](#_Toc203385192)

[4.1.6 ${}的完整处理 56](#_Toc203385193)

[4.2 优化畅想 56](#_Toc203385194)

[4.3 总结 56](#_Toc203385195)

# 整体概览

## 项目概述

### 项目背景和目标

本项目来源于个人学习时的一个练手项目，测试的项目是一个简要的一个电子商务的系统，该系统合计10个接口，该系统后端是一个基于Flask实现的简单服务器，可以当作一个测试的模拟环境，虽然数据的真实性和认证与授权的管理与实际可能有出入，但是并不影响整体的一个测试框架的搭建。

做这个项目的最主要目标其实就是学习搭建一个完整的测试框架来深入接口测试的知识，同时构造一个可用性、复用性强的接口测试框架，利于以后的复用与拓展。

### 项目特点

* **数据驱动**

通过yaml存储外部数据，规范化、格式化存储用例，不依赖paramize而是使用pytest内部generate\_test进行用例的参数化注入，更加灵活自由进行参数化数据驱动。对于用例的外部数据和上下文依赖，通过动态的${}函数替换和extract.yaml上下文管理文件实现。整体项目能够单独抽离，高内聚低耦合，复用性强。

* **用例管理**

用例通过外部数据读取，调用内部generate\_tests钩子注入参数，对于前置条件依赖，通过yaml数据preconditions字段+ runtest\_setup钩子独立处理，不同用例前置条件单独执行，保证用例间的独立性，避免用例间的相互依赖。而前置条件上下文通过extract.yaml文件进行传递。

用例执行、断言通过统一入口，维护成本低，接口请求通过request执行，响应断言支持多种断言方式。此外用例执行提供分批次运行和工作流的复杂执行。

* **日志收集与异常记录**

异常只收集、不拦截，提供详细的日志记录，便于检查缺陷、系统错误。

* **Allure报告**

Allure全面记录用例信息，实现用例的模块划分，基于allure实现美观的测试报告查看。

* **拓展能力**

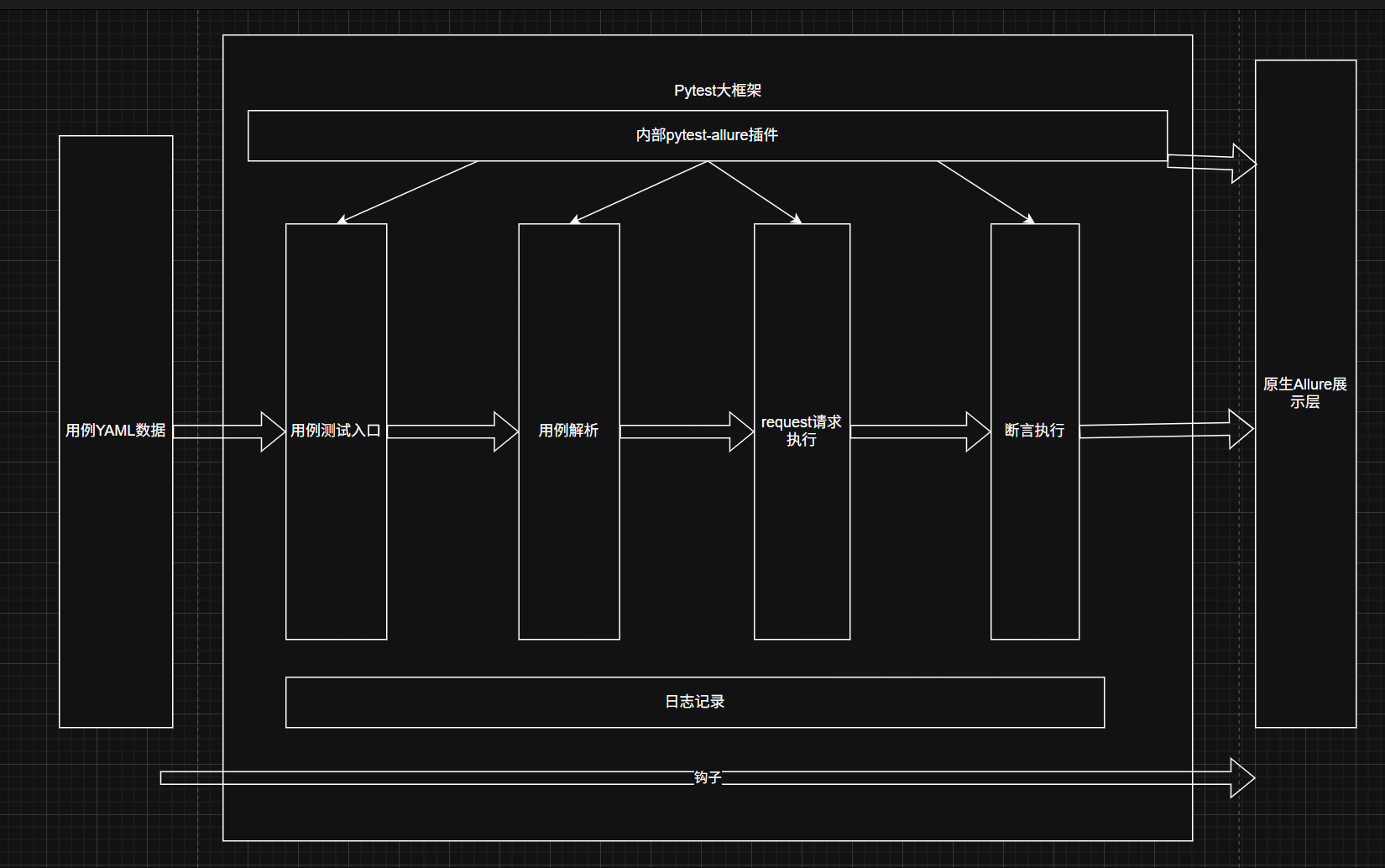
提供Mock、前端加密模拟、分批次执行等功能的拓展，断言部分也提供更加丰富的数据库断言选择，目前仅有mysql，而redis、clickhouse等都可进行拓展。此外邮件发送、钉钉通知等都留下拓展入口。

## 项目总体架构设计

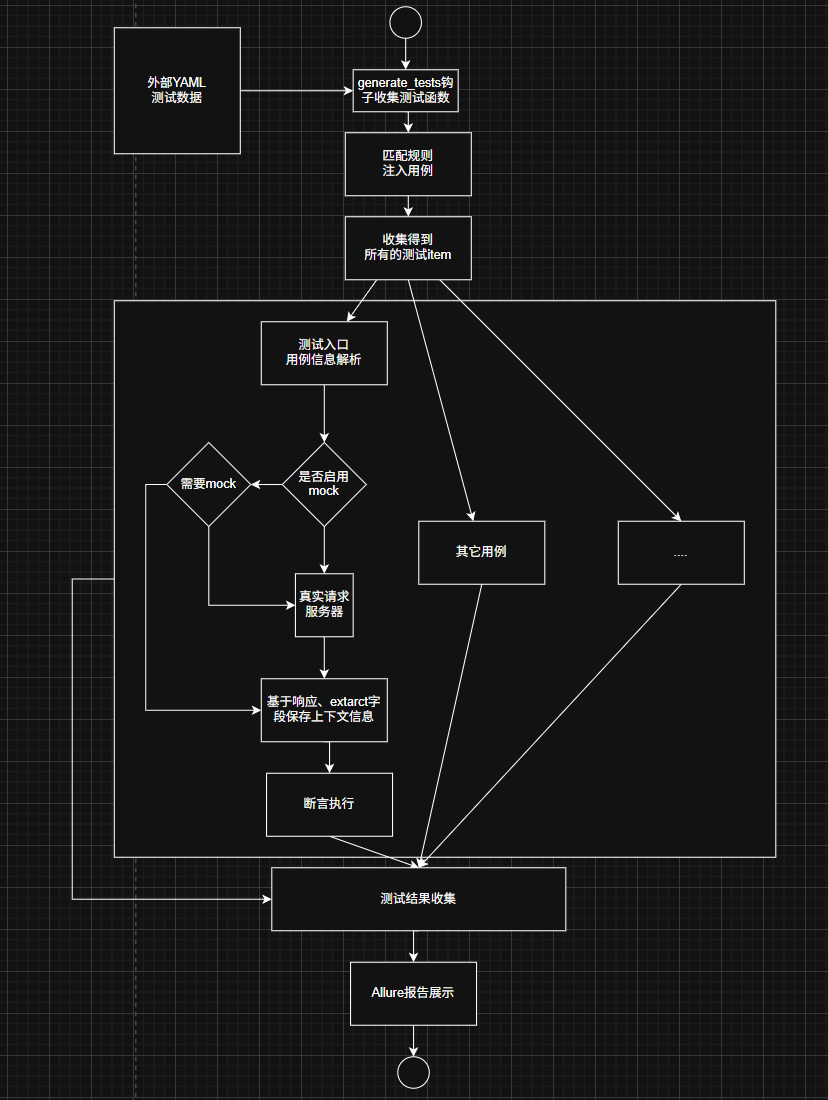
### 架构图

用例入口 → 数据读取 → 请求执行 → 响应断言 → 日志/报告

整体架构



执行流程



### 技术栈

本项目基于 Python 实现，核心测试框架采用 Pytest，使用 YAML 进行数据驱动，Requests 作为接口请求库，日志系统基于 logging 封装，测试报告通过 Allure 生成，支持多环境配置和用例动态参数化，具备一定的可扩展能力。

### 项目结构说明

项目结构如下：

**base：全局模块文件夹**

cases\_loader.py：用例加载模块，配合钩子来主动载入测试用例

env\_xml\_generater.py：动态生成测试环境配置的xml供allure使用

precon.yaml：前置条件运行数据

precondition.py：前置条件执行模块，配合yaml定义和钩子来独立执行前置条件。

remove.py：文件清理模块，主要用于测试前后的文件操作

requestBase.py：测试实际执行模块，整合用例分析、数据替换、请求执行、结果断言。

**conf:配置模块文件夹**

config.yaml：配置数据文件，通过修改其可以达成一些全局配置的修改

configLoader.py：数据加载模块，负责载出yaml文件的数据做全局变量的保存

**logs：日志存储文件夹，下面存储各天的日志**

**data：数据文件夹，存储用例yaml文件，与config.yaml指定的用例文件夹对于即可**

**mock：mock模拟数据文件夹**

mock\_server.py：mock服务器模块，定义mock服务器类

mock.yaml：定义需要mock的请求和其对应需要返回的模拟数据

**report：测试报告文件夹，存储allure生产的临时或持久报告**

**testcases：测试入口文件夹**

conftest.py：模块级的conftest配置

test\_entry.py：测试入口模块，包含实际被识别的测试函数

**utils：工具模块文件夹，包含辅助模块**

apiclient.py：请求模块，负责执行接口请求

assertions.py：断言模块，负责选择、执行以及收集断言

dbquery.py：数据库模块，定义数据库的连接和查询等功能

debugtalk.py：存储用例数据的${}函数动态替换，例如读取上下文、加密等

readyaml.py：yaml文件操作模块，负责包括用例yaml和extract.yaml的读写和转化。

recordlogs.py：日志记录模块，定义和实例化日志对象

**conftest.py：全局conftest模块，主要是定义钩子函数的操作**

**extract.yaml：上下文yaml文件，负责存储一些上下文信息**

**pytest.ini：pytest基础配置**

**run.py：框架运行入口，通过pytest.main来进行框架的运行命令执行**

## 用例设计与数据驱动实现

### 数据驱动设计思想

所谓数据驱动，就是把测试的用例从架构中抽离出来，这样实现数据与执行的分离，使得我们维护用例的数据文件就可以实现对用例的维护，而测试框架也可有更强的可复用性。

而我们数据驱动的数据格式使用yaml存储，yaml的优势是更简洁、易读、规范性强。当我们规范好一套全面的用例设计格式，就可以配合框架实现数据驱动的测试。整体的测试都是基于读取出来的数据运行，而框架内部只考虑执行逻辑，也就是所谓的“数据驱动“。

### Pytest 数据驱动实现

pytest实现数据驱动的最常见用法是使用parametrize来进行参数的注入，而我们使用的方法是通过generate\_tests来实现。

pytest\_generate\_tests(metafunc)是pytest底层的一个钩子，我们首先需要知道，pytest的部分执行流程是：检查指定文件夹与模块->收集所有用例函数->注入参数得到测试item->执行各个item，而generate\_test就是在收集用例函数后注入参数item时执行的钩子；通过generate\_tests实现参数化的原理就是在其收集到所有用例函数后，我们通过用例函数的参数来匹配，然后人为主动加载用例数据到这个测试函数，从而把外部数据注入到了指定的测试函数得到item，从而实现了参数化。

def pytest\_generate\_tests(metafunc):

    cl = ConfigLoader()

    \_,batch\_data = cl.dynamic\_load\_batch()

    if "single\_case\_data" in metafunc.fixturenames:

        cases = batch\_load\_testcases(batch\_data)

        metafunc.parametrize("single\_case\_data", cases)

这里的batch\_load\_testcases就是在把yaml的外部数据格式化加载成一个用例列表，再下面metafunc.parametrize就会把这个cases列表注入到"single\_case\_data"这个参数中，再接下来就是一个个执行用例item了。

### 用例分层思想

“用例分层思想就是对测试用例进行职责划分与逻辑分组，使测试执行更加有序、高效、可维护。”

大白话来说，就是把冒烟用例、单元功能用例、异常用例、回归用例这些用例进行区分，不要简单的一起都放回一次测试中执行。我的项目设计时并没有深入考虑到这一点，但是其实其天然能够执行用例分层，因为我们的框架实现了分批次和测试数据的全局配置调整，也就是说：如果希望只允许某个层次例如冒烟层（也就是进行冒烟测试），可以修改conf.yaml的对应数据文件路径配置，来实现只执行该层的用例；又或者说希望一次性执行，但是又需要进行分层，可以通过批次的配置实现，我们的框架目前是直接支持分批次运行和报告合并的，可以把各个层次配置成不同的批次来一次启动、多次运行框架来实现分层的运行，然后根据需要调整是否需要合并测试报告。

当然，毕竟不是原本就设计好了分层功能，一个可能的优化方式是加入一套自动的处理逻辑，还是需要人为进行分层，提供接口来收集本次执行的异常用例，自动添加到异常层或者回归层等。

