

接口自动化框架编写（章节）

学习目标

1. 场景分析（章导学）
 - 1.1 使用场景
- 2 项目及框架搭建（小节）
 - 2.1 项目环境工具（知识点）
 - 2.2 接口框架搭建（知识点）
3. 接口用例编写（小节）
 - 3.1 接口用例编写（知识点）
4. Requests
 - 4.1 Requests介绍及简单使用
 - 4.2 Requests用例编写
 - 4.3 Requests方法封装
- 5 配置文件
 - 5.1 Yaml 介绍及安装
 - 5.2 Yaml 字典和列表介绍
 - 5.3 Yaml 字典和列表基本操作
 - 5.3.1 字典嵌套字典
 - 5.3.2 字典嵌套列表
 - 5.3.3 列表嵌套字典
 - 5.3.4 列表嵌套列表
 - 5.4 读取 Yaml 文件
 - 5.4.1 读取单个Yaml文档
 - 5.4.2 读取多个Yaml文档
 - 5.5 yaml封装
 - 5.6 配置文件设置
- 6 日志文件
 - 6.1 logging模块简介及快速使用
 - 6.2 logging模块基本使用
 - 6.3 将日志进行封装
- 7 pytest框架
 - 7.1 Pytest 安装与入门
 - 7.2 Pytest 基础使用
 - 7.3 Pytest 常用插件
 - 7.4 Pytest 数据参数化
 - 7.5 Pytest 应用接口用例
- 8 结果断言
 - 8.1 断言介绍
 - 8.2 结果断言验证
 - 8.3 数据库结果验证
- 9 数据驱动
 - 9.1 yaml数据驱动
 - 9.2 Excel数据驱动
- 10 Allure报告
 - 10.1 Allure的安装及快速入门
 - 10.2 Allure 详解
 - 10.3 Allure应用接口
 - 10.4 Allure报告自动生成
- 12 章总结
 - 重难点
 - 拓展延伸

接口自动化框架编写（章节）

学习目标

- 掌握如何进行自动化测试框架的搭建【重点】
- 掌握接口requests使用【重点】
- 掌握pytest框架【重点】
- 掌握数据驱动【重点】
- 掌握测试报告allure使用【重点】
- 掌握邮件配置功能【重点】

1, 场景分析 (章导学)

1.1 使用场景

使用Excel编写用例，通过pytest框架进行测试，生成测试报告


- 测试用例
 image-20190809151357056
- 效果展示

 image-20190809151146657

2 项目及框架搭建 (小节)

2.1 项目环境工具 (知识点)

- python
 - <https://www.python.org/downloads/>
- pycharm
 - <http://www.jetbrains.com/pycharm/>
- git
 - <https://git-scm.com/downloads>

2.2 接口框架搭建 (知识点)

- 创建项目
 - 名称: InterAutoTest
- 创建框架目录
 image-20190716163614449
- Pycharm配置
 - 配置python解释器
 - Settings -> Project -> Project Interpreter
 - 配置git解释器
 - Settings -> Version Control -> Git
- Pycharm+Git配置
 - 配置github

- 创建github项目
 - <https://github.com/>
- 上传github
 - 增加忽略文件
 - 安装.ignore插件
 - 上传
 - 配置github

3. 接口用例编写（小节）

3.1 接口用例编写（知识点）

- 被测接口
 - 登录
 - 用户中心个人信息
 - 获取商品列表数据
 - 添加到购物车
 - 保存订单
- 编写接口测试用例

4. Requests

4.1 Requests介绍及简单使用

- Requests介绍
 - 流行的接口http(s)请求工具
 - 使用功能强大、简单方便、容易上手
- Requests简单使用
 - 安装Requests包
 - `$ pip3 install requests`
 - 简单使用

```
import requests
requests.get("http://www.baidu.com")
```

- Requests请求返回介绍

<code>r.status_code</code>	#响应状态
<code>r.content</code>	#字节方式的响应体，会自动为你解码 gzip 和 deflate 压缩
<code>r.headers</code>	#以字典对象存储服务器响应头，若键不存在则返回None
<code>r.json()</code>	#Requests中内置的JSON
<code>r.url</code>	# 获取url
<code>r.encoding</code>	# 编码格式
<code>r.cookies</code>	# 获取cookie
<code>r.raw</code>	#返回原始响应体
<code>r.text</code>	#字符串方式的响应体，会自动根据响应头部的字符编码进行
<code>r.raise_for_status()</code>	#失败请求(非200响应)抛出异常

4.2 Requests用例编写

- 编写登录用例脚本

```
url = "http://172.17.0.139:5004/authorizations/"
data = {"username": "python",
        "password": "12345678"}
r = requests.post(url, json=data)
```

- 编写个人信息获取脚本

```
url = "http://172.17.0.139:5004/user/"
headers = {'Authorization': 'JWT
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NjUwNjMwOTksInVzZXJfawQiojEsImVtYWlsIj
oiOTUyNjc2NjM4QHFxLmNvbSIsInVzZXJ1eW11Ijoichl0aG9uIn0.f3GdeSnzb3UGs-
wlp1ejZ1rNLaaibOAHunN8_pq8LDE"}'
r = requests.get(url, headers=headers)
```

- 商品列表数据

```
url = "http://172.17.0.139:5004/categories/115/skus"
data = {
    # "category_id": "115",
    "page": "1",
    "page_size": "10",
    "ordering": "create_time", # 'create_time', 'price', 'sales'
}
r = requests.get(url, json=data)
print(r.text)
print(r.json)
```

- 添加到购物车

```
def cart():
    url = "http://172.17.0.139:5004/cart/"
    data = {
        "sku_id": "3",
        "count": "1",
        "selected": "true",
        # "selected": True
    }
    headers = {"Authorization": "JWT
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NjU4NTU2NzcsInVzZXJfawQiojEsImVtYWlsIj
oiOTUyNjc2NjM4QHFxLmNvbSIsInVzZXJ1eW11Ijoichl0aG9uIn0.yothV8wMX7MKHyioFDaGnrjTDTAYSLB0R
8Qsungw_ms"}

    r = requests.post(url, json=data, headers=headers)
    print(r.text)
    print(r.json)

def get_cart():
    url = "http://172.17.0.139:5004/cart/"
    headers = {
        "Authorization": "JWT
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NjU4NTU2NzcsInVzZXJfawQiojEsImVtYWlsIj
oiOTUyNjc2NjM4QHFxLmNvbSIsInVzZXJ1eW11Ijoichl0aG9uIn0.yothV8wMX7MKHyioFDaGnrjTDTAYSLB0R
8Qsungw_ms"}

    r = requests.get(url, headers=headers)
    print(r.text)
    print(r.json)
```

- 保存订单

```
def orders():
    url = "http://172.17.0.139:5004/orders/"
    data = {
        "address": "1",
        "pay_method": "1",
    }
    headers = {"Authorization": "JWT
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NjU4NTU2NzcsInVzZXJfawQiojEsImVtYWlsIj
oiOTUyNjc2NjM4QHFxLmNvbSIsInVzZXJ1YW1IjoicHl0aG9uIn0.yoTHV8wMX7MKHyioFDaGnrjTDTAYSLB0R
8Qsung_ms"}

    r = requests.post(url, json=data, headers=headers)
    print(r.text)
    print(r.json)
```

