接口自动化测试

一、接口自动化基础

1. 什么是接口测试

顾名思义，接口测试是对系统或组件之间的接口进行测试，主要是校验数据的交换，传递和控制管理过程，以及相互逻辑依赖关系。其中接口协议分为HTTP,WebService,Dubbo,Thrift,Socket 等类型,测试类型又主要分为功能测试,性能测试,稳定性测试，安全性测试等。

分层测试策略

分层测试策略中各层工作有明确的测试重心，测试工作通过逐层开展螺旋上升。这样一方面促使开发测试一体化，直接提高了测试效率；另一方面也可以尽早发现程序缺陷，降低缺陷修复成本。

在分层测试的 “金字塔” 模型中，接口测试属于第二层服务集成测试范畴，介于单元测试和界面测试之间，是一种灰盒测试方法，主要测试内部接口功能的完成性。相较于 UI 层（主要是 WEB 或 APP）自动化测试，接口自动化测试它具有自动化成本低和测试效率高的特点，收益更大，且容易实现，维护成本低，有着更高的投入产出比，是每个公司开展自动化测试的首选。

接口测试的工作原理

接口测试的工作原理是接口测试工具模拟客户端向服务器发送报文请求,服务器接受请求并做出响应。然后向客户端返回应答信息，接口测试工具对应答信息进行解析的一个过程。

常用的接口测试工具

1、Apache JMeter：是一款基于 Java 的开源测试工具，主要应用于 WEB 应用程序的负载测试，同时也支持单元测试和接口测试；

2、Postman：是一款功能强大的网页测试工具，支持 WEB API 和 HTTP 请求，能够发送任何类型的 HTTP 请求（GET、HEAD、POST、PUT 等）。Postwomen 与其近似的一款免费开源、轻量级测试工具；

3、SoapUI，是一款用于 SOAP 和 REST 的开源 API 测试自动化框架，可以集成到 Eclipse 等开发工具中，支持用户二次开发；

4、Robot Framework，是一款基于 Python3 的开源自动化测试框架，具有良好的可扩展性，支持关键字驱动，运行用户二次开发。

基于这些接口测试工具，测试人员可以根据自身业务需要开发适合自己的接口自动化测试工具。有了接口自动化测试工具，我们就可以开展自动化测试工作。

2. 接口自动化测试流程

基本的接口功能自动化测试流程如下：
需求分析 -> 用例设计 -> 脚本开发 -> 测试执行 -> 结果分析

1、在测试工具中登记待测交易的接口报文格式；

2、编写测试案例，向案例中添加交易接口并进行配置关联；

3、准备测试数据并对测试数据进行参数化；

4、测试工具自动执行自动化测试案例；

5、测试工具比对预期结果和返回结果，验证案例是否执行成功。

下面我们以一个 会员接口为例，完整的介绍接口自动化测试流程：从需求分析到用例设计，从脚本编写、测试执行到结果分析，并提供完整的用例设计及测试脚本。

示例接口：会员注册接口：

接口信息如下：

要求	描述
接口地址	http://hn216.api.yesapi.cn/?s=App.User.Register
接口描述	进行新用户注册，创建一个新的会员账号，注册密码需要使用 md5 后的密码
请求协议	Http/Https
请求方式	Get/Post
编码格式	Utf-8
返回格式	json

接口参数

参数名字	参数类型	是否必须	默认值	备注	参数说明
app_key	字符串	**必须**		最小：32	**公共参数** 开发者应用的 appkey，[查看我的 appkey](http://open.yesapi.cn/?r=App/Mine)，如果还没有，可免费注册开通。
sign	字符串	**可选**			**公共参数** 动态签名，签名生成算法请见：如何生成签名，或直接使用封装好的 SDK 开发包。 如果不需要签名，可进入小白开放平台接口签名设置关闭或开启签名。通过在线测试工具可进行签名的对比和调试。 企业版支持专属签名算法定制。
uuid	字符串	**可选**		最大：32	**公共参数** UUID，当前登录的应用会员 ID，即全局唯一用户 ID，查看我的应用会员。传递此参数后，可以在开放平台查看每日活跃会员统计图表。
token	字符串	**可选**		最小：64；最大：64	**公共参数** 当前登录会员的会话凭证，可通过会员登录接口获得。
return_data	字符串	**可选**	0		**公共参数** 数据返回结构，其中： **return_data=0**，返回完整的接口结果，示例：{"ret":200,"data":{"err_code":0,"err_msg":"","title":"Hi YesApi，欢迎使用小白开放接口！"},"msg":"V3.1.0 YesApi App.Hello.World"}；**return_data=1**，返回简洁的接口结果，只返回 data 字段，结构简化一级，更扁平，示例：{"err_code":0,"err_msg":"V3.1.0 YesApi App.Hello.World","title":"Hi YesApi，欢迎使用小白开放接口！"}。
username	字符	**必须**		最小：	注册成为应用的会员账号，注册后不可修改。