## 接口请求封装与执行引擎

### 请求模块设计

请求模块其实就是通过yaml的数据来取出数据进行请求执行。下面是执行主函数的部分内容：

api\_name = base\_info['api\_name']

api\_url = base\_info['url']

method = base\_info['method']

headers = self.reload\_yaml\_function(base\_info['headers'])

#test\_case--case\_name处理用例名称

case\_name = test\_case.pop('case\_name',None)

#test\_case--validation处理验证数据

validations = test\_case.pop('validation',None)

#test\_case--extract处理上下文存储数据

extract = test\_case.pop('extract',None)

extract\_list = test\_case.pop('extract\_list',None)

#test\_case--case\_data处理各个类型的参数

data\_type = ['json','data','params']

for key,value in test\_case.items():

    if key in data\_type:

        test\_case[key] = self.reload\_yaml\_function(value)

#执行requests请求

res = self.apiclient.run\_request(api\_name,api\_url, method, case\_name,headers,\*\*test\_case)

其实就是从yaml初步转化后的字典中取出对应的字段，然后通过reload\_yaml\_function来进行内容的动态替换，例如上下文信息的读取、实时时间的替换、数据的加密操作等，然后把数据提交到request进行请求的执行。

请求模块的设计其实并不复杂，其依旧是使用request来发送请求获取结果而已，最主要是添加了很多中间请求信息的收集和异常日志记录。

此外需要注意的是，我的框架没有对cookie、token这些信息进行独立的处理，而是将其归入到headers字段中，也就是不需要额外的处理授权、认证，我会统一处理好对应的格式，会配合提取机制详细介绍具体操作。

### 依赖注入与参数替换机制

参数替换的方案是：yaml中通过${}标识需要替换，${}内部写函数，在用例执行解析用例数据时转化成json字符串，通过正则表达式匹配${}取出函数与参数，然后从DebugTalk动态替换类中取出对应的函数，再传入参数执行函数得到结果进行替换。

def reload\_yaml\_function(self,data):

    str\_data = data

    try:

        #data数据转化成json字符串,用于$函数的替换

        if not isinstance(data,str):

            str\_data = json.dumps(data,ensure\_ascii=False)

        #$函数替换

        pattern = r"\$\{(\w+)\((.\*?)\)\}"

        matches = re.findall(pattern, str\_data)

        for raw\_name,raw\_args in matches:

            args = ast.literal\_eval(f'({raw\_args},)') if raw\_args else []

            func = getattr(DebugTalk(),raw\_name)

            result = func(\*args)

            #替换表达式

            orgin\_exstr = f"${{{raw\_name}({raw\_args})}}"

            str\_data = str\_data.replace(orgin\_exstr,str(result))

            logs.info(f'【{orgin\_exstr}】替换称为【{result}】')

        #还原json字符串为数据

        data = json.loads(str\_data)

    except Exception as e:

        logs.error(f'yaml $ 数据重载处理出现异常：{e}')

        raise

    return data

其中使用的最多的还是上下文内容的替换例如下面的一个用例：

testCase:

    - case\_name: 正常删除商品

      data:

        productId: ${get\_extract\_data('goodsList',0)}

        timeStamp: 2169232689

      validation:

        - status\_code: 200

        - equal: { 'error': ''}

        - contains: { 'error\_code': '0000' }

其需要删除一个具体的商品，而这是需要一个具体的ID的，对于外部用例设计人员不需要也不应该记住这些ID，只需要通过get\_extract\_data函数即可，下面是其具体实现：

    def get\_extract\_data(self,node\_name,controller=None):

        """

        根据节点名称和控制器,调用YamlDataProcess类获取节点数据,根据controller有如下处理:

        1.数字:表明节点取出的内容为列表,再根据数字取值对列表进行操作

            a.-2,返回列表

            b.-1,返回字符串

            c.0,随机返回一个值

            d.自然数,表示取出列表的第controller个值

        2.其它：表明去除内容为正常的字典内容

            a.空,表明node\_name已经可以取出值

            b.非空,即controller表示二级节点,通过嵌套字典取值

        """

        #读数据异常由read\_extract\_data记录及抛出,只需处理controller为数字时异常

        if controller is None or isinstance(controller,str):

            data = self.extract\_reader.read\_extract\_data(node\_name,controller)

        elif isinstance(controller,int):

            list\_data = self.extract\_reader.read\_extract\_data(node\_name)

            try:

                if isinstance(list\_data,list):

                    if controller == -2:

                        #data = list\_data    #['a', 'b', 'c'] -> ['a', 'b', 'c']

                        data = ','.join(map(str, list\_data)).split(',')  #[1, 2, 3] -> ['1', '2', '3']

                    elif controller == -1:

                        data = ','.join(map(str, list\_data))  #['a', 'b', 'c'] -> 'a,b,c'

                    elif controller == 0:

                        data = random.choice(list\_data)

                    elif controller > 0:

                        data = list\_data[controller-1]

                    else:

                        data = None

                        logs.error("不支持的参数值")

                else:

                    data = list\_data

            except Exception as e:

                logs.error(f'contoller【{controller}】数字处理出现异常:{e}')

                raise

        else:

            data = None

            logs.error("不支持的参数类型")

            raise Exception(f"不支持的参数【{str(controller)}】-类型【{str(type(controller))}】")

        return data

配合注释查看，这其实也是一个中间的函数，而这里的read\_extract\_data才是实际 的读取，不过为了配合有时候一些跟丰富的处理情况构造这个功能更加完整的函数。

## 响应断言与提取机制

### 响应断言逻辑

设计的断言类型包含：状态码断言、包含断言、相当断言、不等断言、任意断言、数据库断言。其中包含与等\不等断言都支持列表的断言，也就是能够支持列表结果的处理。而且除了数据库断言，其余断言方式都是支持同类断言一次性执行的，也就是一个{}内可以写下多个键值进行断言。整体断言的处理机制是：针对每个要求的断言去执行，如果通过标志位是0，大于0则不通过，最后汇总。

在用例yaml中的validaton字段可以指定该用例需要进行的断言，指定字段、预期结果或者是数据库语句与预期结果，例如：

validation:

        - status\_code: 200

        - equal: { 'error\_code': '0000' }

        - nequal: { 'goodsList': none }

具体实现来说，对于状态码断言，就是简单通过响应对象的status\_code来比对；而其余处数据库断言以外的断言则是通过解析响应体，通过转化成jsonpath来查找对应的值然后进行对应的比对。

断言最大的难点其实是对空值的处理，需要规范好空值的表示和错误读取的处理方式，因为有时候有键无值表示空值，但是可能会导致代码异常，此外我们也希望提供更加宽泛、稳定性强的做法，下面是具体的实现：

def none\_process(self,value):

        n\_set = {'NONE','NULL',''}

        if str(value).upper() in n\_set:

            return None

        else:

            return value

def equal\_assert(self,val\_value, res\_js):

        equal\_flag = 0

        for expected,v\_value in val\_value.items():

            e\_flag = 0

            v\_value = self.none\_process(v\_value)

            extract\_list = jsonpath.jsonpath(res\_js, f'$..{expected}')

            if not extract\_list:

                if v\_value:

                    e\_flag +=1

                continue

            #处理列表和单值情形：列表要求预期列表和响应结果排序后相等；单值要求响应结果至少有一个等于预期值

            if isinstance(v\_value,list):    #列表

                extract\_list = extract\_list[0]

                if sorted(v\_value) != sorted(extract\_list):

                    e\_flag += 1

            else:

                if not any(self.none\_process(exact\_value) == v\_value for exact\_value in extract\_list):

                    e\_flag += 1

            #根据e\_flag判断单条验证是否通过

            if len(extract\_list) == 1:extract\_list = extract\_list[0]

            assert\_info = f'【{expected}】预期值：【{v\_value}】,实际值：【{extract\_list}】'

            if e\_flag:

                logs.error(f'❌相等断言\*\*失败\*\*，{assert\_info}')

                allure.attach(assert\_info,'❌相等断言失败',allure.attachment\_type.TEXT)

            else:

                logs.info(f'✅相等断言\*\*通过\*\*，{assert\_info}')

                allure.attach(assert\_info,'✅相等断言成功',allure.attachment\_type.TEXT)

            equal\_flag += e\_flag

        return equal\_flag

### 提取变量机制

上下变量的提取配合用例的extract或者extract\_list字段使用，后面的list不是指针对一个键得到一个列表，而是一个键会匹配到多个单独内容，需要收集所有内容用列表存储。在执行主逻辑中，先解析处对应字段，然后在获取到响应结果后调用extract\_data或extract\_data\_list来进行内容提取的写入

extract\_list:

      goodsList: $.goodsList[\*].goodsId

这提取到的内容也自然是存储到extract.yaml中，下面是一个完整的提取逻辑

def extract\_data(self,extract,res):

        """

        输入extract字典,响应内容,通过extract的value表达式从响应内容中提取数据,并写入yaml文件

        支持多种响应内容提取:

        - header. :直接提取的头部的json字段

        - cookies :直接从响应对象获取,而不是从响应头提取,进行拍平存储,便于直接读出

        - cookies. :从cookies中取出字段存储,一般处理cookies内的token等信息

        - jsonpath :使用jsonpath从响应体获取信息,得到且只需要得到[aa]这样的内容,取出aa写入

        - 正则 :使用正则表达式从响应体中提取信息

        yaml格式:

        extract:

          csrf: headers.X-CSRF-Token

          cookies: cookies

          token: cookies.token

          userId: $..userId

          userAge: '"token"\s\*:\s\*"([^"]+)"'

        """

        for key,value in extract.items():

            extract\_data = {}

            try:

                #headers提取,例如：csrf: headers.X-CSRF-Token

                if value.startswith('headers.'):

                    header\_key = value.split('.', 1)[1]

                    if (res\_data := res.headers.get(header\_key)):

                        extract\_data = {key:res\_data}

                    else:

                        raise Exception(f"【{value}】 未在--响应头--找到字段【{header\_key}】,响应头为：{res.headers}:")

                #处理整个cookie提取,例如：cookies: cookies

                elif value == 'cookies':

                    if (cookie\_dict := dict\_from\_cookiejar(res.cookies)):

                        res\_data = "; ".join([f"{k}={v}" for k, v in cookie\_dict.items()])

                        extract\_data = {key:res\_data}

                    else:

                        raise Exception(f"【{value}】 未在--响应头--找到cookies,响应头为：{res.headers}")

                #处理从cookies中提取数据,例如：token: cookies.token

                elif value.startswith('cookies.'):

                    cookie\_key = value.split('.', 1)[1]

                    if (res\_data := res.cookies.get(cookie\_key)):

                        extract\_data = {key:res\_data}

                    else:

                        raise Exception(f"【{value}】未在--响应头cookies--找到字段【{cookie\_key}】,响应头为：{res.headers}")

                #jsonpath提取数据

                elif '$' in value:

                    if (res\_data\_list := jsonpath.jsonpath(res.json(),value)):

                        if res\_data\_list[0] is not None:

                            extract\_data = {key:res\_data\_list[0]}

                    else:

                        raise Exception(f"【{value}】--jsonpath--未在响应内容中找到字段【{key}】数据,响应体为：{res.text}")

                #正则提取数据

                else:

                    if (match\_data := re.search(value, res.text)):

                        if r'(\d)' in value or r'(\d\*)' in value:

                            res\_data = int(match\_data.group(1))

                        else:

                            res\_data = match\_data.group(1)

                        extract\_data = {key:res\_data}

                    else:

                        raise Exception(f"【{value}】--正则表达式--未在响应内容中找到字段【{key}】数据,响应体为：{res.text}")

            except Exception as e:

                logs.error(f'extract写入出现异常：{e}')

                raise

            if extract\_data:

                self.write\_extract\_data(extract\_data)

其支持直接响应头、全cookie、cookie内部内容、响应体jsonpath和正则的提取，其中最需要着重提及的就是cookie的提取，我们不一个个cookie的内容都取出来保存，而是直接从响应头中取出cookie用k=v;的格式直接保存完整cookie，这样下次使用时只需要header的cookie字段写${}来直接替换即可，就不需要额外带一个cookie参数给request了。

## 日志系统设计

### 日志封装实现

日志可以用于输出框架的运行中间信息，相较于简单的print控制台输出，使用日志能够实现持久保存，同时也能实现控制台的输出。

日志类的实现其实就是通过logging来配置好日志的写入级别、输出级别、日志大小、备份数量、输出文件夹等，然后实例化一个logger对象供其它模块使用就能实现日志的记录了。

### 如何将日志挂载到 Allure

目前pytest和allure插件似乎是能够自动捕获每个用例执行期间产生的日志，自动会挂在到报告中。

## Allure 报告集成与优化

### 使用方式

pytest --alluredir= + allure serve

在框架执行时带上--alluredir=指定报告临时数据输出文件夹即可在框架运行时通过allure.attch()来添加报告内容的添加，与日志不同的是，日志会经可能多记录中间过程的数据，尤其是一些异常的数据，而allure注重的是用例的信息和响应与断言结果。

### 报告结构与内容说明

features / stories /title/ steps/attachment属于allure报告结构级别，用于树状划分结构内容。

## 用例依赖与上下文变量机制

### 为什么需要依赖机制？

依赖机制主要是由于部分接口需要的一些token等数据可能是需要依赖于之前执行的接口，再通过extract进行保存，一种初期的依赖实现的策略是指定好各个用例的执行顺序，人为空值顺序来保证前置条件先执行。

### 当前的实现方案

extract.yaml + pytest\_runtest\_setup

但是显而易见，上面的做法是不推荐的，因为这样用例间就会耦合起来，作为前置条件的用例错误失败时会导致后续的用例一起出错。所以我们的解决策略是把前置条件独立出来，每个用例单独执行一次所有的前置条件，虽然运行成本提高，但是用例间的独立性更强。

具体的做法是，为每个用例添加一个preconditions字段，通过pytest\_runtest\_setup钩子来在实际用例执行前先跑一次前置条件，然后存储好上下文信息供后续使用。

还有一种最常见的依赖管理手段就是通过fixture来执行，但是由于我们的框架使用的是统一、高抽象级别的用例入口，所以不适合使用fixture装饰器来处理，而其使用fixture装饰器其实限制性也比较高。