4.3 Requests方法封装

封装requests get方法

```
#1、创建封装get方法
def requests_get(url, headers):
#2、发送requests get请求
    r = requests.get(url, headers = headers)
#3、获取结果相应内容
    code = r.status_code
    try:
        body = r.json()
    except Exception as e:
        body = r.text
#4、内容存到字典
    res = dict()
    res["code"] = code
    res["body"] = body
#5、字典返回
    return res
```

封装requests post方法

```
#post方法封装
#1、创建post方法
def requests_post(url, json=None, headers=None):
#2、发送post请求
    r = requests.post(url, json=json, headers=headers)
#3、获取结果内容
    code = r.status_code
    try:
        body = r.json()
    except Exception as e:
        body = r.text
#4、内容存到字典
    res = dict()
    res["code"] = code
    res["body"] = body
#5、字典返回
    return res
```

封装requests公共方法

- 增加cookies,headers参数
- 根据参数method判断get/post请求

```
def requests_api(self, url, data = None, json=None, headers=None, cookies=None, method="get"):  
  
    if method == "get":  
        #get请求  
        self.log.debug("发送get请求")  
        r = requests.get(url, data = data, json=json, headers=headers, cookies=cookies)  
    elif method == "post":  
        #post请求  
        self.log.debug("发送post请求")  
        r = requests.post(url, data = data, json=json, headers=headers, cookies=cookies)  
  
    #2. 重复的内容, 复制进来  
    #获取结果内容  
    code = r.status_code  
    try:  
        body = r.json()  
    except Exception as e:  
        body = r.text  
    #内容存到字典  
    res = dict()  
    res["code"] = code  
    res["body"] = body  
    #字典返回  
    return res
```

重构get方法

- 调用公共方法request_api,
- 参数: 固定参数: url,method
其它参数: **args

#1、定义方法

```
def get(self, url, **kwargs):  
    #2、定义参数  
    #url,json,headers,cookies,method  
    #3、调用公共方法  
    return self.requests_api(url, method="get", **kwargs)
```

重构post方法

- 调用公共方法request_api,
- 参数: 固定参数: url,method
其它参数: **args

```
def post(self, url, **kwargs):  
    #2、定义参数  
    #url,json,headers,cookies,method  
    #3、调用公共方法  
    return self.requests_api(url, method="post", **kwargs)
```

###

5 配置文件

5.1 Yaml 介绍及安装

- Yaml 介绍

Yaml 是一种所有编程语言可用的友好的数据序列化标准。语法和其他高阶语言类似，并且可以简单表达字典、列表和其他基本数据类型的形态。语法规则如下：

1. 大小写敏感。
 2. 使用缩进表示层级关系。
 3. 使用空格键缩进，而非Tab键缩进
 4. 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可。
 5. 文件中的字符串不需要使用引号标注，但若字符串包含有特殊字符则需用引号标注；
 6. 注释标识为#
- 官网：<https://yaml.org/>

- Yaml 安装

```
$ pip3 install PyYaml
```

- Yaml 快速体验

- 字典{"name": "test_yaml", "result", "success"}写成Yaml的形式，并输出结果
结构通过空格缩进来展示。列表里的项用"-"来代表，字典里的键值对用":"分隔。

- 示例函数：

data.yaml

```
name: "test_yaml"  
result: "success"
```

demo.py

```
import yaml  
  
with open("./data.yaml", 'r') as f:  
    data = yaml.safe_load(f)  
    print(data)
```

- 运行结果

```
{'name': 'test_yaml', 'result': 'success'}
```

5.2 Yaml 字典和列表介绍

5.2.1 字典

- 字典里的键值对用":"分隔。

示例： {"name": "test_yaml", "result", "success"}

```
name: "test_yaml"  
age: "success"
```

总结：

- 字典直接写key和value，每一个键值对占一行。
- : 后面要有空格

5.2.2 列表

- 一组按序排列的值（简称 "序列或列表"）

- 数组前加有“-”符号，符号与值之间需用空格分隔

示例: ["a", "b", "c"]

```
- "a"  
- "b"  
- "c"
```

- 列表中的元素需要用 -来表示
- 每一个元素前面都有一个 -, 后面接元素内容

5.3 Yaml 字典和列表基本操作

5.3.1 字典嵌套字典

- 需求

```
{  
    person1: {  
        "name": "xiaoming",  
        "age": "18"  
    },  
    person2: {  
        "name": "xiaohong",  
        "age": "20"  
    }  
}
```

- 示例

```
person1:  
  name: "xiaoming"  
  age: "18"  
person2:  
  name: "xiaohong"  
  age: "20"
```

- 结果

```
{'person1': {'name': 'xiaoming', 'age': '18'}, 'person2': {'name': 'xiaohong', 'age': '20'}}
```

5.3.2 字典嵌套列表

- 需求

```
{person: ["1", "2", "3"]}
```

- 示例

```
persion:  
- "1"  
- "2"  
- "3"
```

- 结果

```
{'person': ['1', '2', '3']}
```

5.3.3 列表嵌套字典


```
[{
    "username1": "test1"
}, {
    "password1": "111",
    "username2": "test2",
    "password2": "222"
}]
```

- 示例

```
- username1: "test1"
- password1: "111"
  username2: "test2"
  password2: "222"
```

- 结果

```
[{'username1': 'test1'}, {'password1': 111, 'username2': 'test2', 'password2': 222}]
```

5.3.4 列表嵌套列表

- 需求

```
[["a","b","c"],["1","2","3"]]
```

- 示例

```
-
- "a"
- "b"
- "c"
-
- "1"
- "2"
- "3"
```

- 结果

```
[['a', 'b', 'c'], [1, 2, 3]]
```

注意:

- pycharm会自动将yaml中的tab转化成空格。但是不能复制“tab键”，不会进行转化。
- 同级别左侧必须要对齐，少个空格无所谓。

5.4 读取 Yaml 文件

5.4.1 读取单个Yaml文档

- yaml文档

```
#data.yaml
"用户名称": "test123"
"密码": "123456"
```

- 示例函数: demo.py 中文问题(windows): encoding='utf-8'

```
# 打开要读取的文件,中文 encoding='utf-8'
with open('./data.yaml', 'r', encoding='utf-8') as f:
    # 加载文件内容
    data = yaml.safe_load(f)
    # 打印结果
    print(data)
```

读取实际上就是先使用 `with open` 的形式获取文件对象, 再使用 `yaml` 中的 `load` 方法加载这个文件对象, 通过 `load` 方法, 一个字符串变成了一个字典

- 结果

```
{'用户名称': 'test123', '密码': '123456'}
```

5.4.2 读取多个Yaml文档

多个文档在一个yaml文件, 使用 `---` 分隔方式来分段

- yaml文档

```
---
"用户名称1": "test123"
"密码": "123456"
---
"用户名称2": "test456"
"密码": "123456"
```