参数名字	参数类型	是否必须	默认值	备注	参数说明
	串			1；最大：50	
password	字符串	**必须**		最小：32；最大：32	MD5 后的密码，须 md5 后传递，保持全部小写，MD5 加密工具
ext_info	字符串 JSON 格式	**可选**			注册时的用户扩展信息，注册后可修改，需要 JSON 编码后传递。格式：ext_info={"扩展字段名":"值"}，可以同时更新多个字段。JSON 在线解析及格式化验证

返回结果：

返回字段	类型	说明
ret	整型	接口状态码，200 表示成功，4xx 表示客户端非法请求，5xx 表示服务端异常，查看异常错误码
data.err_code	整型	操作码，0 注册成功，1 已注册，－1 可用注册人数已用完
data.err_msg	字符串	错误提示信息，err_code 非 0 时参考此提示信息
data.uuid	字符串	全局唯一 UUID，全局唯一用户 ID，注册成功时返回
msg	字符串	提示信息，面向技术人员的帮助或错误提示信息

ret 异常错误码

错误码	错误类型	错误描述信息	解决方法
ret = 200	成功	请求成功	
ret = 400	客户端非法请	非法请求，参数错误	1、根据接口文档的接口参数，提供正确的参数

错误码	错误类型求	错误描述信息	解决方法
ret = 404	客户端非法请求	接口服务不存在	1、查看小白接口大全，确保接口服务名称拼写正确
ret = 500	小白接口异常	如果出现此错误,请联系技术人员处理	1、进入 QQ 交流群 897815708，反馈问题；2、或者提交工单，一天内回复
ret = 401	客户端非法请求	用户未登录,或登录凭证已过期	1、如果用户未注册，请先用注册接口；2、如果注册未登录或会话过期，请先用登录接口
ret = 406	客户端非法请求	非法 app_key，请核对你所在的接口域名	1、在我的套餐查看并核对接口域名
ret = 407	客户端非法请求	当前应用已过期	1、进行续费或续约，延长应用有效期
ret = 408	客户端非法请求	当前应用存在异常,已被封号	1、联系在线客服人工处理
ret = 409	客户端非法请求	签名错误	1、使用在线测试，校正签名；2、关闭特定或全部签名设置；3、如果过期，可以进行续约/升级
ret = 410	客户端非法请求	权限不足,非系统管理员	1、应用权限不足，可联系在线客服咨询沟通
ret = 411	客户端非法请求	权限不足,应用管理员未登录	1、请提供应用管理员的 adminuuid 和 admintoken 参数后重试
ret = 412	客户端非法请求	权限不足,非应用管理员	1、请在果创云把会员设置为管理员

错误码	错误类型	错误描述信息	解决方法
ret = 413	客户端非法请求	本月接口流量超出,已被临时冻结	1、开通接口流量叠加包，或升级套餐
ret = 414	客户端非法请求	并发过高	1、请降低请求频率，检测是否有死循环调用，或是否被他人恶意使用。如果流量确实很大，可联系在线客服进行定制化或私有云部署。
ret = 415	客户端非法请求	接口已被开发者关闭	1、接口已被开发者关闭，请进入果创云接口开关重新开启
ret = 416	客户端非法请求	权限不足,个人免费版套餐无法使用专业版接口	1、请先升级终身会员/标准版/旗舰

其他接口：

接口名称	接口地址	接口参数
会员登录接口	http://hn216.api.yesapi.cn/?s=App.User.Login	**app_key:usernamepassword**
获取会员个人资料接口	http://hn216.api.yesapi.cn/?s=App.User.Profile	**app_key、uuid、token**
会员搜索	http://hn216.api.yesapi.cn/?s=App.User.Search	**app_key**
获取会员列表接口	http://hn216.api.yesapi.cn/?s=App.User.GetList	**app_key**

3. 接口用例设计

用例设计是在理解接口测试需求的基础上,使用 MindManager 或 XMind 等思维导图软件编写测试用例设计，主要内容包括参数校验，功能校验、业务场景校验、安全性及性能校验等，常用的用例设计方法有等价类划分法，边界值分析法，场景分析法，因果图，正交表等。针对会员注册接口功能测试部分，我们主要从参数校验，功能校验，业务场景校验三方面，设计测试用例如下：