# 从零搭建接口测试框架

下面是我从零开始搭建项目的流程，属于跟着别人的代码边看边学，然后融入自己的想法和其它地方进行学习优化的过程。

## 测试入口与用例提取

所谓数据驱动，其实就是通过读取外部数据来注入统一的用例执行逻辑，达到数据与执行分离的效果。最常见的实现方法就是使用pytest的parametrize参数化来注入参数：

@pytest.mark.parametrize(['base\_info','test\_case'],read\_testcase\_from\_yaml('./testcases/cartAPI/cart\_shoppingJoinCart.yaml'))

    def test\_add\_to\_cart(self, base\_info, test\_case):

上面示例的装饰器就是通过read\_testcase\_from\_yaml来读取出测试用例的用例信息，注入到测试函数中，这样我们实现了数据的外部导入。

### 实现

下面看一种用例文件的写法和对应的read\_testcase\_from\_yaml()函数：可以自行修改，需要数据格式与读取逻辑对应

#### 用例模板：

注意：目前我们从单用例的接口做起，流程的测试也是建立在单接口的基础上。

这里我们通过validation字段来处理断言、通过extract以及extract\_list实现上下文的保存

- baseInfo:

    api\_name: 购物车删除商品接口

    url: /coupApply/cms/delCart

    method: POST

    headers:

      Content-Type: application/x-www-form-urlencoded

  testCase:

    - case\_name: 正常删除商品

      data:

        productId: "33809635011"

        timeStamp: 2169232689

      validation:

        - contains: { 'error\_code': '0000' }

    - case\_name: 有效删除-非必须参数timeStamp缺失

      data:

        productId: "33809635011"

      validation:

        - contains: { 'error\_code': '0000' }

#### 函数实现：

通过这个函数，我们需要得到的一个用例列表，每个元素都是基础信息（端点、请求头等用例共用信息）+用例专属信息的列表或者元组。通过规范化的数据内容来处理数据。

def read\_testcase\_from\_yaml(file):

    test\_list = []

    try:

        with open(file, 'r',encoding='utf-8') as f:

            yaml\_data = yaml.safe\_load(f)

            if len(yaml\_data) <= 1:

                data = yaml\_data[0]

                base\_info = data['baseInfo']

                testcase\_list = data['testCase']

                for tc in testcase\_list:

                    test\_list.append((base\_info,tc))

                return test\_list

            else:

                return yaml\_data

## 用例执行主逻辑

目前我们的做法是每个接口使用独立的用例入口，通过参数化注入数据，而所有用例执行的主逻辑是一样的，例如下面的excute\_test()函数，其负责处理主要的执行逻辑，而这里的测试函数只是负责让pytest识别到来执行测试。

@pytest.mark.parametrize(['base\_info','test\_case'],read\_testcase\_from\_yaml('./testcases/cartAPI/cart\_shoppingJoinCart.yaml'))

    def test\_add\_to\_cart(self, base\_info, test\_case):

        RequestBase().excute\_test(base\_info,test\_case)

这里我们维护一个requestbase模块中的RequestBase类来进行主逻辑维护，通过excute\_test执行主逻辑。

### 执行函数

我们由读取函数从yaml文件得到的一个用例可以解包得到基础信息+用例信息，执行函数要做的就是从基础信息读取出接口名称、端点、请求头，从用例信息中读取出请求数据、上下文信息、断言信息，然后根据这些信息来执行接口请求，得到响应体后，根据响应体内容处理上下文信息保存和断言处理：

#### 数据提取

前面我们使用yaml.load会得到的是python字典数据，我们只需要以读取字典的方式读出对应的数据即可：

            api\_name = base\_info['api\_name']

            api\_url = base\_info['url']

            method = base\_info['method']

            headers = self.reload\_yaml\_function(base\_info['headers'])

            #test\_case--case\_name处理用例名称

            case\_name = test\_case.pop('case\_name',None)

            #test\_case--validation处理验证数据

            validations = test\_case.pop('validation',None)

            #test\_case--extract处理上下文存储数据

            extract = test\_case.pop('extract',None)

            extract\_list = test\_case.pop('extract\_list',None)

            #test\_case--case\_data处理各个类型的参数

            data\_type = ['json','data','params']

            for key,value in test\_case.items():

                if key in data\_type:

                    test\_case[key] = self.reload\_yaml\_function(value)

注意：

1. 对于请求数据，由于请求类型可能有表格（data）、json、查询参数（params）等，所以我们需要分别处理，不能简单粗暴读取
2. 为了实现上下文的读取，也就是某些接口可能会依赖前置条件的结果，而为了更灵活的实现我们选择使用extract.yaml文件的持久化保存来处理数据，通过用例数据yaml文件的${}标识函数的动态提取或者一些其它处理，后面我们会详细介绍reload\_yaml\_function

#### 动态转换

有些时候我们连yaml数据驱动的数据也需要动态处理，使用${函数}来表示需要动态替换函数操作加载数据，例如上下文动态提取、前端一些加密处理、实时的时间加载等都会用到：

def reload\_yaml\_function(self,data):

        str\_data = data

        try:

            #data数据转化成json字符串，用于$函数的替换

            if not isinstance(data,str):

                str\_data = json.dumps(data,ensure\_ascii=False)

            #$函数替换

            pattern = r"\$\{(\w+)\((.\*?)\)\}"

            matches = re.findall(pattern, str\_data)

            for raw\_name,raw\_args in matches:

                args = ast.literal\_eval(f'({raw\_args},)') if raw\_args else []

                func = getattr(DebugTalk(),raw\_name)

                result = func(\*args)

                #替换表达式

                orgin\_exstr = f"${{{raw\_name}({raw\_args})}}"

                str\_data = str\_data.replace(orgin\_exstr,str(result))

                logs.info(f'【{orgin\_exstr}】替换称为【{result}】')

            #还原json字符串为数据

            data = json.loads(str\_data)

        except Exception as e:

            logs.error(f'yaml $ 数据重载处理出现异常：{e}')

            raise

        return data

这里我们先忽视异常处理和日志处理，这里的动态加载实现，我们遍历通过正则表达式查找$字段，ast.literal\_eval来处理函数的传参，getattr()来处理函数获取，这里我们维护一个DebugTalk类来管理所有的动态替换的函数，包括extract.yaml的加载和一些加密的处理，我们找到函数执行获取数据后再在元数据进行替换。

注意：

1. json.dumps会把字典拍扁成一长条的字符串形式，便于我们识别内容和替换
2. 上面代码置于RequestBase类或者requestbase模块或者readyaml模块都可
3. 上面功能置于模块readyaml.py中

#### 请求执行、上下文管理与断言执行

请求执行、上下文管理与断言执行我们都通过独立的模块来处理，主执行逻辑只负责主要逻辑的组织

* **请求执行**

我们从前面提取与替换得到数据后，通过requests库来执行请求，得到响应内容用于上下文保存和断言执行。

res = self.apiclient.run\_request(api\_name,api\_url, method, case\_name,headers,\*\*test\_case)

* **上下文管理**

正如前面所说，我们使用extract.yaml与用例yaml的${}实现动态替换

if extract:

     self.extract\_data(extract,res)

if extract\_list:

     self.extract\_data\_list(extract\_list,res)

* **断言执行**

断言执行我们从yaml用力数据得到了validation字段后，通过一个断言类来统一的各种断言的处理

assert self.assertions.assert\_result(validations,res)

## 用例上下文管理

上下文的管理我们通过extract.yaml实现，主要就是读写操作

### 上下文保存

上下文保存通过extract，就是写如extract.yaml即可。

### 上下文提取

通过yaml的提取我们结合动态替换${}即可实现，

#### 基础实现：

最基础的实现其实就是把extract.yaml读取出来，通过字典的方式把内容提取出来（置于readyaml.py模块中）

def read\_extract\_data(self,node\_name,scd\_node\_name = None):

        try:

            with open(self.extract\_yaml\_path, 'r', encoding='utf-8') as f:

                extract\_data = yaml.safe\_load(f)

                if scd\_node\_name is None:

                    return extract\_data[node\_name]

                else:

                    return extract\_data[node\_name][scd\_node\_name]

        except Exception as e:

            logs.error(f"读取【{self.extract\_yaml\_path}】文件【{node\_name}】【{scd\_node\_name}】时出现错误: {e}")

            raise

#### 进阶管理：

虽然前面实现了基础的数据提取，但是我们需要考虑一些问题：列表该如何处理、多级字典的内容如何兼顾。处理列表时又会又问题就是：不知道列表长度及只需要一个值时如何处理？

这里我们引入一个contoller变量来统一字典索引和列表

**Controller**

再上面基础实现的基础上，我们通过contoller来统一处理，如果其是数字，要求第一节点得到的数据是列表，然后根据contoler的值来处理列表的内容；如果是字符串，将其视为原本的第二节点，进行嵌套字典的索引。我们有如下规则：

**1.数字:**表明节点取出的内容为列表，再根据数字取值对列表进行操作

   a.-2，返回列表

   b.-1，返回字符串

   c.0，随机返回一个值

   d.自然数，表示取出列表的第controller个值

**2.其它**：表明去除内容为正常的字典内容

   a.空，表明node\_name已经可以取出值

   b.非空，即controller表示二级节点，通过嵌套字典取值

此外，我们保留基础实现，在其中间加上一层负责分发与列表处理即可（下面代码置于Debugtalk类用于动态替换进行调用）：

    def get\_extract\_data(self,node\_name,controller=None):

        if controller is None or isinstance(controller,str):

            data = self.extract\_reader.read\_extract\_data(node\_name,controller)

        elif isinstance(controller,int):

            list\_data = self.extract\_reader.read\_extract\_data(node\_name)

            try:

                if isinstance(list\_data,list):

                    if controller == -2:

                        data = ','.join(map(str, list\_data)).split(',')

                    elif controller == -1:

                        data = ','.join(map(str, list\_data))

                    elif controller == 0:

                        data = random.choice(list\_data)

                    elif controller > 0:

                        data = list\_data[controller-1]

                    else:

                        data = None

                        logs.error("不支持的参数值")

                else:

                    data = list\_data

            except Exception as e:

                logs.error(f'contoller【{controller}】数字处理出现异常:{e}')

                raise

        else:

            data = None

            logs.error("不支持的参数类型")

            raise Exception(f"不支持的参数【{str(controller)}】-类型【{str(type(controller))}】")

        return data

## 接口执行

接口的执行其实很容易，就是使用request库来处理即可，主要在于接口信息的处理，我们维护一个ApiClient类，负责收集请求和响应信息，以及发送接口请求即可，最主要的功能还是集成后面的日志记录和Allure报告内容的收集。

实现方法就是一个send\_request方法处理实际的发送请求，run\_request来收集处理主逻辑的内容，进行url、接口名称、测试数据等数据的日志记录和allure报告记录，然后再调用send\_requsest执行请求，得到响应后也会记录一次响应内容到allure。

class ApiClient:

    def \_\_init\_\_(self, base\_url=configLoader.API\_HOST):

        self.base\_url = base\_url

    def send\_request(self, \*\*kwargs):

        ….

    def run\_request(self, api\_name,

                    url\_endpoint,

                    method,

                    case\_name,

                    headers,

                    \*\*tc\_kwargs):

…..

## 测试断言

断言是pytest测试的终点线，通过断言来判断测试用例的是否通过，我们通过一个Assertions类来统一处理断言。我们断言的处理逻辑配合yaml validation字段的处理逻辑是这样的：validation字段通过 类型: {数据}来实现断言，断言类型目前提供：状态码断言、包含断言、相等断言、不等断言、数据库断言（、任意断言），下面我们由整体断言逻辑到各自断言函数介绍：

### 断言主逻辑

我实现断言的方法是：断言类获取是否通过，返回真假值，测试主逻辑进行断言，而具体到每种断言中，通过标志位来管理是否通过，0表示没有失败，大于0表示失败的个数，因为目前我们支持一个字段可以通过嵌套字典处理多个同类型断言。

    def assert\_result(self,validations,res):

        """

        传入validations的验证字典列表，根据列表元素的键决定处理方式，如下：

        'status\_code':判断response的http状态码

        'contains':判断字段内容是否包含

        'equal':判断字段内容是否相等

        'nequal':判断字段内容是否不相等

        'any':判断任意字段内容是否相等，可以进入嵌套字典

        'db':数据库断言

        """

        #logs.info(f'开始执行断言\n预期结果：{validations}\n实际结果：{res.json()}')

        sucess\_flag = 0

        res\_data = res.json()

        for val in validations:

            for key,value in val.items():

                if key == 'status\_code':

                    sucess\_flag += self.status\_code\_assert(value, res.status\_code)

                elif key == 'contains':

                    sucess\_flag += self.contains\_assert(value, res\_data)

                elif key == 'equal':

                    sucess\_flag += self.equal\_assert(value, res\_data)

                elif key == 'nequal':

                    sucess\_flag += self.nequal\_assert(value, res\_data)

                elif key == 'any':

                    sucess\_flag += self.any\_assert(value, res\_data)

                elif key == 'db':

                    sucess\_flag += self.db\_assert(value)

                else:

                    raise ValueError(f'未知验证类型：【{key}】')

        return not bool(sucess\_flag)

### 状态码断言

状态码断言很简单，就是从响应对象获取即可，主要是状态码不是直接从响应体得到，因此直接独立出来处理。

def status\_code\_assert(self,val\_value, sc):

        """

        比较http状态码，yaml格式如下：

        - status\_code: 200

        """

        assert\_info = f'预期状态码：【{val\_value}】,实际状态码：【{sc}】'

        if sc == val\_value:

            return 0

        else:

            return 1

需要注意的是，后面需要把日志记录和allure记录补上，在返回0、1前记录预期与实际结果。

### 空值处理

再处理后面三种断言前，我们需要先处理一下空值，由于包含、等、不等断言都是需要读取响应体的内容与yaml的内容进行对比，而且我们通过jsonpath进行快速的字段匹配；我们需要知道：

1. jsonpath查找字段内容得到的是一个列表，也就是说，如果响应内容是空，后端可能直接不给字段，可能是字段+’’，列表可能是字段+[]，所以我们得到的jsonpath结果就可能是[]、[‘’]、[[]]甚至[[],[]]这样子的内容
2. yaml validation的空值写法也可能不统一，为了更灵活，我们可以支持null、none、空这样的写法

响应内容的数据可以再断言时处理，但是yaml读取出来的字段内容确实需要比较频繁的判断，由此我们可以把空值处理独立出来，识别到后替换位None对象：

def none\_process(self,value):

        """