- 示例

- import yaml

```
with open('./data.yaml','r') as f:
    data = yaml.safe_load_all(f)
    for i in data:
        print(i)
```

- 结果

- {'用户名称1': 'test123', '密码': '123456'}
- {'用户名称2': 'test456', '密码': '123456'}

5.5 yaml封装

```
import os
import yaml
#1、创建类
class YamlReader:
#2、初始化, 文件是否存在
    def __init__(self,yamlf):
        if os.path.exists(yamlf):
            self.yamlf = yamlf
        else:
            raise FileNotFoundError("文件不存在")
        self._data = None
        self._data_all = None
#3、yaml读取
    #单个文档读取
    def data(self):
        #第一次调用data, 读取yaml文档, 如果不是, 直接返回之前保存的数据
        if not self._data:
            with open(self.yamlf,"rb") as f:
```

```
self._data = yaml.safe_load(f)
return self._data
#多个文档读取
def data_all(self):
    #第一次调用data，读取yaml文档，如果不是，直接返回之前保存的数据
    if not self._data_all:
        with open(self.yamlf,"rb") as f:
            self._data_all = list(yaml.safe_load_all(f))
    return self._data_all
```

5.6 配置文件设置

- 配置文件conf.yaml

```
BASE:
    #log等级
    log_level: "debug"
    #扩展名
    log_extension: ".log"
    test:
        url: "http://211.103.136.242:8064"
        case_file: "testdata.xlsx"
        case_sheet: "美多商城接口测试"
```

- 基本目录配置

```
import os
from utils.YamlUtil import YamlReader
#1、获取项目基本目录
#获取当前项目的绝对路径
current = os.path.abspath(__file__)
#print(current)
BASE_DIR = os.path.dirname(os.path.dirname(current))
#print(BASE_DIR)
#定义config目录的路径
_config_path = BASE_DIR + os.sep + "config"

#定义data目录的路径
_data_path = BASE_DIR + os.sep + "data"

#定义conf.yaml文件的路径
_config_file = _config_path + os.sep + "conf.yaml"
#定义db_conf.yaml路径
_db_config_file = _config_path + os.sep + "db_conf.yaml"
#定义logs文件路径
_log_path = BASE_DIR + os.sep + "logs"

#定义report目录的路径
_report_path = BASE_DIR + os.sep + "report"

def get_report_path():
    """
    获取report绝对路径
    :return:
    """
    return _report_path

def get_data_path():
    return _data_path
```

```
def get_db_config_file():  
    return _db_config_file
```

```
def get_config_path():  
    return _config_path
```

```
def get_config_file():  
    return _config_file
```

```
def get_log_path():  
    """  
    获取Log文件路径  
    :return:  
    """  
    return _log_path
```

- conf.yml配置文件读取

#2、读取配置文件

#创建类

```
class ConfigYaml:
```

#初始yaml读取配置文件

```
    def __init__(self):  
        self.config = YamlReader(get_config_file()).data()  
        self.db_config = YamlReader(get_db_config_file()).data()
```

#定义方法获取需要信息

```
    def get_excel_file(self):  
        """  
        获取测试用例excel名称  
        :return:  
        """  
        return self.config["BASE"]["test"]["case_file"]
```

```
    def get_excel_sheet(self):  
        """  
        获取测试用例sheet名称  
        :return:  
        """  
        return self.config["BASE"]["test"]["case_sheet"]
```

```
    def get_conf_url(self):  
        return self.config["BASE"]["test"]["url"]
```

```
    def get_conf_log(self):  
        """  
        获取日志级别  
        :return:  
        """  
        return self.config["BASE"]["log_level"]
```

```
    def get_conf_log_extension(self):  
        """  
        获取文件扩展名  
        :return:  
        """  
        return self.config["BASE"]["log_extension"]
```

```
    def get_db_conf_info(self, db_alias):  
        """
```

根据db_alias获取该名称下的数据库信息

```
:param db_alias:
:return:
"""
return self.db_config[db_alias]

def get_email_info(self):
"""
获取邮件配置相关信息
:return:
"""
return self.config["email"]
```

6 日志文件

6.1 logging模块简介及快速使用

logging模块是Python内置的标准模块，主要用于输出运行日志，可以设置输出日志的等级、日志保存路径、日志文件回滚等；

- 优点：
 1. 可以通过设置不同的日志等级，在release版本中只输出重要信息，而不必显示大量的调试信息；
 2. 尤其是代码运行在服务器上，当出现问题时可以通过查看日志进行分析

logging快速使用

- 打印日志到标准输出中

- 默认的日志级别

```
#1、导入Logging包
import logging
#2、设置配置信息
logging.basicConfig(level=logging.INFO,format='%(asctime)s-%(name)s-%(levelname)s-%(message)s')
#3、定义日志名称getlogger
logger = logging.getLogger("log_demo")
#4、info,debug
logger.info("info")
logger.debug("debug")
logger.warning("warning")
```

6.2 logging模块基本使用

日志输出文件

- 设置logger名称

```
logger = logging.getLogger(log_name)
```

- 设置log级别

```
logger.setLevel(logging.info)
```

- 创建一个handler，用于写入日志文件

```
fh = logging.FileHandler(log_file)
```

- 设置日志级别，默认为logging.WARNING

```
fh.setLevel(logging.WARNING)
```

- 定义handler的输出格式

```
formatter = logging.Formatter('%(asctime)s %(name)s [line:%(lineno)d] %(levelname)s %(message)s')
fh.setFormatter(formatter)
```

- 添加handler

```
logger.addHandler(fh)
```

备注：

- format

%(lineno)s: 打印日志级别的数值
 %(levelname)s: 打印日志级别名称
 %(pathname)s: 打印当前执行程序的路径，其实就是sys.argv[0]
 %(filename)s: 打印当前执行程序名
 %(funcName)s: 打印日志的当前函数
 %(lineno)d: 打印日志的当前行号
 %(asctime)s: 打印日志的时间
 %(thread)d: 打印线程ID
 %(threadName)s: 打印线程名称
 %(process)d: 打印进程ID
 %(message)s: 打印日志信息

- 代码

```
import logging

#输出控制台
#1、设置logger名称
logger = logging.getLogger("log_file_demo")
#2、设置log级别
logger.setLevel(logging.INFO)
#3、创建handler
fh_stream = logging.StreamHandler()
#写入文件
fh_file = logging.FileHandler("./test.log")
#4、设置日志级别
fh_stream.setLevel(logging.DEBUG)
fh_file.setLevel(logging.WARNING)
#5、定义输出格式
formatter = logging.Formatter('%(asctime)s %(name)s %(levelname)s %(message)s ')
fh_stream.setFormatter(formatter)
fh_file.setFormatter(formatter)
#6、添加handler
logger.addHandler(fh_stream)
logger.addHandler(fh_file)
#7、运行输出

logger.info("this is a info")
logger.debug("this is a debug")
logger.warning("this is a warning")
```