- 参数校验： 依次校验接口的各个参数的合法和非法情况（参数为空、输入值类型非法、输入值长度非法）;
- 功能校验：验证接口的功能需求业务逻辑情况；

接口用例设计思路

1）针对所有接口优先级

1、暴露在外面的接口，因为通常该接口会给第三方调用；

2、供系统内部调用的核心功能接口；

3、供系统内部调用非核心功能接口；

2）针对单个接口的优先级

1、正向用例优先测试,逆向用例次之；

2、是否满足前提条件 > 是否携带默认参值参数 > 参数是否必填 > 参数之间是否存在关联 > 参数数据类型限制 > 参数数据类型自身的数据范围值限制

通常，设计接口测试用例需要考虑以下几个方面：

- 是否满足前提条件

有些接口需要满足前置条件，才可成功获取数据。如常见的结合需要登录后的 Token。

逆向用例：针对是否满足前置条件(假设为 n 个条件)，设计 0~n 条用例

- 是否携带默认值参数

正向用例：带默认值的参数都不填写、不传参，必填参数都填写正确且存在的“常规”值，其它不填写，设计 1 条用例

- 参数是否必填

逆向用例：针对每个必填参数，都设计 1 条参数值为空的逆向用例

- 参数之间是否存在关联

有些参数彼此之间存在相互制约的关系，据此设计逆向用例：根据实际情况，可能需要设计 0~n 条用例

- 参数数据类型限制

参数存在数据类型限制的，比如只能输入整数的情况，

逆向用例：针对每个参数都设计 1 条参数值类型不符的逆向用例

- 参数数据类型自身的数据范围值限制（边界值）

正向用例：针对所有参数，设计 1 条每个参数的参数值在数据范围内为最大值的正向用例

逆向用例：针对每个参数(假设 n 个)，设计 n 条每个参数的参数值都超出数据范围最大值的逆向用例

针对每个参数(假设 n 个)，设计 n 条每个参数的参数值都小于数据范围最小值的逆向用例

3、业务规则、功能需求

这里根据实际情况，结合接口参数说明，可能需要设计 n 条正向用例和逆向用例

会员注册接口分析后得到以下的接口测试点：

考察点　　测试数据

所有参参　{'app_key':'5227C53B83002D99A28D874326F07BB6',

考察点	测试数据
数正确填写	'username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'}
app_key 为空	{'app_key':'', 'username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'}
app_key 非法	{'app_key':'5227C53B83002D99A28D874326F07**123**', 'username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'}
username 已存在	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'}
Username 为空	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'','password':'96e89a298e0a9f469b9ae458d6afae9f'}
username 长度非法	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head20211234567890'（**51 个字符**）,'password':'96e89a298e0a9f469b9ae458d6afae9f'}
password 密码字母大写	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head2021','password':'96E89A298E0A9F469B9AE458D6AFAE9F'}
password 为空	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head2021','password':''}
password 未加密	d{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head2021','password':'head1234'}
password 长度不合法	{'app_key':'5227C53B83002D99A28D874326F07BB6', 'username':'head2021','password':'8e0a9f469b9ae458'}
请求方式为 Get	
请求协议为 Http	

接口测试用例设计如下：

A	B	C	D	E	F	G	H	I	J	K	L	M	N
项目名称	模块名	用例编号	接口名称	用例标题	请求方式	请求URL	前置条件	请求参数	请求报文	返回报文	结果验证	测试结果	测试人员
电商项目	会员管理		App.User.Register	所有参参数正确填写	Post	http://hn216.api.yesapi.cn/?s=App.User.Register	N/A	app_key':'5227C53B83002D99A28D874326F07BB6','username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'	data={'app_key':'5227C53B83002D99A28D874326F07BB6','username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9f'}	{"ret":200,"data":{"err_code":0,"err_msg":"","uuid":"FBD3BEBFB63DFD1EC4F221E2B981A837"},"msg":"V3.3.0 YesApi App.User.Register"}	ret=200 err_code=0		

二、Python Requests 应用

Requests 库是一个优雅而简单的 Python HTTP 库，主要用于发送和处理 HTTP 请求。目前主流的接口自动化框架均基于 Requests 库进行开发。

Requests 的基本使用

Requests: 让 HTTP 服务人类

发行版本 v2.18.1. (安装说明)

license Apache 2.0 | wheel yes | python 2.7 | 3.5 | 3.6 | 3.7 | 3.8 | 3.9 | codecov unknown | Say Thanks!