        处理None值，便于断言处理

        """

        n\_set = {'NONE','NULL',''}

        if str(value).upper() in n\_set:

            return None

        else:

            return value

这个可以拓展n\_set实现更加广泛的空值处理

### 包含断言

在处理断言前，一定要先规范yaml中的数据格式，我们规范支持下面的validation字段：

- contains: { 'msg': '成功' }

- contains: { 'goodsId': ['1222', '1223'] }

- contains: { 'goodsId': ['1222', '1223'] , 'msg': '成功'}

我们是支持一次处理多个同类型断言的，在内部我们的处理步骤是这样的：

1. 对于预期值，我们先进行一次空值检查替换
2. 从响应体使用jsonpath获取内容列表，可能是[‘’] [‘xxxx’] [[]]等
3. 第一次判断：如果内容列表为空做判断：预期值也为空则通过，预期值不为空则标志位加一
4. 分列表与单值处理
5. 如果预期为列表，从内容列表获取第一个列表（预期结构应该是[[….],[…]]）与预期列表进行子集的判断
6. 如果是单值，通过not any()来要求内容列表中，至少有一个值包含了预期字段
7. 根据标志位判断此部分断言是否通过

包含日志与allure的实现：

    def contains\_assert(self,val\_value, res\_js):

        """

        比较字段内容是否包含,单值要求字符串包含,列表要求集合包含,返回0表示通过,

        """

        contain\_flag = 0    #整体标志位

        for expected,v\_value in val\_value.items():

            c\_flag = 0

            v\_value = self.none\_process(v\_value)

            extract\_list = jsonpath.jsonpath(res\_js, f'$..{expected}')  #jsonpath得到的一个包含键对应的值组成的列表[[xx,xx],xx]

            #列表为空，找不到值，视为错误

            if not extract\_list:

                if v\_value:

                    c\_flag +=1

                continue

            if isinstance(v\_value,list):    #列表,要求集合包含

                extract\_list = extract\_list[0]

                if not set(v\_value).issubset(set(extract\_list)):

                    c\_flag += 1

            else:

                #单值，要求结果内至少有一个包含预期值

                if not any(str(v\_value) in str(self.none\_process(exact\_value)) for exact\_value in extract\_list):    #str()也可以处理None值

                    c\_flag += 1

            #根据c\_flag判断单条验证是否通过

            if len(extract\_list) == 1:extract\_list = extract\_list[0]

            assert\_info = f'【{expected}】预期值：【{v\_value}】,实际值：【{extract\_list}】'

            if c\_flag:

                logs.error(f'❌包含断言\*\*失败\*\*，{assert\_info}')

                allure.attach(assert\_info,'❌包含断言失败',allure.attachment\_type.TEXT)

            else:

                logs.info(f'✅包含断言\*\*通过\*\*，{assert\_info}')

                allure.attach(assert\_info,'✅包含断言成功',allure.attachment\_type.TEXT)

            contain\_flag += c\_flag

        return contain\_flag

注意：

1. 列表处的extract\_list = extract\_list[0]用于提取内容比对，而断言预期前的if len(extract\_list) == 1:extract\_list = extract\_list[0]用于美化日志和报告内容
2. 注意none\_process替换空值的处理

### 相等与不相等断言

相等与不等断言的处理与包含基本一致，主要在于如何实现“等”“不等”的判断：

#### 相等断言

相等断言处理内容列表为空而预期不为空的情况肯定是认为错误，处理列表时，我们要求两个列表排序后相等，也就是不要求顺序但是排序后完全相等

    def equal\_assert(self,val\_value, res\_js):

        """

        比较字段内容是否相等,列表要求完全相同,单值要求至少存在一个完全相同,返回0表示通过,

        """

        equal\_flag = 0

        for expected,v\_value in val\_value.items():

            e\_flag = 0

            v\_value = self.none\_process(v\_value)

            #extract\_list = jsonpath.jsonpath(res\_js, f'$.{expected}')   #只检查一级字典

            extract\_list = jsonpath.jsonpath(res\_js, f'$..{expected}')

            if not extract\_list:

                if v\_value:

                    e\_flag +=1

                continue

            #处理列表和单值情形：列表要求预期列表和响应结果排序后相等；单值要求响应结果至少有一个等于预期值

            if isinstance(v\_value,list):    #列表

                extract\_list = extract\_list[0]

                if sorted(v\_value) != sorted(extract\_list):

                    e\_flag += 1

            else:

                if not any(self.none\_process(exact\_value) == v\_value for exact\_value in extract\_list):

                    e\_flag += 1

            #根据e\_flag判断单条验证是否通过

            if len(extract\_list) == 1:extract\_list = extract\_list[0]

            assert\_info = f'【{expected}】预期值：【{v\_value}】,实际值：【{extract\_list}】'

            if e\_flag:

                logs.error(f'❌相等断言\*\*失败\*\*，{assert\_info}')

                allure.attach(assert\_info,'❌相等断言失败',allure.attachment\_type.TEXT)

            else:

                logs.info(f'✅相等断言\*\*通过\*\*，{assert\_info}')

                allure.attach(assert\_info,'✅相等断言成功',allure.attachment\_type.TEXT)

            equal\_flag += e\_flag

        return equal\_flag

#### 不等断言

类似相等断言，判断反过来即可

    def nequal\_assert(self,val\_value, res\_js):

        nequal\_flag = 0

        for expected,v\_value in val\_value.items():

            ne\_flag = 0

            v\_value = self.none\_process(v\_value)

            #extract\_list = jsonpath.jsonpath(res\_js, f'$.{expected}')   #只检查一级字典

            extract\_list = jsonpath.jsonpath(res\_js, f'$..{expected}')

            if not extract\_list:

                if not v\_value:

                    ne\_flag +=1

                continue

             #处理列表和单值情形：列表要求预期列表和响应结果排序后不相等；单值要求响应结果每一个都不等于

            if isinstance(v\_value,list):    #列表

                extract\_list = extract\_list[0]

                if sorted(v\_value) == sorted(extract\_list):

                    ne\_flag += 1

            else:

                if any(self.none\_process(exact\_value) == v\_value for exact\_value in extract\_list):

                    ne\_flag += 1

            #根据ne\_flag判断单条验证是否通过

            if len(extract\_list) == 1:extract\_list = extract\_list[0]

            assert\_info = f'【{expected}】预期值：【{v\_value}】,实际值：【{extract\_list}】'

            if ne\_flag:

                logs.error(f'❌不等断言\*\*失败\*\*，{assert\_info}')

                allure.attach(assert\_info,'❌不等断言失败',allure.attachment\_type.TEXT)

            else:

                logs.info(f'✅不等断言\*\*通过\*\*，{assert\_info}')

                allure.attach(assert\_info,'✅不等断言成功',allure.attachment\_type.TEXT)

            nequal\_flag += ne\_flag

        return nequal\_flag

### 数据库断言

数据库断言相较于前面几种断言比较特殊，数据库断言不需要响应对象，至于要预期结果与数据库查询语句，以sql为例：

- db: { expected:'200034', sql:'SELECT good\_id FROM goods WHERE good\_name = "iPhone X"' }

由于肯预期结果是列表，因此不能实现预期:sql这样的写法，因此数据库断言只支持一次一条处理。

#### 数据库封装

由于后端可能使用不同的数据库，我们需要封装redis、MySQL等数据库，而数据库在测试主要支持查询即可，当然也可以支持实现更多的操作，主要是可能需要撤回一些操作，下面以MySQL为例：

class ConnectMySQL:

    def \_\_init\_\_(self):

        #从配置文件读取

        mysql\_conf = configLoader.MYSQL\_CONFIG

        try:

            self.conn = pymysql.connect(\*\*mysql\_conf, charset='utf8')

            # cursor=pymysql.cursors.DictCursor,将数据库表字段显示，以key-value形式展示

            self.cursor = self.conn.cursor(cursor=pymysql.cursors.DictCursor)

        except Exception as e:

            logs.error(f"MySql连接异常:{e}")

            raise

    def close(self):

        try:

            self.cursor.close()

            self.conn.close()

        except Exception as e:

            logs.error(f"MySQL关闭连接异常:{e}")

            raise

    def delete(self, sql):

        try:

            self.cursor.execute(sql)

            self.conn.commit()

        except Exception as e:

            logs.error(f" MySQL删除异常:{e}")

            raise

    def query(self,sql):

        """

        查询语句，返回列表格式：

        - 如果查询结果有数据，返回 [value1, value2,...]扁平列表

        - 如果查询结果为空,或sql语句错误等,返回空列表

        """

        try:

            self.cursor.execute(sql)

            query\_result = self.cursor.fetchall()

            if query\_result:

                return [list(row.values())[0] for row in query\_result]

            else:return []

        except Exception as e:

            logs.error(f"MySQL语句【{sql}】查询异常:{e}")

            raise

注意：

1. cursor=pymysql.cursors.DictCursor来把查询内容进行格式化的处理
2. 把查询结果拍扁，只支持一次查询一个字段的内容得到一个扁平的列表

#### 断言处理

得到规范处理的查询结果后，处理就简单了，如果预期是列表，只需要排序后比较，不是列表则取第一个值处理即可，但是要注意一些错误语句、空值的处理

    def db\_assert(self,val\_value):

        """

        比较数据库查询结果是否相等,单次只能比较一条，支持列表对比

        """

        flag = 0

        sql\_db = ConnectMySQL()

        expected = val\_value.get('expected',None)

        if not (sql := val\_value.get('sql',None)):

            logs.error(f'数据库断言失败，sql语句为空')

            return 1

        query\_list = sql\_db.query(sql)  #得到查询结果的列表

        #空值

        none\_set = {'NONE','NULL','[]',''}

        if not query\_list:

            #预期与得到皆为空

            if str(expected).upper() in none\_set:pass

            #预期非空，得到为空

            else:flag += 1

        #单值

        elif not isinstance(expected,list) and query\_list:

            if expected == query\_list[0]:  pass#只检查第一个值;其中一个满足即可：if expected in query\_list:

            else:flag += 1

        #列表

        elif isinstance(expected,list) and query\_list:

            if sorted(expected) != sorted(query\_list):#列表排序后比较

                flag += 1

        #其它

        else:flag += 1

        sql\_db.close()

        assert\_info = f'sql语句：【{sql}】\n预期结果：【{expected}】,实际结果：【{query\_list}】'

        if flag:

            logs.error(f'❌数据库断言\*\*失败\*\*，{assert\_info}')

            allure.attach(assert\_info,'❌数据库断言失败',allure.attachment\_type.TEXT)

        else:

            logs.info(f'✅数据库断言\*\*通过\*\*，{assert\_info}')

            allure.attach(assert\_info,'✅数据库断言成功',allure.attachment\_type.TEXT)

        return flag

## 日志与异常处理

日志记录和异常处理都是一个接口测试框架非常重要的部分，他们不参与主要的测试，但是对查找错误、维持测试系统稳定性有着非常重要的作用。

### 日志记录

“在计算机行业中，“日志（Log）”是指程序在运行过程中输出的一种文本信息，用于记录程序执行的状态、行为、异常等信息。日志广泛应用于开发、调试、测试和运维阶段。”

日志记录主要依靠的是logging库，在测试框架中的处理是：通过定义一个日志类，在模块中实例化这个类，其它模块通过这个全局的实例化的日志记录对象来进行日志的记录。

#### 日志类

日志类最主要需要考虑的问题有：

1. 日志文件保存到哪里
2. 日志的级别控制
3. 日志如何保存（格式、内容）
4. 日志的配置、初始化。

* **文件保存**

日志文件需要保存且进行区分，最简单的方式就是通过时间进行分区，每天一个大文件名，根据内容与备份文件名下面可能有备份等，通过设置在logger对象配置日志处理器，设置RotatingFileHandler来指定日志文件的文件名、最大内容大小、备份数、编码方式等，可以设置好日志文件的保存方式。

* **级别控制**

日志的级别控制主要就是日志级别以及输出流级别，各个级别如下：

|  |  |
| --- | --- |
| **等级** | **描述** |
| DEBUG | 调试信息，最详细，开发阶段使用 |
| INFO | 关键信息，程序正常运行时的提示 |
| WARNING | 警告信息，不影响程序运行但可能导致问题 |
| ERROR | 错误信息，程序某部分出错但未崩溃 |
| CRITICAL | 严重错误，程序即将崩溃或必须立即处理 |

对于测试框架主要是使用的都是DEBUG

* **日志内容**

日志的内容主要是通过logging.Formatter来设置，如下

log\_format = logging.Formatter(

                '%(levelname)s - %(asctime)s - %(filename)s:%(lineno)d -[%(module)s:%(funcName)s] - %(message)s')

            """INFO - 2025-05-20 11:58:35,604 - assertions.py:198 -[assertions:assert\_result] - 测试成功"""

实际运行记录日志时，logging模块会自动注入内容，这样就可以控制日志输出的内容样式

基于上面的内容，加上一个设置的日志内容保存日期可以得到下面的一个日志类：

class RecordLogs:

    """

    日志类,负责日志对象获取,初始化设置、过期日志管理

    """

    def \_\_init\_\_(self, log\_folder):

        self.log\_folder = log\_folder

        self.logfile\_name = os.path.join(log\_folder,f'test.{time.strftime("%Y%m%d")}.log')  #日志文件名,按天生成日志文件

        self.handle\_overdue\_logs()  #每次启用处理过期日志

    def handle\_overdue\_logs(self):

        now\_time = datetime.datetime.now()  #datetime对象：2025-05-30 21:17:46.430581

        time\_offset = datetime.timedelta(days=configLoader.RENTENTION\_DAYS)  #时间差对象

        overdue\_time = now\_time - time\_offse

        for file in os.listdir(self.log\_folder):

            if file.endswith('.log'):

                file\_ctime = os.path.getctime(os.path.join(self.log\_folder, file))

                #检查是否早于过期时间

                if file\_ctime < overdue\_time.timestamp():

                    os.remove(os.path.join(self.log\_folder, file))

    def get\_logger(self):

        """

        根据config.yaml配置文件进行日志对象的获取与初始化配置

        """

        logger = logging.getLogger(\_\_name\_\_)

        #配置logger

        if not logger.handlers:

            loglevel = logging.DEBUG

            logger.setLevel(loglevel)

            #设置日志格式

            log\_format = logging.Formatter(

                '%(levelname)s - %(asctime)s - %(filename)s:%(lineno)d -[%(module)s:%(funcName)s] - %(message)s')

            """INFO - 2025-05-20 11:58:35,604 - assertions.py:198 -[assertions:assert\_result] - 测试成功"""

            fh = RotatingFileHandler(filename=self.logfile\_name,

                                    mode='a',   #追加

                                    maxBytes=configLoader.LOG\_MAXBYTES,

                                    backupCount=configLoader.LOG\_BACKUPCOUNT,

                                    encoding='utf-8')

            fh.setFormatter(log\_format)

            fh.setLevel(loglevel)

            #设置日志控制台输出

            stream\_loglevel = configLoader.LOG\_STREAM\_LEVEL

            sh = logging.StreamHandler()

            sh.setLevel(stream\_loglevel)

            sh.setFormatter(log\_format)

            #载入日志处理器

            logger.addHandler(fh)

            logger.addHandler(sh)

        return logger

该日志类存放于./utils/recordlogs.py中

#### 日志的记录

基于上面的日志类，我们通过类函数get\_logger来获取实例化的日志对象就可以进行日志记录了：

recordlogs = RecordLogs(log\_folder)

logs = recordlogs.get\_logger()

在其他模块中，我们通过引入logs对象即可实现日志记录：

from utils.recordlogs import logs

…

logs.info(f'接口名称：{api\_name}')

logs.info(f'接口地址：{full\_url}')

logs.info(f'接口方法：{method}')

logs.info(f'请求头：{headers}')

logs.info(f'用例名称：{case\_name}')

或异常捕获时：

logs.error(f"MySql连接异常:{e}")

需要注意区分的是：

* logger.setLevel(...)