6.3 将日志进行封装

```

import logging
from config import Conf
import datetime, os
from config.Conf import ConfigYaml
#定义日志级别的映射
log_l = {
    "info": logging.INFO,
    "debug": logging.DEBUG,
    "warning": logging.WARNING,
    "error": logging.ERROR
}

#1、创建类
class Logger:
#2、定义参数
    #输出文件名称, Loggername, 日志级别
    def __init__(self, log_file, log_name, log_level):
        self.log_file = log_file #扩展名 配置文件
        self.log_name = log_name #参数
        self.log_level = log_level # 配置文件
#3、编写输出控制台或文件
    # 设置logger名称
    self.logger = logging.getLogger(self.log_name)
    # 设置log级别
    self.logger.setLevel(log_l[self.log_level]) #logging.INFO
    #判断handlers是否存在
    if not self.logger.handlers:
        # 输出控制台
        fh_stream = logging.StreamHandler()
        fh_stream.setLevel(log_l[self.log_level])
        formatter = logging.Formatter('%(asctime)s %(name)s %(levelname)s %(message)s ')
        fh_stream.setFormatter(formatter)
        # 写入文件
        fh_file = logging.FileHandler(self.log_file)
        fh_file.setLevel(log_l[self.log_level])
        fh_file.setFormatter(formatter)

        # 添加handler
        self.logger.addHandler(fh_stream)
        self.logger.addHandler(fh_file)

#1、初始化参数数据
#日志文件名称, 日志文件级别
#日志文件名称 = logs目录 + 当前时间+扩展名
#log目录
log_path = Conf.get_log_path()
#当前时间
current_time = datetime.datetime.now().strftime("%Y-%m-%d")
#扩展名
log_extension = ConfigYaml().get_conf_log_extension()
logfile = os.path.join(log_path, current_time+log_extension)
#print(logfile)
#日志文件级别
loglevel = ConfigYaml().get_conf_log()
#print(loglevel)
#2、对外方法, 初始log工具类, 提供其它类使用
def my_log(log_name = __file__):
    return Logger(log_file=logfile, log_name=log_name, log_level=loglevel).logger
    
```

```
if __name__ == "__main__":  
    my_log().debug("this is a debug")
```

7 pytest框架

7.1 Pytest 安装与入门

- 介绍
 - 简单灵活,
 - 容易上手,
 - 文档丰富
 - 支持参数化, 能够支持简单的单元测试和复杂的功能测试
 - 具有很多第三方插件, 并且可以自定义扩展
 - 可以很好的和Jenkins工具结合

- 安装pytest

1. 命令行执行以下命令

```
$ pip3 install -U pytest
```

2. 检查版本

```
$ pytest --version  
This is pytest version 4.5.0, imported from  
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/pytest.py
```

- 创建你的第一个测试用例

1. 创建一个简单的测试方法

```
#test_sample.py  
def func(x):  
    return x + 1  
  
def test_a():  
    print("---test_a---")  
    assert func(3) == 5 #断言失败  
  
def test_b():  
    print("---test_b---")  
    assert 1 #断言成功
```

2. 执行一下

- 命令行模式

- 命令下执行

```
$ pytest -s test_sample.py
```

- 主函数模式

- 增加主函数


```
if __name__ == '__main__':
    pytest.main(["-s", "test_sample.py"])
#-s 表示支持控制台打印，如果不加，print 不会出现任何内容
#-q 安静模式，不打印信息
```

◦ 执行结果

```
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-4.5.0, py-1.8.0, pluggy-0.11.0
rootdir: /Users/wuyan hong/PycharmProjects/Inter_AutoTest_w, inifile: pytest.ini
plugins: rerunfailures-7.0, ordering-0.6, metadata-1.8.0, html-1.20.0, allure-
pytest-2.6.4
collected 2 items

test_sample.py ---test_a----
F---test_b---
.

===== FAILURES =====
_____ test_a _____

    def test_a():
        print("---test_a----")
>       assert func(3) == 5 # 断言失败
E       assert 4 == 5
E       + where 4 = func(3)

test_sample.py:9: AssertionError
===== 1 failed, 1 passed in 0.09 seconds =====

Process finished with exit code 0
```

由于func(3)并不等于5，这次测试返回了一个失败报告。

- . 表示成功
- 如果需要更多信息，可以使用-v或--verbose
- F 表示失败

7.2 Pytest 基础使用

7.2.1 函数级别方法

运行于测试方法的始末，运行一次测试函数会运行一次 setup 和 teardown。

• 示例代码

```
import pytest
class TestLogin:
    # 函数级开始
    def setup(self):
        print("----->setup_method")
    # 函数级结束
    def teardown(self):
        print("----->teardown_method")
    def test_a(self):
        print("----->test_a")
    def test_b(self):
        print("----->test_b")
```

• 执行结果

```
scripts/test_login.py ----->setup_method # 第一次 setup()
----->test_a
.----->teardown_method # 第一次 teardown()
----->setup_method # 第二次 setup()
----->test_b
.----->teardown_method # 第二次 teardown()
```

7.2.2 类级别方法

运行于测试类的始末，在一个测试内只运行一次 `setup_class` 和 `teardown_class`，不关心测试类内有多少个测试函数。

- 示例代码

```
class TestLogin:
    # 测试类级开始
    def setup_class(self):
        print("----->setup_class")
    # 测试类级结束
    def teardown_class(self):
        print("----->teardown_class")
    def test_a(self):
        print("----->test_a")
    def test_b(self):
        print("----->test_b")
```

- 执行结果

```
scripts/test_login.py ----->setup_class # 第一次 setup_class()
----->test_a
.----->test_b
.----->teardown_class # 第一次 teardown_class()
```

7.3 Pytest 常用插件

插件列表网址：<https://plugincompat.herokuapp.com> 包含很多插件包，大家可依据工作的需求选择使用。

7.3.1 测试报告

- 应用场景

自动化测试脚本最终执行是通过还是不通过，需要通过测试报告进行体现。

- 安装

```
$ pip3 install pytest-html
```

- 使用

在配置文件中的命令行参数中增加 `--html=用户路径/report.html`

- 示例

```
pytest.ini
```

```
addopts = -s --html=report/report.html
```

- 结果

7.3.2 失败重试

- 应用场景

自动化测试脚本可能会使用到网络，如果网络不好可能最终会使脚本不通过。像这种情况可能并不是脚本本身的问题，仅仅是因为网络忽快忽慢，那么我们可以使用失败重试的插件，当失败后尝试再次运行。一般情况最终成功可以视为成功，但最好进行进行排查时候是脚本问题。

- 安装

```
pip3 install pytest-rerunfailures
```

- 使用

在配置文件中的命令行参数中增加 --reruns n

- 示例

pytest.ini

```
addopts = -s --reruns 3
```

test_login.py

```
class TestLogin:
```

```
    def test_a(self): # test开头的测试函数
        print("----->test_a")
        assert 1 # 断言成功
```

```
    def test_b(self):
        print("----->test_b")
        assert 0 # 断言失败
```