Requests 唯一的一个**非转基因**的 Python HTTP 库，人类可以安全享用。

警告：非专业使用其他 HTTP 库会导致危险的副作用，包括：安全缺陷症、冗余代码症、重新发明轮子症、啃文档症、抑郁、头疼、甚至死亡。

看吧，这就是 Requests 的威力：

Requests 安装

一般都是直接采用 pip 命令安装：pip install requests

Requests 使用

官方文档的路径：https://cn.python-requests.org/zh_CN/latest/
在官方文档你可以看到关于 requests 的用法。

requests 下载后在使用的时候需要引入

```
import requests
```

接口自动化测试主要是基于 http 请求进行的，这里我们主要介绍 http 的 post，get 请求；

Requests 请求与应答

发送 get 请求：

```
response=requests.get(url,params=data)
```

发送 post 请求：

```
response = requests.post(url,json=data)
```

应答结果的话主要有三种应答方式：文本响应内容、二进制响应内容和 json 响应内容；

```
response.text  #文本响应内容

response.content #二进制响应内容

response.json() #json 响应内容
```

下面以查询天气预报接口为例进行演示：

```
import requests
url='http://hn216.api.yesapi.cn/?s=App.User.Register'
data={'app_key':'5227C53B83002D99A28D874326F07BB6',
              'username':'head2021','password':'96e89a298e0a9f469b9ae458d6afae9
f'}
r = requests.get(url,params=data)
print(r.text)
```

json 响应内容返回结果如下：

```
{"ret":200,"data":{"err_code":0,"err_msg":"","uuid":"B5BE5C29335D7CE3A2B60B80C4
```

DC490A"},"msg":"V3.3.0 YesApi App.User.Register"}

请求头信息

发送请求时若需要传递 header 请求头信息，可以参数传递进行

```
import requests
url = 'https://api.github.com/some/endpoint'
headers = {'user-agent': 'my-app/0.0.1'}

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

接口返回的结果如下

```
{"message":"Not Found","documentation_url":"https://docs.github.com/rest"}
```

响应头信息

获取应答响应头，可以通过 r.headers

```
print(r.headers)

{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'nginx/1.0.4',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json'
}
```

requests 上传文件

requests 上传文件需要把 传递 file 参数,注意：读取文件的时候要以 rb 模式读取：

```
upload_files = {'file': open('data.xls', 'rb')}
r = requests.post(url, files=upload_files)
```

requests 传递 cookie 参数

```
cookie = {'token': '12345', 'status': 'working'}
r = requests.get(url, cookies=cookie)
```

下面是封装的不保存 cookies 的类

```
class HttpRequest:
"""不保存 cookies 的请求类"""
    def get(self, url, param=None, headers=None, cookies=None):
     """get 请求"""
        res = requests.get(url, param=param, headers=headers, cookies=cookies)
```

```
        return res
    def post(self, url, data=None, json=None, param=None, headers=None, cookies
=None):
        """post 请求"""
        res = requests.post(url, data=data, json=json, param=param, headers=hea
ders, cookies=cookies)
        return res

    def visit(self, url, method, param=None, data=None, json=None, headers=None,
 cookies=None):
        """可在 method 参数选择 get、post 请求方式"""
        res = None
        if method == "POST" or method == "post":
            res = requests.post(url, param=param, data=data, json=json, headers
=headers, cookies=cookies)
        elif method == "GET" or method == "get":
            res = requests.get(url, param=param, headers=headers, cookies=cooki
es)
        return res

    def get_json(self, url, method, param=None, data=None, json=None, headers=N
one, cookies=None):
        """返回的是 json 格式的响应体"""
        res = None
        if method == "POST" or method == "post":
            res = requests.post(url, data=data, json=json, headers=headers, coo
kies=cookies)
        elif method == "GET" or method == "get":
            res = requests.get(url, param=param, headers=headers, cookies=cooki
es)
        return res.json()
```

以上是直接用的 requests 封装的类，其实还可以用 Session()类来封装，用 Session()类封装时有一个好处的，用这个类封装的话就不用去手动传 cookies 了。

Requests 的封装

get 方法封装

```
import requests

#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
```

post 方法封装

```
import requests

#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
```

requests 重构

```
#1、创建类
class Request:
#2、定义公共方法
    def __init__(self):
        self.log = my_log("Requests")
    def requests_api(self,url,data = None,json=None,headers=None,cookies=None,m
```