日志系统的总门槛，如果小于这个等级，不会进入系统，也不会被任何 Handler 处理

**控制：**哪些等级的日志“可以被接收”

* handler.setLevel(...)（比如 fh.setLevel(...) 或 sh.setLevel(...)）

即使日志进来了，也可能因为等级太低不被某个 Handler 输出

**控制：**这条日志是否输出到这个“具体通道”

**常见策略**：

* 控制台只输出 INFO 以上（日志干净）
* 文件输出 DEBUG 以上（内容详细）
* logger.info(...) / logger.debug(...) / logger.error(...)：

这是你设置的具体日志记录语句

决定：这条日志属于哪个等级（INFO/DEBUG/ERROR）

### 异常处理

异常处理是pytest进行测试的重要机制，pytest通过异常来判断用例执行的错误并且注意处理最终的异常，断言用例的对错也是通过触发异常来“让pytest”知道用例出现错误。

所以我们需要做的是什么呢？

我们需要做的不是拦截异常防止出错，而是捕获异常再抛出或者主动触发异常来先记录用例失败的原因再提供错误给pytest，达到更详细的记录失败的信息来更好的用例的执行。

def excute\_test(self,base\_info,test\_case):    """

    try:

        ……

    except Exception as e:

        logs.error(e)

        raise e

    #断言只做判断,不错异常处理,直接决定用例是否通过,日志在内部记录

    try:

        ……

    except Exception as e:

        logs.error(f'用例【{api\_name}--{case\_name}】执行失败')

        raise e

所以异常处理并不是“异常处理”，而是异常记录，因为如果直接到最后面让pytest来记录的话针对性不强、可读性不行、日志无法记录，所以通过中间拦截+日志记录就是一个比较可行的方式。

## Allure报告集成

“**Allure 是一个开源的、功能强大的测试报告生成工具**，用于将测试执行过程中的各种信息（如用例步骤、断言、截图、日志等）以**图形化的方式展示**出来。”

也就是说Allure可以自动收集我们记录的内容来展示测试报告，与日志不同的是，日志我们会记录很多的中间信息，包括一些工具函数的处理以及异常内容什么的，而我们主动记录的allure主要是测试用例的信息、接口请求的信息以及断言的信息，此外，allure执行测试用例时是可以主动捕获用例运行时的日志的，所以日志其实也会成为Allure报告的一部分。

### Allure+Pytest原理

Pytest使用allure的原理是：在执行pytest运行语句时带上：“--alluredir=”字段指定一个allure记录内容的临时目录，然后我们再记录测试的allure报告内容，这些内容就会通过特定的格式存储到这个文件夹中，随后用例执行完成后，通过

'allure generate {ATD} -o {ARD} --clean'

就可以把ATD的临时记录内容转化成ARD的持久内容，然后再通过

'allure open {ARD}'

就可以开启allure服务器查看测试报告了。

### Allure报告级别

Allure的内容级别主要是：

Epic（可选）→ Feature → Story → Title→ Step + Attachment

这些级别类似树状结构来构造整体的报告级别，表示从属关系。

* Epic

表示一个整体的大项目

* Feature

表示一个功能模块，例如用户模块

* Story

表示具体的一个接口功能，例如用户模块可能有注册接口、登录接口

* Title

表示具体的一个测试用例的标题，例如用户登录可能有正常登录、错误密码等这样的一些更具体的用例。

* Step

接口测试中主要用于多接口联合测试中，用于区分不同的执行步骤，此时一个step就是一个具体接口

* Attachment

也就是具体的内容记录，可以是文本、截图等，例如记录具体的请求信息、测试结果等。

此外Allure的日志级别记录还包含装饰器静态设置以及dynamic动态设置

@allure.feature()、@allure.title()------allure.dynamic.feature()，主要区别在于前置只能针对函数级别的添加，例如我为这个接口专门设置的一个测试类，其是强绑定固定的函数的，而后者更加灵活，可以实现更灵活的数据驱动的的标题写入。

### Allure报告记录

上面的级别配置主要是用于区分各个功能块、用例，而Allure的用例具体内容记录是通过allure.attach:---allure.attach(body, name=None, attachment\_type=None, extension=None)这里的body就是具体的内容，name相当于一个标题，type用于表明body的数据类型，extension表示附件拓展名；例：

allure.attach(json.dumps(headers, indent=2, ensure\_ascii=False), '接口请求头', allure.attachment\_type.JSON)

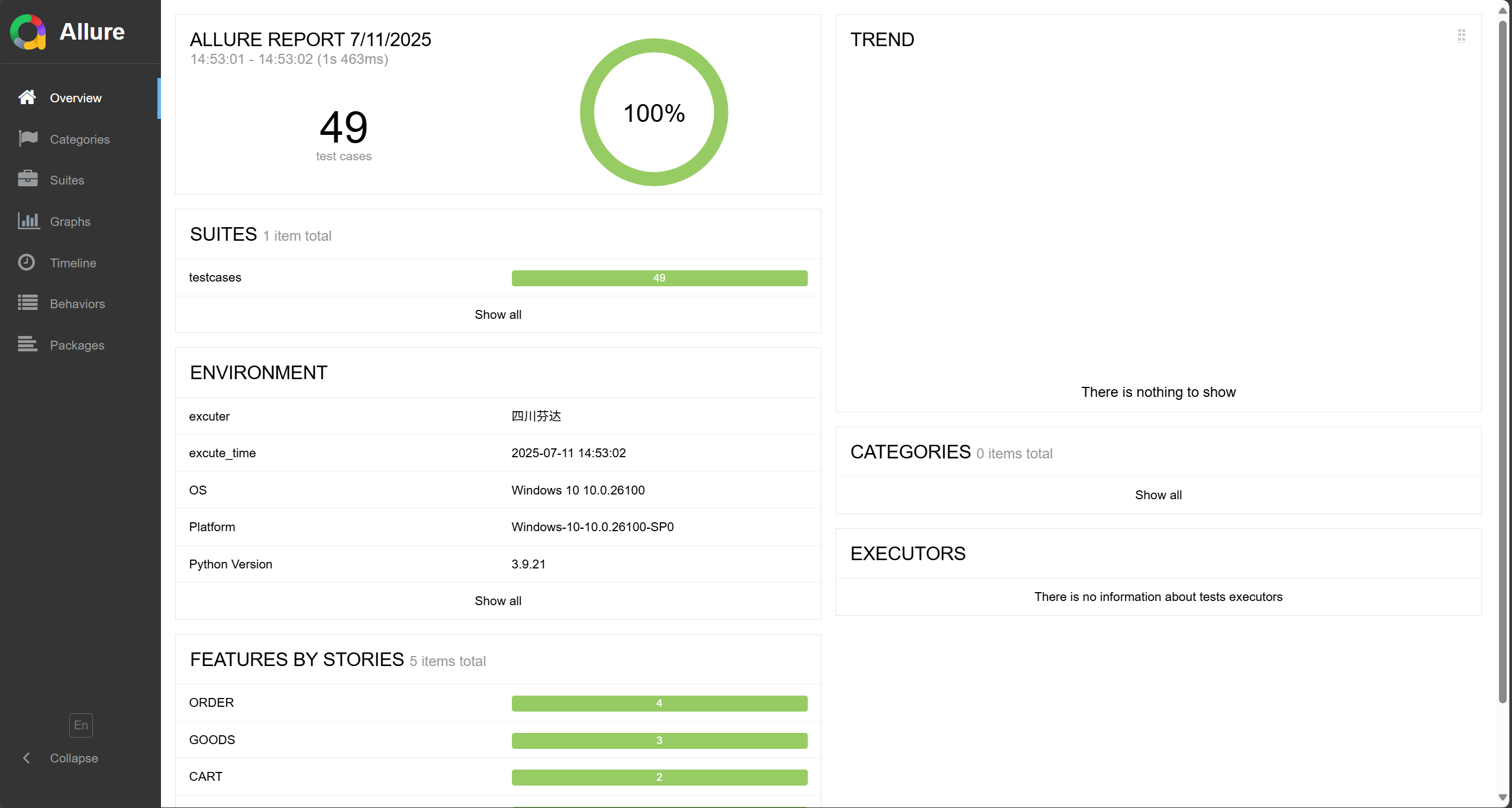
type有：

|  |  |
| --- | --- |
| **类型** | **描述** |
| TEXT | 文本文件（默认） |
| JSON | JSON 数据 |
| HTML | HTML 代码片段 |
| MARKDOWN | Markdown 内容 |
| XML | XML 内容 |
| PNG | PNG 图片 |
| JPG | JPG 图片 |
| SVG | SVG 矢量图 |
| CSV | CSV 文件 |
| PDF | PDF 文件 |
| MP4 / WEBM | 视频 |
| LOG | 日志文件 |

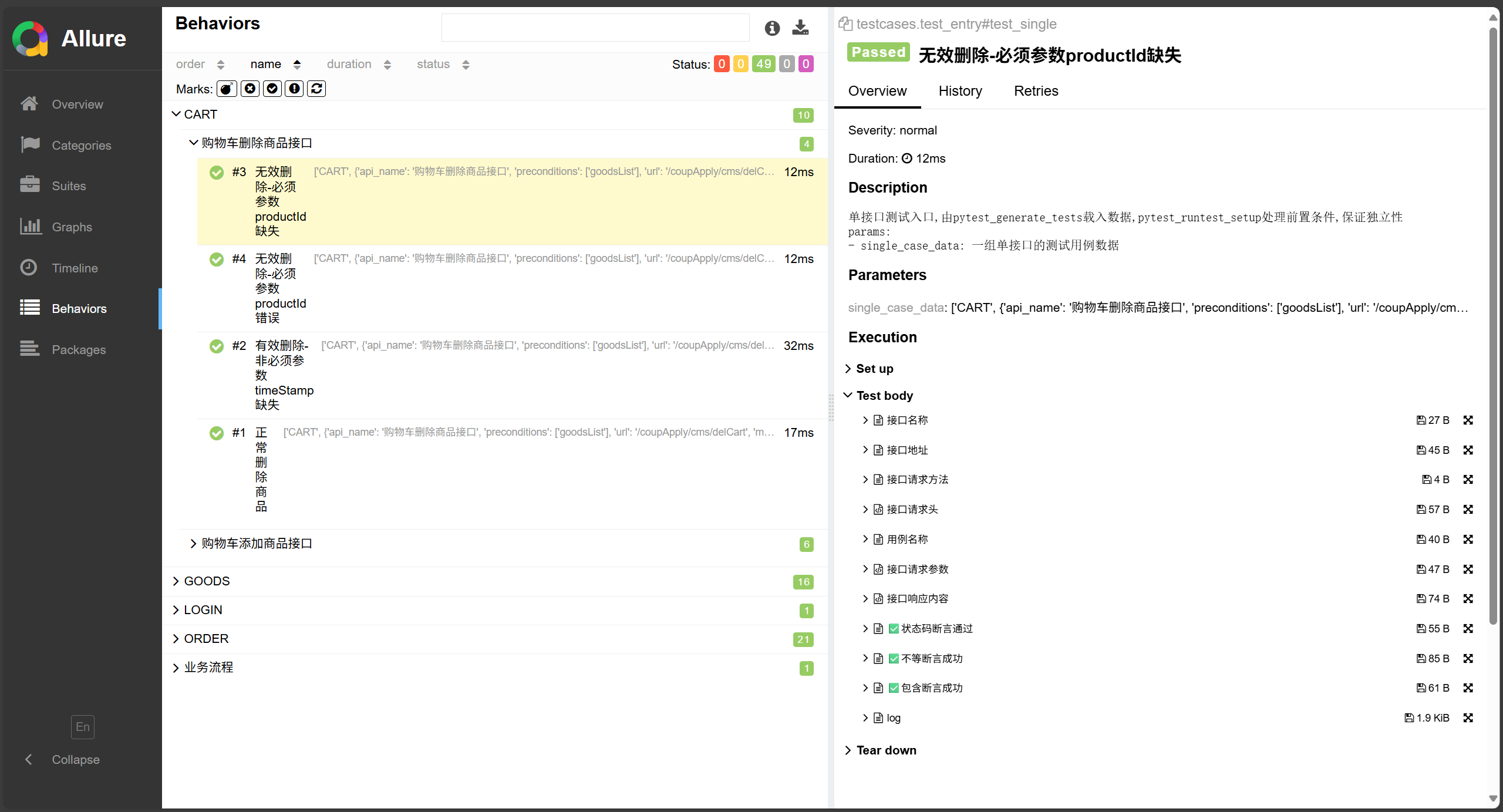
这样就可以在用例执行的过程中记录需要在报告展示的一些报告信息了。

### Allure报告展示

运行“allure open”等待处理报告内容即可自动打开web页面，整体页面如下



点击behaviors栏可以得到更加详细的具体用例内容的展示，这里的CART对应的就是feature，而xxxx接口就是对应的story，下面具体的就是title，然后就是具体的报告内容，其中logs是其自动捕获的日志内容，其余是通过attachment记录的内容。