- 结果

```
scripts/test_login.py ----->test_a
.----->test_b
R----->test_b
R----->test_b
R----->test_b
R----->test_b
F
```

R 表示重试

- 注意点

重试时，如果脚本通过，那么后续不再重试

```
.test_hello_003 # 再运行3
.test_hello_001
```

如果你期望加上出错重试的等待时间，请使用如下命令(reruns-delay是等待时间):

```
pytest --reruns 5 --reruns-delay 1
```

如果你只想对某几个测试用例应用重试策略，你可以使用装饰器:

```
@pytest.mark.flaky(reruns=5, reruns_delay=2)
```

例如:

```
@pytest.mark.flaky(reruns=5, reruns_delay=2)
```

```
def test_example():  
    import random  
    assert random.choice([True, False])
```

7.4 Pytest 数据参数化

- 应用场景

登录功能都是输入用户名，输入密码，点击登录。但登录的用户名和密码如果想测试多个值是没有办法用普通的操作实现的。数据参数化可以帮我实现这样的效果。

- 方法名

- pytest.mark.parametrize

```
# 数据参数化  
# 参数:  
# argnames: 参数名  
# argvalues: 参数对应值, 类型必须为可迭代类型, 一般使用list  
@pytest.mark.parametrize(argnames, argvalues, indirect=False, ids=None,  
scope=None)
```

7.4.1 一个参数使用方式

1. argnames 为字符串类型，根据需求决定何时的参数名
2. argvalues 为列表类型，根据需求决定列表元素中的内容
3. 在测试脚本中，参数，名字与 argnames 保持一致
4. 在测试脚本中正常使用

- 示例

```
import pytest  
class TestLogin:  
    @pytest.mark.parametrize("name", ["xiaoming", "xiaohong"])  
    def test_a(self, name):  
        print(name)  
        assert 1
```

- 结果

```
scripts/test_login.py xiaoming  
.xiaohong  
.
```

7.4.2 多个参数使用方式

- 示例

```
import pytest  
class TestLogin:  
    @pytest.mark.parametrize(("username", "password"), [("zhangsan",  
"zhangsan123"),  
("xiaoming", "xiaoming123")])  
    def test_a(self, username, password):  
        print(username)  
        print(password)  
        assert 1
```

- 结果

```
zhangsan123
.xiaoming
xiaoming123
.
```

多个参数还可以将装饰器写成 `@pytest.mark.parametrize("username,password", [("zhangsan", "zhangsan123"), ("xiaoming", "xiaoming123")])` 效果是一样的。

7.5 Pytest 应用接口用例

- 运行原则

- 在不指定运行目录，运行文件，运行函数等参数的默认情况下，pytest会执行当前目录下的所有以test为前缀(test.py)或以_test为后缀(test.py)的文件中以test为前缀的函数
- pytest会找当前以及递归查找子文件夹下面所有的test.py或_test.py的文件，把其当作测试文件
- 在这些文件里，pytest会收集下面的一些函数或方法，
- 当作测试用例 不在类定义中的以test开头的函数或方法 在以Test开头的类中(不能包含init方法)，以test开头的方法

- 如何执行多条测试?

- pytest会执行当前目录及子目录下所有test_*.py及*_test.py格式的文件
- 可以设置pytest.ini配置文件,自定义执行文件格式

```
addopts = -s
# 当前目录下的scripts文件夹 -可自定义
testpaths = testcase
# 当前目录下的scripts文件夹下，以test_开头，以.py结尾的所有文件 -可自定义
python_files = test_*.py
# 当前目录下的scripts文件夹下，以test_开头，以.py结尾的所有文件中，以Test_开头的类 -可自定义
python_classes = Test_*
# 当前目录下的scripts文件夹下，以test_开头，以.py结尾的所有文件中，以Test_开头的类内，以test_
# 开头的方法 -可自定义
python_functions = test_*
```

- 登录用例脚本

```
url = "http://211.103.136.242:8064/authorizations/"
data = [{"username": "python",
        "password": "12345678"},
        {"username": "python1",
        "password": "test123"},
        ]

@pytest.mark.parametrize("login", data)
def test_login(login):
    res= requests.post(url,json=login)
    print(res.json())

if __name__ == "__main__":
    pytest.main(["-s"])
```

- 结果

```
{"user_id":1,"username":"python","token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOiJlNjUzNDU5NTcsInVzZXJfawQiojEsImVtYWlsIjoioTUyNjcZnM4QHFxLmNvbSIsInVzZXJlIjoicHl0aG9uIn0.K0Kx8lKkex0CjNTuWZ0cA67FU33nkNYBf1_s-5LxNP8"}
{"non_field_errors":["无法使用提供的认证信息登录。"]}
```

8 结果断言

8.1 断言介绍

介绍

- 断言是自动化最终的目的，一个用例没有断言，就失去了自动化测试的意义了
- 断言用到的是 `assert` 关键字。预期的结果和实际结果做对比，符合预期就是 `pass`，不符合就 `fail`

常用断言

pytest里面的断言就是python里`assert`的断言方法

- `assert xx` 判断xx为真
- `assert not xx` 判断xx不为真
- `assert a in b` 判断b包含a
- `assert a == b` 判断a等于b
- `assert a != b` 判断a不等于b

案例

```
def is_true(num):
    if num>0:
        return True
    else:
        return False

def test_01():
    """判断是不是为真"""
    a = 2
    b = 0
    assert is_true(a)
    assert not is_true(b)

def test_02():
    """判断b包含a"""
    a = "hello"
    b = "hello world"
    assert a in b

def test_03():
    """判断是否相等"""
    a = b = "hello"
    c = "hello world"
    assert a == b
    assert a != c
```

备注

- unittest框架实现结果断言，pytest可以直接兼容运行
- ```
pytest.main()
```

## 8.2 结果断言验证

- 结果断言封装

```
from utils.LogUtil import my_log
import json
#1、定义封装类
class AssertUtil:
#2、初始化数据, 日志
 def __init__(self):
 self.log = my_log("AssertUtil")
#3、code相等
 def assert_code(self,code,expected_code):
 """
 验证返回状态码
 :param code:
 :param expected_code:
 :return:
 """
 try:
 assert int(code) == int(expected_code)
 return True
 except:
 self.log.error("code error,code is %s,expected_code is %s"%(
 code,expected_code))
 raise
#4、body相等
 def assert_body(self,body,expected_body):
 """
 验证返回结果内容相等
 :param body:
 :param expected_body:
 :return:
 """
 try :
 assert body == expected_body
 return True
 except:
 self.log.error("body error,body is %s,expected_body is %s"%(
 body,expected_body))
 raise
#5、body包含
 def assert_in_body(self,body,expected_body):
 """
 验证返回结果是否包含期望的结果
 :param body:
 :param expected_body:
 :return:
 """
 try:
 body = json.dumps(body)
 print(body)
 assert expected_body in body
 return True
 except:
 self.log.error("不包含或者body是错误, body is %s,expected_body is %s"%(
 body,expected_body))
 raise
```

```
AssertUtil().assert_code(res["status_code"], status_code)
AssertUtil().assert_in_body(str(res["result"]), expect)
```