下面是通过step来指定步骤的情形，其中这里“正常提交订单”、“正常支付订单“就是各个step，内部也是各种的attachment



## 配置文件全局管理

目前为止，其实整体的测试框架已经可以运行，有基本的数据驱动有报告生成了，下面的内容其实都是进一步的优化。全局文件配置的想法是这样的：既然我们用例都是数据驱动的了，我们也可以把一些全局的配置都抽离出来以数据驱动的方式统一配置。

### 能够抽离的配置

能够抽离的配置内容可以是所有的使用第三方能够指定的参数，但是根据具体实际而已，主要是把一些常用、不同环境常转化的内容进行抽离，包括但是不限于：

批次数量、是否发送钉钉、是否发送邮件、是否运行前清理临时文件、extract.yaml的文件path、allure的临时路径、allure持久路径、日志级别、日志最大大小、日志备份数量、数据库账号密码等等。

### 全局配置管理

确定需要抽离的配置后，把配置内容通过yaml或者其它文件存储，最终使用时只需要更改这个文件即可，而有了这个配置文件（例如叫conf.yaml）后，我们还需要通过配置加载类来把内容加载出来，整体的原理其实就和读写extract文件基本一致。整体的思路是先把所有的配置内容读出成全局变量，然后其它模块直接使用，这样在项目初始化时就会处理一次性加载好，不需要每次使用动态读取，缺点就是这些配置变量生命周期较长。

需要注意的是，例如日志级别这种特殊类型需要额外处理转化。

下面是一个示例的配置加载类：

class ConfigLoader:

    """

    配置加载类,提供数据接在和一些特殊配置的加载

    """

    def \_\_init\_\_(self, config\_file = r"conf\config.yaml"):

        self.config\_file = config\_file

        self.batch\_list = self.load\_data('file','batch\_data')

    def load\_data(self,fst\_node, snd\_node = None, thd\_node = None,file = None):

        """

        类似readyaml的数据读取,本质为数据加载为字典后进行访问,此处提供最多三级的节点访问。

        """

        if file is None:

            file = self.config\_file

        try:

            with open(file, 'r', encoding='utf-8') as f:

                extract\_data = yaml.safe\_load(f)

                if snd\_node is None and thd\_node is None:

                    return extract\_data[fst\_node]

                elif snd\_node is not None and thd\_node is None:

                    return extract\_data[fst\_node][snd\_node]

                else:

                    return extract\_data[fst\_node][snd\_node][thd\_node]

        except Exception as e:

            return None

    def get\_log\_level(self, level\_name):

        """

        获取日志级别,由于不是字符类型,需要getattr进行转换

        """

        level\_str = self.load\_data("logs", level\_name)

        return getattr(logging, level\_str.upper(), logging.INFO)

    def dynamic\_load\_batch(self):

        """

        动态加载batch数据,避免不同批次懒加载

        """

        batch\_num = self.load\_data('batch\_num')

        batch\_kinf,batch\_data = self.batch\_list[batch\_num-1]

        return batch\_kinf,batch\_data

此处包含简单的文件读取、log级别的string转化、批次的动态加载。

## 工作流程的多接口用例执行

目前来说，还无法处理多接口的用例执行，下面将引入多接口的处理，本质上类似allure多一个step标签，多接口的执行其实就是吧原本一个接口的执行拓充变成多个接口的执行，但是这意味着我们可能需要新的一套数据驱动格式，我们知道，原本的这个数据驱动是一个yaml文件表示一个接口的多个用例，而现在我们需要一个用例由多个接口组成，一种常见的解决策略是为多步骤的创建一种新的yaml格式，然后对应调整一个新的测试入口函数，但是单个接口的执行是可以保持一致的：

### yaml：

需要注意的是，由于feature、story无法再根据具体一个接口的baseinfo来取出，一次可以加上一个descrip或者依旧使用baseinfo字段来指定，然后此处内部的steps就是各个步骤，本质上是与单接口一致的。

descrip:

  feature\_name: 业务流程

  story\_name: 提交订单-支付订单

steps:

  - baseInfo:

      api\_name: 提交订单接口

      url: /coupApply/cms/placeAnOrder

      method: POST

      headers:

        Content-Type: application/json;charset=UTF-8

    testCase:

      case\_name: 正常提交订单

      json:

        goods\_id: '18382788819'

        number: 2

        propertyChildIds: "2:9"

        inviter\_id: 127839112

        price: "128"

        freight\_insurance: "0.00"

        discount\_code: "002399"

        consignee\_info:

          name: "张三"

          phone: 13800000000

          address: "北京市海淀区西三环北路74号院4栋3单元1008"

      validation:

        - status\_code: 200

        - equal: { 'error': ''}

        - equal: { 'error\_code': '0000' }

      extract:

        orderNumber: $..orderNumber

        userId: $..userId

  - baseInfo:

      api\_name: 订单支付接口

      url: /coupApply/cms/orderPay

      method: POST

      headers:

        Content-Type: application/json;charset=UTF-8

    testCase:

      case\_name: 正常支付订单

      json:

        orderNumber: ${get\_extract\_data('orderNumber')}

        userId: ${get\_extract\_data('userId')}

        timeStamp: 169232689

      validation:

        - status\_code: 200

        - equal: { 'error': ''}

        - equal: { 'error\_code': '0000' }

### 入口函数：

入口函数整体基本与单接口一致，组主要是加上step的循环执行逻辑

def test\_workflow(workflow\_case\_data):

    #解包成descrip描述信息和steps步骤信息

    descrip,steps = workflow\_case\_data

    allure.dynamic.feature(descrip['feature\_name'])

    allure.dynamic.story(descrip['story\_name'])

    for case\_data in steps:

        with allure.step(case\_data['testCase']['case\_name']):

            base\_info = case\_data['baseInfo']

            test\_case = case\_data['testCase']

            RequestBase().excute\_test(base\_info,test\_case)

### 可能的优化

一种更好用的可能的优化思路是：把单接口向多接口统一而不是多接口向单接口统一，因为单接口其实就是多接口执行的只有一步的特殊情形，不过整体的baseinfo的管理可能需要考虑一种更加普适的方案。

## conftest钩子优化架构

其实前面的入口函数可能已经有点看不懂，我是输入注入用例的，其实我就是通过pytest更底层的钩子函数来进行处理，pytest运行的生命周期其实是有很多钩子来构建流程的：

pytest\_addoption/configure

↓

pytest\_sessionstart

↓

pytest\_collection

↓

pytest\_generate\_tests

↓

pytest\_collection\_modifyitems

↓

pytest\_runtest\_setup

↓

pytest\_runtest\_call

↓

pytest\_runtest\_teardown

↓

pytest\_runtest\_logreport

↓

pytest\_sessionfinish

↓

pytest\_unconfigure

其中用到的有，pytest\_sessionstart：测试会话开启前的钩子，用于在一次整体测试前执行；pytest\_generate\_tests：pytest的用例收集函数，因为pytest是先收集所有的测试用例，获得一个确定的测试用例列表再一个个执行用例的；pytest\_runtest\_setup：**每个**测试用例执行前执行的钩子；pytest\_sessionfinish：所有用例执行完后执行。

### pytest\_sessionstart

这个钩子的主要作用是进行一些运行前的清理操作，例如上一次测试可能依旧存在一些allure的临时文件，一次新的执行肯定是需要清理先前的内容的，避免之前内容导致的冲突，此外，启用mock服务器也是在这个钩子内进行：

def pytest\_sessionstart(session):

    if configLoader.IF\_BEFORE\_CLEAN:

        YamlDataProcess().clean\_extract\_data()

    if configLoader.IF\_MOCK:

        from mock.mock\_server import MockServer

        MockServer().thread\_run()

### pytest\_generate\_tests

pytest的用例执行是先收集所有的测试函数，然后通过这个generate钩子来把用例一个个注入到用例入口，所以我们要做的就是主动加载所有的用例内容，然后再注入到需要的入口函数中：

def pytest\_generate\_tests(metafunc):

    """

    参数化钩子函数，在收集测试用例阶段调用，动态生成测试用例

    metafunc代表测试函数的元信息，.fixturenames代表其参数名列表

    """

    cl = ConfigLoader()

\_,batch\_data = cl.dynamic\_load\_batch()

    if "single\_case\_data" in metafunc.fixturenames:

        cases = batch\_load\_testcases(batch\_data)

        metafunc.parametrize("single\_case\_data", cases)

    if "workflow\_case\_data" in metafunc.fixturenames:

        cases = batch\_load\_testcases(batch\_data)

        metafunc.parametrize("workflow\_case\_data", cases)

这里的metafunc表示一个测试函数的元信息，fixturesnames表示参数列表；是通过这个钩子，其匹配到对应的内容（此处是通过参数来进行匹配）后，通过自定义的用例加载函数来把所有的用例列表加载后再进行参数化的注入。

用例加载函数：

其实就是把所有的yaml都读取出来整理成一个列表。

def batch\_load\_testcases(file\_path):

    batch\_cases\_list = []

    for file in os.listdir(file\_path):

        batch\_cases\_list.extend(read\_testcase\_from\_yaml(os.path.join(file\_path, file)))

    return batch\_cases\_list

整体的原理就是：pytest根据命名等规则收集到所有的测试函数，例如现在有test\_single(single\_case\_data)和test\_workflow(workflow\_case\_data)，构成一个metafuncs的列表，然后这些metafunc一个个注入generate\_tests钩子，进入钩子后根据参数列表判断是哪个类型的用例入口，然后加载参数化的用例列表进行注入，然后再到下一个metafunc重复一次流程，这样就收集加载完成所有的test item，供后面执行。

### pytest\_runtest\_setup

前面已经收集到了所有的item，下面就是一个个执行用例了，需要注意的是，此时item是已经包含参数化的数据的，而且这一步执行在用例正式执行前，因此我们可以通过这个钩子来处理用例的前置条件。

我的处理策略是：在用例的baseinfo加上“preconditions“字段用来指定前置条件的处理，然后通过一个前置条件模块和前置条件的yaml来预先把前置条件指定好，这样每次运行用例前就会先执行一次标定的前置条件，保存好上下文数据供正式执行使用，这样也确保了测试用例的独立性。

* **前置条件执行类：**

本质上其实也与正式执行一样，知识通过字段来执行固定的用例内容，而且不需要记录一些中间内容

class Precondition:

    def \_\_init\_\_(self):

        self.yaml\_data\_process = YamlDataProcess()

        with open(configLoader.PRE\_DATA, 'r',encoding='utf-8') as f:

            self.pre\_data = yaml.safe\_load(f)

    def preconBase(self,base\_info,test\_case):

        try:

            #base\_info处理

            api\_url = base\_info['url']

            method = base\_info['method']

            headers = base\_info['headers']

            #test\_case--case\_name处理用例名称

            test\_case.pop('case\_name',None)

            #test\_case--extract处理上下文存储数据

            extract = test\_case.pop('extract',None)

            extract\_list = test\_case.pop('extract\_list',None)

            #执行requests请求

            res = requests.request(url=configLoader.API\_HOST+api\_url,

                                    method=method,

                                    headers=headers,

                                    verify=False,

                                    \*\*test\_case)

            #extract上下文写入

            if extract:

                self.yaml\_data\_process.extract\_data(extract,res)

            if extract\_list:

                self.yaml\_data\_process.extract\_data\_list(extract\_list,res)

        except Exception as e:

            logs.error(e)

            raise e

    def precon\_entry(self,pre\_name):

        """

        根据字段内容,动态执行precon.yaml内的前置条件请求

        """

        logs.info(f"执行前置条件:{pre\_name}")

        if self.pre\_data.get(pre\_name,None):

            login\_data = copy.deepcopy(self.pre\_data[pre\_name])

            self.preconBase(login\_data['baseInfo'],login\_data['testCase'])

        else:

            logs.error(f"不存在【{pre\_name}】的前置条件")

* **钩子：**

钩子中取出对应的字段，然后通过前置条件类执行即可

def pytest\_runtest\_setup(item):

    if hasattr(item, "callspec"):   #callspec标识参数规格pytest.CallerSpec对象，只有参数化才会有

        case\_data = item.callspec.params.get("single\_case\_data", [])

        if case\_data:

            \_, base\_info, \_ = case\_data

            preconds = base\_info.get("preconditions", [])

            if preconds:

                for precond in preconds:

                    global\_prec.precon\_entry(pre\_name=precond)

需要注意的是，目前多接口的流程测试没有设置好前置条件的处理功能，可能的思路就是在“describe“字段内也加上”preconditions“，整体不复杂。

### pytest\_sessionfinish与pytest\_terminal\_summary

这两个钩子都是主要负责善后工作，前置对应pytest\_sessionstart，后置是负责一些控制台内容的输出：

def pytest\_runtest\_setup(item):

    if hasattr(item, "callspec"):   #callspec标识参数规格pytest.CallerSpec对象，只有参数化才会有

        case\_data = item.callspec.params.get("single\_case\_data", [])

        if case\_data:

            \_, base\_info, \_ = case\_data

            preconds = base\_info.get("preconditions", [])

            if preconds:

                for precond in preconds:

                    global\_prec.precon\_entry(pre\_name=precond)

def pytest\_sessionfinish(session, exitstatus):

    if configLoader.IF\_AFTER\_CLEAN:

        YamlDataProcess().clean\_extract\_data()

#钩子：测试结果总结处理

def pytest\_terminal\_summary(terminalreporter, exitstatus, config):

    #生成测试结果摘要字符串

    total = terminalreporter.\_numcollected

    passed = len(terminalreporter.stats.get('passed', []))

    failed = len(terminalreporter.stats.get('failed', []))

    error = len(terminalreporter.stats.get('error', []))

    skipped = len(terminalreporter.stats.get('skipped', []))

    duration = time.time() - terminalreporter.\_sessionstarttime

    summary = f"""

    自动化测试结果，通知如下，请着重关注测试失败的接口，具体执行结果如下：

    测试用例总数：{total}

    测试通过数：{passed}

    测试失败数：{failed}

    错误数量：{error}

    跳过执行数量：{skipped}

    执行总时长：{duration}

    """

    print(summary)