## 8.3 数据库结果验证

PyMySQL是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2中是使用mysqldb。

- PyMySQL安装

- `pip3 install pymysql`

- PyMySQL简单使用

```
导入pymysql模块
import pymysql

连接database
conn = pymysql.connect(
 host="数据库地址",
 user="用户名",
 password="密码",
 database="数据库名",
 charset="utf8")

获取执行SQL语句的光标对象
cursor = conn.cursor() # 结果集默认以元组显示
获取执行SQL语句，结果作为字典返回
cursor = conn.cursor(cursor=pymysql.cursors.DictCursor)

定义要执行的SQL语句
sql = "select username,password from tb_users"

执行SQL语句
cursor.execute(sql)
执行
cursor.fetchone()
关闭光标对象
cursor.close()

关闭数据库连接
conn.close()
```

- PyMySQL封装

```
from utils.LogUtil import my_log
import pymysql
#1、创建封装类
class Mysql:
#2、初始化数据，连接数据库，光标对象
 def __init__(self,host,user,password,database,charset="utf8",port=3306):
 self.log = my_log()
 self.conn = pymysql.connect(
 host=host,
 user=user,
 password=password,
 database=database,
```



```

 port=port
)
 self.cursor = self.conn.cursor(cursor=pymysql.cursors.DictCursor)
#3、创建查询、执行方法
def fetchone(self,sql):
 """
 单个查询
 :param sql:
 :return:
 """
 self.cursor.execute(sql)
 return self.cursor.fetchone()

def fetchall(self,sql):
 """
 多个查询
 :param sql:
 :return:
 """
 self.cursor.execute(sql)
 return self.cursor.fetchall()

def exec(self,sql):
 """
 执行
 :return:
 """
 try:
 if self.conn and self.cursor:
 self.cursor.execute(sql)
 self.conn.commit()
 except Exception as ex:
 self.conn.rollback()
 self.log.error("Mysql 执行失败")
 self.log.error(ex)
 return False
 return True

#4、关闭对象
def __del__(self):
 #关闭光标对象
 if self.cursor is not None:
 self.cursor.close()
 #关闭连接对象
 if self.conn is not None:
 self.cursor.close()

if __name__ == "__main__":
 mysql = Mysql("211.103.136.242",
 "test",
 "test123456","meiduo",
 charset="utf8",
 port=7090)
 #res = mysql.fetchall("select username,password from tb_users")
 res = mysql.exec("update tb_users set first_name='python' where username =
'python'")
 print(res)
 #1、创建db_conf.yml, db1,db2
 #2、编写数据库基本信息

```

- 数据库配置文件

```
db_1:
 db_host: "211.103.136.242"
 db_user: "test"
 db_password: "test123456"
 db_name: "meiduo"
 db_charset: "utf8"
 db_port: "7090"

db_2:
 db_host: "111211.103.136.242"
 db_user: "test"
 db_password: "test123456"
 db_name: "meiduo"
 db_charset: "utf8"
 db_port: "7090"

db_3:
 db_host: "333211.103.136.242"
 db_user: "test"
 db_password: "test123456"
 db_name: "meiduo"
 db_charset: "utf8"
 db_port: "7090"
```

- 数据库初始化

```
#1、定义init_db
def init_db(db_alias):
#2、初始数据化信息，通过配置
 db_info = ConfigYaml().get_db_conf_info(db_alias)
 host = db_info["db_host"]
 user = db_info["db_user"]
 password = db_info["db_password"]
 db_name = db_info["db_name"]
 charset = db_info["db_charset"]
 port = int(db_info["db_port"])
#3、初始化mysql对象
 conn = Mysql(host,user,password,db_name,charset,port)
 print(conn)
 return conn
```

- 数据库结果验证

```
def assert_db(db_name,result,db_verify):
 assert_util = AssertUtil()
 #sql = init_db("db_1")
 sql = init_db(db_name)
 # 2、查询sql, excel定义好的
 db_res = sql.fetchone(db_verify)

 #log.debug("数据库查询结果: {}".format(str(db_res)))
 # 3、数据库的结果与接口返回的结果验证
```

# 获取数据库结果的key

```
verify_list = list(dict(db_res).keys())
根据key获取数据库结果，接口结果
for line in verify_list:
 #res_line = res["body"][line]
 res_line = result[line]
 res_db_line = dict(db_res)[line]
 # 验证
 assert_util.assert_body(res_line, res_db_line)
```

## 9 数据驱动

### 9.1 yaml数据驱动

目标

- 掌握使用yaml进行数据驱动

定义数据文件

- 分模块定义数据文件
  - yaml文件

```
#TestLogin.yml

"case": "test_login_1"
"url": "authorizations"
"data":
 username: "python"
 password: "12345678"
"expect": "'user_id': 1, 'username': 'python'"

"case": "test_login_2"
"url": "authorizations"
"data":
 username: "test123"
 password: "test123"
"expect": "无法使用提供的认证信息登录"
```

@pytest.mark.parametrize实现参数化

- 接口用例

```
#登录
login_yaml = "TestLogin.yml"
data_path = Conf.get_data_path()

filename = data_path+os.sep + login_yaml
#print(filename)
def get_login_info():
 return YamlReader(filename).data_all()

@pytest.mark.parametrize("login",get_login_info())
def test_1(login):
 url = login["url"]
```

```

data = login["data"]
print("测试用例中login的返回值:%s%s" % (url, data))
#my_log().info("test_1")

if __name__ == "__main__":
 pytest.main(["-s", "analysis_yaml_data.py"])

```

## 9.2 Excel数据驱动

- 掌握使用excel进行数据驱动

### 9.2.1 Excel用例格式设计

- 直接使用功能测试
  - 功能用例图片
- 分析并增加额外的测试项
  - 分析接口测试需求
  - 增加后的图片

### 9.2.2 读取Excel

- 读取

```

#1、导入包, xlrd
import xlrd
#2、创建workbook对象
book = xlrd.open_workbook("testdata.xlsx")
#3、sheet对象
#索引
#sheet = book.sheet_by_index(0)
#名称
sheet = book.sheet_by_name("美多商城接口测试")
#4、获取行数和列数
rows = sheet.nrows #行数
cols = sheet.ncols #列数
#5、读取每行的内容
for r in range(rows):
 r_values = sheet.row_values(r)
 #print(r_values)
#6、读取每列的内容
for c in range(cols):
 c_values = sheet.col_values(c)
 #print(c_values)
#7、读取固定列的内容
print(sheet.cell(1,1))

```

- 定义数据

- 解耦合

```

#定义类
class DataConfig:
 #定义列属性

```

| #用例ID | 模块   | 接口名称 | 请求URL | 前置条件 | 请求类型    | 请求参数类型  |               |
|-------|------|------|-------|------|---------|---------|---------------|
| #请求参数 | 预期结果 | 实际结果 | 备注    | 是否运行 | headers | cookies | status_code 数 |