    #执行邮件、钉钉等管理通知操作

    if configLoader.IF\_SEND\_EMAIL:

        pass

    if configLoader.IF\_SEND\_DD:

        pass

## 分批次运行与报告合并（未完全解决）

这其实出自一个考虑：如果一个系统进行测试的内容过多，而pytest需要获取所有的测试内容后才会执行，这样会导致内存不足等问题，因此一个常见的解决策略就是通过分批次运行，也就是我的run.py中执行多次pytest.main来运行框架进行测试，每次执行不同的用例内容；然后由于不同批次的运行会得到不同的不同的报告，如何合并报告内容来进行一次性展示也是一个需要考虑的问题。

### 分批次运行

分批次最主要的问题是，需要让本次测试的session知道目前属于哪个批次，一个简单的处理方式就是run.py来控制，然后写入文件持久化管理，每次session就可以通过读取文件的内容来知道属于的批次，然后使用对应的数据了。

需要注意的是，我们需要在conf.yaml中设置好各个批次的数据路径等

### 报告合并

报告合并其实是与目前的处理有一定的冲突了，因为allure报告记录是根据函数来命名或者标识，这意味着两个批次都通过这个统一入口，第一个批次可能有uuid是a1、a2，而第二个由于也有a1、a2就会导致汇总时导致相互覆盖，而且uuid不是简单通过文件名标识。之前的折中解决策略是把单接口的运行和多接口的运行分开，这样就不会由于两个批次使用同样的测试函数导致uuid容易导致冲突。

* **可能的解决策略：**

因为之前没有了解到还有allure.dynamic.id()所以没有解决上面的问题，目前知道有这个函数后似乎可以再title级别进行uuid的自定义，这样可以一次运行多个批次使用同一个临时目录然后直接汇总而不冲突。但是还没有尝试过，不过是一个理想的解决策略。

## Mock骨架实现

Mock就是“模拟”，在接口测试中，主要就是用于对于一些无法处理的第三方数据，或者后端还未完成的情形，pytest本身提供mock插件，但是似乎局限较大，只能针对函数级别，对于这种高抽象程度的用例并不适合，因此我们写一个简单的服务器来进行mock的实现。

我的做法是通过配置文件设置是否启用mock，启用则把所有的接口请求都先发送到mock服务器，然后mock服务器进行内容的匹配，如果匹配到需要mock的内容就把设定好的mock结果进行返回，否则就替换回原的url，重新发送请求到原本的服务器再把这个真实结果返回给测试框架。

* **mock服务器类：**

在得到请求后其先与mock.yaml进行匹配，检查是否有需要mock的内容

class MockServer:

    """

    基于flask实现的mock server,根据yaml进行模拟信息配置，

    """

    def \_\_init\_\_(self, config\_path="mock/mock.yaml", host=configLoader.MOCK\_HOST, port=configLoader.MOCK\_PORT):

        self.config\_path = config\_path

        #本地启用的mock服务器的入口

        self.host = host

        self.port = port

        self.app = Flask(\_\_name\_\_)                  # 创建flask app实例

        self.mock\_routes = self.load\_mock\_config()  # 加载mock配置

        self.setup\_routes()                         # 设置URL路由

    def load\_mock\_config(self):

        """

        加载mock配置

        """

        with open(self.config\_path, "r", encoding="utf-8") as f:

            return yaml.safe\_load(f)

    def match\_request(self, mock, req):

        """

        单个mock匹配规则,可以根据实际需求精细化匹配

        """

        if mock["method"].upper() != (req.method).upper():

            return False

        if mock["url"] != req.path:

            return False

        return True

    def setup\_routes(self):

        """

        通过@app.route注册路由规则

        """

        @self.app.route("/<path:path>", methods=["GET", "POST", "PUT", "DELETE"])

        def catch\_all(path):

            for mock in self.mock\_routes:

                #进行内容匹配

                if self.match\_request(mock, request):

                    return (

                        jsonify(mock["response"]["body"]),

                        mock["response"].get("status\_code", 200)

                    )

            # 没匹配到，代理转发请求到真实服务器

            url = f"{configLoader.API\_HOST}/{path}"

            method = request.method

            # 根据请求方法，转发请求并拿到响应

            try:

                resp = requests.request(

                    method=method,

                    url=url,

                    headers={key: value for key, value in request.headers if key != 'Host'},  # 避免Host冲突

                    params=request.args,

                    data=request.get\_data(),

                    cookies=request.cookies,

                    allow\_redirects=False,

                )

                # 构造Flask响应才可以返回给客户端，否则flask无法解析返回内容

                excluded\_headers = ['content-encoding', 'content-length', 'transfer-encoding', 'connection']

                headers = [(name, value) for name, value in resp.raw.headers.items()

                        if name.lower() not in excluded\_headers]

                response = Response(resp.content, resp.status\_code, headers)

                return response

            except Exception as e:

                return jsonify({"error": f"代理请求失败: {e}"}), 502

        # 查看所有 mock 配置

        @self.app.route("/mock/configs", methods=["GET"])

        def show\_configs():

            return jsonify(self.mock\_routes)

    def server\_run(self):

        logs.info(f"MockServer running at http://{self.host}:{self.port}")

        self.app.run(host=self.host, port=self.port)

    def thread\_run(self):

        """

        启动mock server的线程

        """

        thread = Thread(target=self.server\_run, daemon=True)

        thread.start()

        return thread

mock.yaml文件

该文件设置好匹配的内容以及返回内容

- name: 登录

  method: post

  url: /dar/user/login

  json:

  data:

  response:

    status\_code: 200

    body:

      error\_code: null

      msg: 登录成功

      msg\_code: 200

      orgId: "4140913758110176843"

      token: "13Db9f9f2A0Bf0ee583AB3E51a7eF"

      userId: "3501704179608244733"

当然这只是一个半成品的简单mock，实际中可能会只用更加复杂的MITM等mock策略。

# 项目问答

## 框架设计与模块解耦

### 你为什么选择用 Pytest，而不是 unittest 或 nose？它对你的框架有什么优势？

1. Pytest的功能更强大、使用更广泛、框架拓展性更强、功能更灵活，例如conftest的各个级别的测试配置和强大的钩子函数等，都使得其比unittest在整体的大规模测试中更加好用。
2. 相比较于pytest，unitest和nose都相对落后，例如unittest代码较为固定且前置条件等处理相对复杂；而nose虽然部分优于unittest，但是其更新和维护已经停止，很难有进一步发展。
3. 说白了其实就是因为pytest广泛使用、接受度高，我在学习测试开发的过程中才会学习到这个框架；而其优势就我来看主要就是那么几点：
4. 灵活，能够实现很多自定义的功能，函数与测试的命名也更加自由
5. **强大的钩子函数**，对于初学者或者简单的使用可以使用fixture、parameterize装饰器来实现前置依赖和参数化，而复杂内容则可以通过内部的generate\_tests和test\_setup这种更底层的钩子进行处理。
6. 集成难度低，一方面其提供灵活的插件、一方面代码灵活度高，使得其集成其它的插件或者自定义功能容易很多，可以快速、便捷地实现内容的扩展。能够支持各种测试类别。

### 你的框架中是如何实现测试数据与逻辑解耦的？具体通过什么方式加载 YAML？

数据与逻辑的结构其实就是把用例的信息例如URL、请求头、请求体、需要保存的上下文等信息都写到yaml文件中存储，而框架只负责根据规范的格式来把内容读取出来，然后执行请求、保存上下文、断言。

更具体的说，就是把所有的用例按照需要的区分写到yaml中，框架中我是使用genenrate\_tests来主动载入用例，也就是框架出发这个钩子时会把匹配到的测试入口函数进行内容的匹配，我的做法是通过自定义的参数进行匹配，匹配到通过一个函数（基于yaml.safeload加载yaml数据）把yaml中的数据加载成一个用例数据的字典列表，然后注入到测试函数中，这样就得到了一个个的测试的item用于执行。这样就实现了外部数据导入的数据驱动。

### 请你详细说说 pytest\_generate\_tests 在你框架中的作用，它在哪个阶段生效？为什么选它？

generate\_tests函数生效在框架根据设定的文件夹、命名方式收集完测试函数后，其会得到一个metafunc的列表然后注入到钩子中，在这个阶段，我会人为注入测试数据，也就是通过metafunc.parametrize（）来实现人为的参数化注入，这一步后框架才会得到接收的参数得到的item。

选择这个钩子其实也是学习时歪打正着，因为开始时使用的也是parameterize来进行参数化，后来为了实现可以动态变化数据源而采用了这个钩子，也就是用其实现动态参数化加载，此外这个钩子内部也能实现一些更复杂的操作，而parameterize参数化相较而言不够灵活；

### 你有没有实现前置条件复用，比如登录、鉴权？是用 fixture、钩子，还是代码封装实现的？

我的前置条件是通过钩子+代码封装实现的，也就是我依赖yaml中的preconditions字段来指定一些固定的前置条件操作，通过runtest\_setup钩子在正式执行用例前根据该字段到封装好的前置条件类中先执行一次前置条件，确保上下文信息更新好。

所以其实我是没有实现前置条件复用的，这是因为我在做我的项目练手时出现一个问题：一个接口会有多个用例，在执行这个接口前只执行一次前置条件，然后用例的顺序是有效、有效、无效、无效、有效或者再下一个接口也依赖这个前置条件时，此处的无效用例会导致前置条件无效化，也就是出发服务器机制导致这个前置条件失效了。因此我才采用现在的方法，保证用例间的独立性、解耦，也有效避免这种情况的发生。

当然，一个更好的解决策略是根据需求可以调整是否重复执行还是相同接口只执行一次。

### 你的请求封装模块中是否支持自定义请求头、超时、重试机制？设计上如何避免耦合？

当然是支持自定义的，请求头通过yaml中对用例请求的配置即可实现，而超时、重试等可以通过全局的配置文件conf.yaml结合confloader模块实现加载，这样的配置加载都是从外部数据，首先实现用例和配置数据与执行逻辑的解耦，其次执行逻辑仅做请求的调用，请求类也只负责读取配置数据、接收执行主逻辑提供的数据记录后执行请求。

简单来说，就是我的执行主逻辑只负责解析用例数据和调用请求类、上下文提取类、断言类等，各司其职，只负责整合功能进行用例执行。

## 用例设计与执行控制

### 如何通过配置或参数控制执行不同环境（dev/test/prod）下的接口？

我的框架是天然支持测试环境切换的，只需要再conf.yaml中制定好当前环境dev/test等和对应的base\_url即可快速实现环境接口的切换；此外我的项目目前也提供了mock服务器骨架，通过MITM技术实现接口数据模拟。

### 框架中是否支持用例失败重跑？如果有，是用插件实现还是自定义逻辑？

目前暂未实现用例的失败重跑，但是我了解到有一个比较成熟的插件解决策略，就是通过pytest-rerunfailures插件，只需要在运行参数中加上“—reruns=”和“—reruns-delay=”就可简单实现重跑机制，且其与allure能够良好配合。此外，也可以通过pytest\_runtest\_makereport来检查失败用例来构造重跑队列实现重跑。

### 如果一个用例依赖另一个接口返回的 token 或 ID，你是如何管理这些上下文变量的？

我通过extact.yaml文件来持久化存储上下文变量，而用例数据中通过extract或extract\_list字段来表明需要保存的字段。然后当其它接口用例依赖这些上下文时，通过${}的表示让主逻辑进行用例信息提取时进行数据的替换。

目前来说我的框架是独立的前置条件处理，流程是这样的：执行用例前先执行所有前置条件-保存上下文到extract.yaml-执行用例-用例中包含有${}要求要取出extract.yaml中的某个值-取出替换，这样就完成了上下文内容的共享。

### 你有没有处理接口依赖问题，比如某接口必须先执行其他接口才能测？如何保证用例的独立性？

最开始时我们通过空值接口执行的顺序和extract的写入来实现接口上下文的处理，后来发现存在接口间相互依赖导致耦合的风险，于是通过runtest\_setup钩子来单独处理前置条件，在前置条件的独立执行中写入extract上下文，这样用例执行时就可以保证前置条件的独立完成和上下文的存在。

### 如何控制测试用例的并发或批量分组运行？有没有做过分批执行和报告合并？

我的框架目前还不支持并发执行，但是支持分批次的执行：同样通过conf.yaml能够修改各个批次组别的测试用例数据来源，而不同批次的测试session通过conf.yaml中动态变化的批次号来识别目前执行的。

pytest+allure是需要指定一个该session的报告内容临时目录的，其实一次运行中每个批次使用同一个临时目录是没关系的，最主要的问题其实在于uuid的处理——因为我使用的是高抽象级别的用例入口，而allure默认的定义uuid的策略大概是函数+序号，所以开始时发现不同批次都用同一个入口函数会导致汇总时出现用例间的冲突，当前的解决策略是把单接口和多接口进行批次拆分，这样入口函数不同也就不会出现冲突；

后来了解到的一个更好的解决方法是通过allure.dynamic.id()和uuid库来人为指定allure用例的编号来进行区分，这样就不会导致冲突了。

### 你是如何对测试用例进行分层管理的？是否区分了核心用例、冒烟用例、回归用例？

我的框架通过分批运行获知执行修改数据源可以天然支持分层管理；数据层可以简便地通过内容的文件组织来实现用例级别的管理，进一步拓展来说的话，可以在session的结束钩子加上一个失败用例的收集，来添加到回归用例中。

### 有没有设计用例标签机制，比如用 @pytest.mark.xxx 实现筛选？

由于我的框架使用的是整体的统一用例入口和数据驱动，所以是不适合通过装饰器来进行标签机制的，如果需要使用标签功能其实也很容易，在yaml数据多加一个mark字段然后自定义个mark类来把标签处理进行封装管理即可。

### 如果某个接口在执行中频繁抛错影响其他用例，你会如何隔离这类不稳定用例？

一般来说我由于进行前置条件单独处理，用例间的独立性是比较强的，一般不会相互影响；如果确实存在这种情况，一种常见的解决方式就是通过日志查找错误原因解决问题，或者是直接跳过这个用例。

### 如果断言失败，但实际是字段顺序变化导致，如何设计容错机制？

我目前的包含或者相对这些断言的执行是不受字段顺序处理的，因为是单值多次进行匹配，也就是一次只会进行一个字段的断言，多次执行，因此不是一次性大批量的处理，所以不会收到顺序影响。

如果是json scheme断言的话，就比较容易出现这种问题，这样的话就需要进行专门的针对性处理。

## 日志与报告系统

### 你是如何设计日志系统的？日志的输出格式、级别、保存策略具体是如何配置的？

日志系统其实最主要的配置就是输出方式、用例级别和实际的用例记录了，输出方式就是配置好文件夹、文件命名方式、日志大小、备份数，级别空值我的设置是日志级别和输出级别都是DEBUG来更详细展示信息。

保存策略我是通过日来进行日志的命名，然后设置自动的日志回收。在日志记录时，除了一些重要的中间信息的记录，最多、最重要的记录其实是异常的记录，对于异常信息的处理遵循只记录、不拦截的原则，确保异常信息正常记录、也确保失败用例正确被识别到。

### 你的日志是否能精确到每个用例？在 Allure 报告中是否能展示该用例独立日志？

是的，而且似乎该功能是pytest和allure插件自动实现的，我主要就定义了日志类、获取日志实例对象来在我人为重要的地方记录日志内容，而pytest-allure插件能够自动收集上一个用例结束到该用例结束间属于该用例执行的日志内容，并且是自动匹配用例在报告中展示的。

### Allure 中用到了哪些动态内容？你在哪些位置使用了 allure.dynamic.xxx()？

由于我的框架属于是高抽象级别的统一入口，所以所有的features、story这些内容全部都是通过allure.dynamic来实现的，一般来说就是在统一入口就把这些类别信息设置好，主执行逻辑不涉及allure，然后分下到请求、断言这些内容时就会处理一些allure的附件。

### 你有没有遇到 Allure 报告合并冲突或 UUID 重复的问题？是如何解决的？

是的，在处理分批次运行时遇到过，最初的解决方案就是把单接口和工作流的用例来批次拆分强行通过区别函数来避免冲突。现在发现可以通过allure.dynamic.id()配合uuid库来人为指定报告内容的uuid来避免冲突。

### 你是否自定义过 Allure 的附件或截图功能？用在哪些场景？attach 的 MIME type 是什么？

Allure的附件或截图功能是可视化报告非常重要、常用的一个功能，我对其的主要使用就是记录接口的一些请求和返回信息，不同于日志记录，allure附件的记录默认用例的执行是正确的、没有broken的，只关注因和果的内容，因记录传入的请求头、URL、参数等信息，果就是记录断言是否通过、响应内容。

MIME type其实就是用于描述附件类型的一个参数，常用的就是attachment\_type.JSON.TEXT、JSON这种，而对于UI测试可能还会用到PNG这种真正的“截图”附件。

## 配置与扩展能力

### 框架是否支持多环境配置管理？配置项是写在哪，如何被读取并生效？

是的，我的项目天然支持环境的配置管理，简单的环境切换只需要在conf.yaml内配置好env的选择和响应的url切换即可，而如果需要全面的包括日志、远程数据库的切换，可以直接配置多个针对不同环境的配置yaml文件，直接切换yaml文件夹即可。

而对于配置内容的读取则是通过一个confloader类，其会读取设定好的conf配置文件，把配置内容或动态或静态地加载出来供测试框架使用。

### 你是否对框架进行了封装，比如统一入口 run.py 或 runner？支持命令行参数吗？

当然进行了封装，对于整体的一个集成完成的框架，我们通过一个run.py作为框架session执行的入口，而这个run.py执行入口主要就是负责批次的切换空值、执行命令的构造，这里的执行命令就是支持各种各样的命令行参数，通过pytest.mian来执行命令语句，避免命令行执行命令难管理保存的问题。当然对于框架的控制一部分是由这里的命令行参数控制，一部分是有全局的配置文件conf.yaml来配置，还有一部分简单功能由pytest.ini控制。

### 框架是否支持插件扩展，如：pytest-rerunfailures、pytest-xdist？是否实际使用过？

pytest的这个框架原本就是非常支持插件拓展的，例如目前的框架就使用有pytest-allure、rerunfalures等插件。

### 你有没有为这个框架写过 CI/CD 脚本或考虑在 Jenkins/GitLab Runner 中运行？

由于缺少实际的生产环境，框架目前还是不支持CI/CD，但是我初步了解过框架的CI/CD的原理：

1. 测试开发人员将自动化测试框架编写完成后，提交到 Git 代码仓库。
2. 项目配置了 CI/CD 流程（如 Jenkins），通过 Git Hook 或代码变更触发策略来监听仓库变更。
3. 当开发人员提交新的代码时，Jenkins 会自动拉取最新代码，执行测试框架中的用例，生成测试报告。
4. 最终的测试结果会通过邮件、钉钉等方式反馈给相关开发人员或项目经理，用于及时发现问题，保障代码质量。

### 如何设计框架结构让新手也能快速上手使用和写用例？有没有自动生成用例模版的功能？

目前没有自动生成用例的功能，但是我们提供了完整规范的用例模板和说明文档，新手能够快速的上手编写测试用例。

此外我畅想过一个低代码的功能，通过一个平台就可以联系后端的测试框架，然后基于可视化的用例类别设置、标题设置、请求内容设置等就可以自动格式化生成用例，然后这个平台也可以操作后端框架来执行用例，结果的展示也直接集成allure。

### 你是否支持参数化依赖，例如从 Excel / CSV 批量导入测试数据？

目前是只支持yaml作为数据源，不过支持这些其它格式的拓展，其实就是把字段内容对应好即可，我在数据分析方向的经验能让我有充分的经验来解决不同格式数据的处理。

## 健壮性与边界处理

### 如果接口返回非常大的数据或乱码，你的框架能否稳定解析或打印？

这个问题目前还没有考虑到，我目前的处理方法就是直接提供返回对象给断言类，断言类主要通过jsonpath来解析返回体内容来获取数据进行断言，而上下文的保存也是类似的处理方法，并没有考虑到超大数据的问题，不过乱码问题会导致出现异常，而我全面的异常处理和日志记录是能够发现问题的。

而对于超大数据的处理可能需要考虑.超大数据是否是正常的？如果是正常、已知后端就是可能返回超大数据的情况就更具后端的返回规范来进行处理，例如是文件的话就额外进行保存即可；如果是异常，直接通过返回内容的大小进行拦截，遇到过大文件直接进入异常处理。

### 你是如何处理 YAML 数据格式错误或关键字段缺失导致的异常？

我的框架有着非常精细的异常捕获和记录，如果由于格式错误或者关键字确实等错误会最终触发异常而被捕获进行日志的记录。

### 你的框架是否做了异常捕获机制？是全局捕获还是每步 try/except？

我的框架进行了充分的异常处理机制，但是并不是异常捕获，而是异常信息的收集再传导，如果直接捕获其实可能会导致用例失败的原因错误，因为可能从由异常导致的“Broken”变成断言导致的“Fail”。

我的异常处理是根据实际要求实现的，在主逻辑中用例信息收集到用例执行算是采用的大try/except捕获，而断言采用独立的异常捕获；由于主逻辑本质上是负责整合，具体到一些具体的功能内都会有更加精细化的异常处理和传导，这样可以让日志更具体、更详细记录到异常出现的位置，便于查看用例执行失败的原因。

### 对接口响应做断言时，是不是支持多种方式（键值匹配、子集、正则等）？

我们支持状态码、相等、包含、不等、数据库等多种断言方式，其中状态码是直接获取状态码断言、数据库是查询语句与预期结果进行断言，其余则是通过字段来进行键值匹配，因为后端返回的内容最常用的传递或者说解析方式就是json格式，而使用json可以直接解析成python中的字典，也能够通过jsonpath进行快速的内容查找，所以主要是支持键值的匹配，而子集的断言方式主要体现在包含断言中。

### 有没有处理过 HTTP 非200状态码？怎么判断请求失败与业务失败的区别？

处理过的，这个其实通过一个状态码断言即可进行处理。狭义的理解请求失败就是发送请求时由于超时或者服务器错误得到不符合预期的状态码例如404这种，广义上来说，只要是非业务失败都可以叫做请求失败，因为中间一些异常导致用例崩溃自然没有成功执行请求，所以自然也是请求失败；而业务失败就是请求成功但是得到的结果与预期结果不符。

在我的框架中，状态码也是断言的一部分，如果是业务失败，日志和报告中会正确说明是“断言不通过”，而请求失败则是直接终端，日志中可以查看到具体的异常信息。

## 进阶与优化方向

### 如果接口是异步的，比如发出请求需要轮询状态，你是怎么设计等待和断言逻辑的？

目前没有考虑过这个问题，我的框架目前只支持同步接口；针对这种情况，我了解到的一种解决策略是：在请求执行模块添加对轮询的处理，待到异步请求执行完成再继续返回，而这也要求实现了解好哪些接口是需要异步执行的。

### 你有使用过 JSON Schema 来断言接口返回格式吗？是否封装成通用断言函数？

Json scheme就是一份用来检验接口返回格式的模板说明书，我的框架目前不支持这种断言方式，主要还是业务层面的预期内容的检验，不过这种方式的断言拓展也非常容易，只需要在断言类添加一个更具接口返回内容识别请求类别而继续scheme匹配的函数、以及规范一个scheme检查的字段即可。

### 你考虑过对框架增加 Swagger 自动化生成用例功能吗？

我的框架目前不支持，其实主要因为我的框架源于一个简单的练手项目，没有swagger接口文档。不过我有了解过这种自动生成用例的做法，因为swagger其实就是一个格式化的json接口文档，整体的做法其实和数据分析很类似，读出对应的字段，分析结构，根据设定的规范来构造yaml文件即可得到用例。这个功能其实更适合在一个测试管理平台构造。

### 你是否对报告做过二次封装，如将报告上传、推送到通知渠道（钉钉、企业微信）？

目前是没有实现二次封装，不过学习时了解到这部分功能，所以是预留有接入的空间的，同时也提供了全局配置文件对是否启用钉钉发送、邮箱发送等管理。

### 你是否思考过如何将这个框架迁移为更通用的服务化测试平台？支持 Web 界面管理和触发？

考虑过的，其实就是把用例设置、配置管理通过可视化web界面进行展示，用低代码的方式来管理测试框架，不过这里面可能不太方便进行复杂的内容管理，比较好实现的功能其实就是用例设计、配置管理、框架执行和结果展示，然后内部的复杂钩子其实是不好处理的。

# 总结

## 不足与经验

### 单接口与多接口用例未统一

目前虽然支持单接口用例和多接口工作流用例，但是两者格式上并不统一，目前依赖两个入口函数来进行类别的区分，未来的一个优化方向就是把所有的用例格式进行统一，因为本质上单接口也就是多接口工作流的一个特殊情况，这样就可以真正实现一种规范、一个入口来执行实现所有的用例。

### 分批次运行与报告合并的优化

目前支持分批次运行，但是conf.yaml和conloader定义的批次加载写法还是有点问题，还有一个就是目前还没有用到allure.dynamic.id()来动态设置allure用例信息的uuid，所以目前报告的合并还是只能通过分批次分入口来避免冲突的。

### 数据源支持的丰富性

目前只支持的是yaml作为数据源，不过拓展csv、excel等也不过是通过现成的解析库来把数据内容读取出来做转换即可。

### CI/CD的拓展

目前还没有接入CICD，不过实际生产中会有专门的运维人员进行，目前了解整体的一个CICD逻辑与过程即可。

### 测试环境的切换

框架天然支持环境切换但缺少实际的生产环境进行切换管理。后面可能需要结合实际的生产环境进行测试环境的灵活配置。

### ${}的完整处理

目前${}主要支持的是extract的提取替换，但是实际预留了很多的时间加载、内容加密等很多的功能还没有完成。这些功能存在主要是补全一些接口的前端逻辑，因为有时候前端是需要进行一些加密或者时间等信息的自动补全加载的，而进行接口测试时没有通过前端进行请求，也就需要主动进行处理。

## 优化畅想

一套完整的优化是这样的：构建一个web前端，其连接测试框架进行管理，能够实现可视化的低代码测试用例编写，例如经理添加用例模板或者自动识别swagger构造用例模板以及获取字段，只需要点击选择字段输入即可自动构造yaml测试数据；此外还能够对conf.yaml进行管理，实现测试框架的整体配置，还有就是能够通过这个web前端进行框架执行和展示结果，这样就有了一个实际生产中内部使用的测试平台，同时还可以提供对pingcode、禅道、jira等的集成。

## 总结

本电子商务接口自动化测试框架的开发，源于对接口测试知识的深入探索与实践。从最初的练手项目起步，通过逐步构建数据驱动体系、完善用例管理机制、集成日志与报告系统，最终形成了一套功能相对完整、可复用性较强的测试框架。

在技术实现上，框架以Python为基础，结合Pytest的钩子机制实现了灵活的用例参数化与动态加载，通过YAML实现测试数据与执行逻辑的解耦，借助Requests完成接口请求封装，搭配Allure生成直观的测试报告，同时通过日志系统与异常处理机制保障了测试过程的可追溯性。这些设计不仅满足了基础的接口测试需求，更通过前置条件独立执行、上下文变量管理、分批次运行等功能，提升了用例的独立性与框架的扩展性。

开发过程中，我们深刻体会到“高内聚、低耦合”设计原则的重要性。将用例加载、请求执行、断言处理、日志记录等功能拆分为独立模块，既便于单独维护，也为后续拓展（如支持多数据源、集成CI/CD、开发Web管理界面）预留了空间。同时，通过解决用例依赖、报告合并、Mock服务等实际问题，积累了接口自动化测试框架设计的实战经验——例如，通过extract.yaml与动态参数替换解决上下文传递问题，通过Allure动态标签实现报告的清晰分类，这些方案均为同类场景提供了可参考的思路。

当然，框架仍存在诸多可优化之处：单接口与多接口用例格式尚未统一、数据源支持需进一步丰富、${}函数的功能拓展未完全实现等。但这些不足也为后续迭代指明了方向。未来，我们期望通过构建可视化Web平台，实现低代码用例编写、自动化配置管理与测试结果展示，将框架升级为更通用的服务化测试平台，同时集成更多第三方工具（如Swagger、Jira等），使其在实际生产环境中发挥更大价值。

总而言之，本项目不仅是对接口自动化测试技术的一次实践，更是对“测试框架设计思维”的深入探索。通过持续优化与完善，这套框架有望从练手项目逐步成长为能够支撑企业级接口测试需求的实用工具，为保障系统稳定性与提升测试效率贡献力量。