数据库验证

```
#用例ID
case_id = "用例ID"
case_model = "模块"
case_name = "接口名称"
url = "请求URL"
pre_exec = "前置条件"
method = "请求类型"
params_type = "请求参数类型"
params = "请求参数"
expect_result = "预期结果"
actual_result = "实际结果"
is_run = "是否运行"
headers = "headers"
cookies = "cookies"
code = "status_code"
db_verify = "数据库验证"
```

- 封装excel工具类

```
import os
import xlrd

#目的: 参数化, pytest list
#自定义异常
class SheetTypeError:
 pass
#1、验证文件是否存在, 存在读取, 不存在报错
class ExcelReader:
 def __init__(self, excel_file, sheet_by):
 if os.path.exists(excel_file):
 self.excel_file = excel_file
 self.sheet_by = sheet_by
 self._data = list()
 else:
 raise FileNotFoundError("文件不存在")
#2、读取sheet方式, 名称, 索引
 def data(self):
 #存在不读取, 不存在读取
 if not self._data:
 workbook = xlrd.open_workbook(self.excel_file)
 if type(self.sheet_by) not in [str, int]:
 raise SheetTypeError("请输入Int or Str")
 elif type(self.sheet_by) == int:
 sheet = workbook.sheet_by_index(self.sheet_by)
 elif type(self.sheet_by) == str:
 sheet = workbook.sheet_by_name(self.sheet_by)
#3、读取sheet内容
 #返回list, 元素:字典
 #格式[{"a": "a1", "b": "b1"}, {"a": "a2", "b": "b2"}]
 #1. 获取首行的信息
 title = sheet.row_values(0)
 #2. 遍历测试行, 与首行组成dict, 放在list
 #1 循环, 过滤首行, 从1开始
 for col in range(1, sheet.nrows):
 col_value = sheet.row_values(col)
 #2 与首组成字典, 放list
 self._data.append(dict(zip(title, col_value)))
```

```
#4、结果返回
 return self._data
if __name__ == "__main__":
 reader = ExcelReader("../data/testdata.xlsx", "美多商城接口测试")
 print(reader.data())
```

- 封装获取接口数据

```
class Data:

 def __init__(self, filename):
 self.reader = ExcelReader(filename).data

 def get_run_data(self):
 """
 获取运行用例，过滤is_run字段，只取yes
 # run_list=[]
 # for data in self.reader:
 # if str(data[is_run]).lower()=="y":
 # run_list.append(data)
 """
 is_run = data_config.is_run
 #使用列表推导式：等同于上面注释代码
 run_list=[data for data in self.reader if
str(data[is_run]).lower()=="y"]
 return run_list
```

- 参数化运行

```
def run_list():
 get_data = Data(data_config.case_file)
 return get_data.get_run_data()

def run_api(url, method, data=None, json=None, headers=None, cookies=None):
 request = RequestUtil.Request()
 if str(method).lower()=="get":
 res = request.get(url,
data=data, json=json, headers=headers, cookies=cookies)
 elif str(method).lower() == "post":
 res = request.post(url,
data=data, json=json, headers=headers, cookies=cookies)
 else:
 print("错误请求method")
 return "-1"
 return res

@pytest.mark.parametrize("case", run_list())
def test_run(case):
 url = case[config.url]
 method = case[config.method]
 request_data = case[config.params]
 expect = case[config.expect_result]
 headers = case[config.headers]
 cookies = case[config.cookies]
 case_id = case[config.case_id]
 case_name = case[config.case_name]
 status_code = case[config.status]
 params_type = case[config.params_type]
 log.info("运行用例名称: %s"%case_name)
```

```
if cookies:
 cookie = json.loads(cookies)
else:
 cookie = cookies

if headers:
 header = json.loads(headers)
else:
 header = headers
if str(params_type).lower() == "json":
 if len(str(request_data).strip()) is not 0:
 request_data = json.loads(request_data)
 res = run_api(url, method, json=request_data, headers=header,
cookies=cookie)
else:
 res = run_api(url, method, data=request_data, headers=header,
cookies=cookie)

AssertUtil().assert_code(res["status_code"], status_code)
AssertUtil().assert_in_body(str(res["result"]), expect)
```

## 10 Allure报告

### 10.1 Allure的安装及快速入门

#### 10.1.1 Allure介绍

[Allure](#)是一款非常轻量级并且非常灵活的开源测试报告生成框架。Allure 是一个独立的报告插件，生成美观易读的报告，它支持绝大多数测试框架，例如TestNG、Pytest、JUnit等。它简单易用，易于集成。

- 官网: <http://allure.qatools.ru>
- 帮助文档: <https://docs.qameta.io/allure/>

#### 10.1.2 Allure安装

- 安装python插件
  - 使用命令安装

```
$ pip3 install allure-pytest
```
  - 下载源码安装
    - <https://pypi.org/project/allure-pytest>
- 安装allure
  - 下载: <https://bintray.com/qameta/generic/allure2>
  - 前置条件: 已部署java环境
  - 解压缩到一个目录 (不经常动的目录)
  - 将压缩包内的 bin 目录配置到 path 系统环境变量
  - 在命令行中敲 allure 命令，如果提示有这个命令，即为成功

#### 10.1.3 Allure使用

- 示例代码
  - pytest.ini

[pytest]

```
添加行参数
addopts = -s --alluredir ./report/result
文件搜索路径
testpaths = ./scripts
文件名称
python_files = test_*.py
类名称
python_classes = Test*
方法名称
python_functions = test_*
```

- allure\_test1.py

```
import pytest

@pytest.fixture()
def sample():
 print('打印输出sample')

def test_01(sample):
 print('test1')

def test_02():
 print('test2----')

def test_03(sample):
 print('test3----')

if __name__ == '__main__':
 pytest.main(['allure_test1.py'])
```

- 运行结果，查看report/result目录
- 生成html

- 运行命令

```
$ allure generate report/result -o report/html --clean
```

- 使用火狐浏览器访问index.html，不要使用谷歌（谷歌是loading）
- 生成在线

```
allure serve report/result
```

## 10.2 Allure 详解

### 10.2.1 title 标题

可以自定义用例标题，标题默认为函数名。@allure.title

- 示例

```
import allure
import pytest

@pytest.mark.parametrize('title', ['用例标题0'])
def test_0():
 pass
```



```
def test_1():
 pass

def test_2:
 pass
```

- 结果

### 10.2.2 description 描述

可以添加测试的详细说明

- 示例

```
@allure.title("用例标题0")
@allure.description("这里是对test_0用例的一些详细说明")
def test_0():
 pass

@allure.title("用例标题1")
def test_1():
 """
 test_1的描述
 """
 pass

@allure.title("用例标题2")
def test_2():
 pass
```

- 结果

### 10.2.3 标签 @allure.feature

- 示例

```
@allure.feature('这里是一级标签: test')
class TestAllure:

 @allure.title("用例标题0")
 @allure.description("这里是对test_0用例的一些详细说明")
 def test_0(self):
 pass

 @allure.title("用例标题1")
 def test_1(self):
 pass

 @allure.title("用例标题2")
 def test_2(self):
 pass
```

- 结果

### 10.2.4 标签 @allure.story

- 示例

```
@allure.feature('这里是一级标签: test')
class TestAllure:
```

```
@allure.title("用例标题0")
@allure.description("这里是对test_0用例的一些详细说明")
@allure.story("这里是二级标签: test_0")
def test_0(self):
 pass
@allure.story("这里是二级标签: test_1")
@allure.title("用例标题1")
def test_1(self):
 pass

@allure.story("这里是二级标签: test_2")
@allure.title("用例标题2")
def test_2(self):
 pass
```

- 结果

### 10.2.5 标签 @allure.severity

定义用例的级别，Graphs主要有BLOCKER,CRITICAL,MINOR,NORMAL,TRIVIAL等几种类型，默认是NORMAL。

- 示例

```
@allure.feature('这里是一级标签: test')
class TestAllure:

 @allure.severity(allure.severity_level.BLOCKER)
 @allure.title("用例标题0")
 @allure.description("这里是对test_0用例的一些详细说明")
 @allure.story("这里是二级标签: test_0")
 def test_0(self):
 pass

 @allure.severity(allure.severity_level.CRITICAL)
 @allure.story("这里是二级标签: test_1")
 @allure.title("用例标题1")
 def test_1(self):
 pass

 @allure.severity(allure.severity_level.NORMAL)
 @allure.story("这里是二级标签: test_2")
 @allure.title("用例标题2")
 def test_2(self):
 pass
```

- 只运行指定级别的用例

```
--allure_severities=critical,blocker
```

### 10.2.6 动态生成 allure.dynamic

可以使用函数体内的数据动态生成

- 示例

```
params_1={"name":"动态获取test1","method":"post","url":"http://www.baidu.com"}
params_2={"name":"动态获取test2","method":"get","url":"http://www.baidu.com"}

@allure.title("用例标题0")
@allure.severity(severity_level=allure.severity_level.CRITICAL)
def test_0():
 pass
```

```
@allure.title("用例标题1")
def test_1():
 pass

@pytest.mark.parametrize("params",[params_1,params_2])
def test_2(params):
 allure.dynamic.title(params["name"])
```

- 结果

## 10.3 Allure应用接口

```
#allure
#sheet名称 feature 一级标签
allure.dynamic.feature(sheet_name)
#模块 story 二级标签
allure.dynamic.story(case_model)
#用例ID+接口名称 title
allure.dynamic.title(case_id+case_name)
#请求URL 请求类型 期望结果 实际结果描述
desc = "请求URL: {}
" \
 "请求类型: {}
" \
 "期望结果: {}
" \
 "实际结果: {}".format(url,method,expect_result,res)
allure.dynamic.description(desc)
```

## 10.4 Allure报告自动生成

### 配置文件pytest.ini

- 增加生成allure属性
- ```
addopts = -s --alluredir ./report/result
```

生成html格式报告

- 代码实现自动生成html报告

```
import os
import subprocess

def init_report():
    cmd = "allure generate report/result -o report/html --clean"
    subprocess.call(cmd, shell=True)
    res_path= os.path.abspath(os.path.dirname(__file__))
    report_path = res_path + "/report/" + "index.html"
```

设置程序主运行函数

- run.py
- ```
if __name__ == '__main__':
 report_path = Conf.get_report_path()+os.sep+"result"
 report_html_path = Conf.get_report_path()+os.sep+"html"
 pytest.main(["-s","--alluredir",report_path])
```

## - 邮件配置yaml格式

```
```yaml
#conf.yaml
mail:
  #发送邮件信息
  smtpserver : "smtp.itcast.cn"
  receiver : "*****@itcast.cn"
  username : "*****@itcast.cn"
  password : "*****"
```

• 邮件配置读取

```
#Conf.py
class ConfigYml:
    def __init__(self):
        logging.info(_config_file)
        #print(get_log_path())
        self.r_config = YamlReader(_config_file).data
        self.r_db_config = YamlReader(_config_db).data
    def get_mail_info(self):
        """
        获取邮件相关设置信息
        :return:
        """
        return self.r_config["mail"]
```

• 邮件读取封装

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

class SendMail(object):
    def __init__(self, username, passwd, recv, title, content,
                 file=None,
                 smtp_server='smtp.itcast.cn', port=25):
        self.username = username
        self.passwd = passwd
        self.recv = recv
        self.title = title
        self.content = content
        self.file = file
        self.smtp_server = smtp_server
        self.port = port
    def send_mail(self):
        msg = MIMEMultipart()
        #发送内容的对象
        if self.file: #处理附件的
            att = MIMEText(open(self.file).read())
            att["Content-Type"] = 'application/octet-stream'
            att["Content-Disposition"] = 'attachment; filename="%s"%self.file
            msg.attach(att)
        msg.attach(MIMEText(self.content)) #邮件正文的内容
        msg['Subject'] = self.title # 邮件主题
        msg['From'] = self.username # 发送者账号
        msg['To'] = self.recv # 接收者账号列表
        self.smtp = smtplib.SMTP(self.smtp_server, port=self.port)
```

#发送邮件服务器的对象

```
self.smtp.login(self.username,self.passwd)
try:
    self.smtp.sendmail(self.username,self.recv,msg.as_string())
except Exception as e:
    print('出错了。。',e)
else:
    print('发送成功!')
```

- 设置run.py

```
def send_email():
    mail_info = Conf.ConfigYml().get_mail_info()
    title= "接口自动化测试报告"
    username = mail_info["username"]
    passwd = mail_info["password"]
    receiver = mail_info["receiver"]
    smtp_server = mail_info["smtpserver"]
    content=""
    report_file= report_html_path+os.sep+"index.html"
    try:
        mail = SendMail(
            smtp_server=smtp_server,
            username=username,
            passwd=passwd,
            recv=receiver,
            title=title,
            content=content,
            file = report_file,
        )

        mail.send_mail()
    except:
        log.error('发送邮件失败，请检查邮件配置')
        raise
```

12 章总结

pyteset测试框架主要流程

- 读取Excel测试用例数据 -> 生成需要执行测试用例 -> 执行测试用例 (pytest+Request) -> 结果断言验证 -> 生成Allure报告 -> 邮件发送

重难点

- Requests使用【重点】【难点】
 - get
 - post
- pytest测试框架使用【重点】【难点】
 - 常用插件
 - 数据参数化
- 结果断言使用【重点】【难点】
 - Assert使用
 - 数据库断言
- 数据驱动使用【重点】【难点】

- Allure测试报告使用【重点】【难点】
 - Allure需要java环境
 - 动态生成相关信息
- 【易错点】：Allure生成报告时，如果想动态生成要使用allure.dynamic

```
allure.dynamic.feature(sheet_name)
allure.dynamic.title(case_id+case_name)
allure.dynamic.story(case_model)
```

拓展延伸

思考：请思考一下，。(必须提供)

延伸资料：给和课程相关的文章链接。(可选)

pycharm+git配置：<https://www.cnblogs.com/caseast/p/6085837.html>

日志：<https://www.cnblogs.com/qianyuliang/p/7234217.html>