```
ethod="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=c
ookies)
    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/post 方法

基于上述重构的 request_api ，重新重构 get/post 方法。

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

三、pytest 测试框架

3.1 Pytest 基本介绍

3.1.1 Pytest 介绍

pytest 是一个非常成熟的全功能的 Python 的单元测试框架，同自带的 unittest 框架类似，但 pytest 框架使用起来更简洁，效率更高。

pytest 主要特点

- 简单灵活，容易上手，能够支持简单的单元测试和复杂的功能测试；
- pytest 具有丰富的第三方插件，如：pytest-selenium、pytest-html、pytest-rerunfailures、pytest-xdist 等
- 测试用例的 skip 和 xfail 处理，可以跳过指定用例，或对某些预期失败的 case 标记成失败。
- 可以很好的和 jenkins 集成，实现持续集成
- 支持与 allure 结合自动化生成测试报告
- 支持运行由 nose、unittest 编写的测试用例
- 支持参数化方式，ddt 数据驱动；

3.1.2. Pytest 安装

在 win+R 进入 cmd 命令行中运行以下命令：

```
>pip install -U pytest
```

检查是否安装了正确的版本：

```
> pytest --version
pytest 6.2.1
```

Pytest 官方文档：https://docs.pytest.org/en/latest/contents.html

3.1.3. Pytest 的简单实例

在 pytest 框架中，测试用例规范：

- 1、所有测试用例文件必须以 test 开头或者以 test 结尾，如：test_*.py 或 *_test.py ；
- 2、所有测试类必须以 Test 开头，并且不能带有 __init__() 方法;
- 3、测试函数或者测试类中的用例方法以 test_开头;
- 4、在执行 pytest 命令时，会自动从当前目录及子目录中寻找符合上述约束的测试函数来执行 ；

根据以上用例规范，在编辑器中创建一个简单的 pytest 的测试脚本：

```
import pytest

class TestDemo():
    def test_a(self):
        print('--> test a')
```

```
        assert 1 == 1
    def test_b(self):
        print(" --> test b")
        assert 2 < 1
if __name__ == '__main__':
    pytest.main(["-s", "test_demo.py"])
```

pytest 的运行模式

- **测试类主函数模式**：直接在 pycharm 中对测试用例脚本文件进行执行

```
pytest.main(["-s", "test_demo.py"])
```

参数说明：

-s 表示输出用例执行的详细结果;

test_demo.py 是要执行的脚本名称;

- **命令行模式**：在 cmd 命令行中进入文件所在文件目录，通过 pytest 文件路径 / 测试文件名对测试用例脚本文件进行执行

```
>pytest -s ./test_demo.py
```

执行结果如下：

```
C:\Python\Python39\python.exe C:/Users/lenovo/PycharmProjects/pytestDemo/test_case/test_demo.py
============================= test session starts =============================
platform win32 -- Python 3.9.1, pytest-6.2.2, py-1.10.0, pluggy-0.13.0
Using --randomly-seed=1617948941
rootdir: C:\Users\lenovo\PycharmProjects\pytestDemo\test_case
plugins: allure-pytest-2.8.16, cov-2.8.1, forked-1.1.3, html-2.1.1, instafail-0.4.1.post0, metadata-1.10.0, ordering-0.6, randomly-3.3.1, reportlog-0.1.2, sugar-0.9.4, xdist-1.31.0
collected 2 items

test_demo.py  --> test b
F--> test a
.

================================== FAILURES ===================================
_____________________________ TestDemo.test_b _____________________________

self = <test_demo.TestDemo object at 0x00000262A0A15D30>

    def test_b(self):
```

```
        print(" --> test b")
>       assert 2 < 1
E       assert 2 < 1

test_demo.py:15: AssertionError
========================== short test summary info ==========================
FAILED test_demo.py::TestDemo::test_b - assert 2 < 1
======================== 1 failed, 1 passed in 0.15s ========================

Process finished with exit code 0
```

Pytest Exit Code 含义清单

Exit Code	含义
Exit code 0	所有用例执行完毕，全部通过
Exit code 1	所有用例执行完毕，存在 Failed 的测试用例
Exit code 2	用户中断了测试的执行
Exit code 3	测试执行过程发生了内部错误
Exit code 4	pytest 命令行使用错误
Exit code 5	未采集到可用测试用例文件

3.1.4 Pytest 的执行控制

pytest 的运行模式也可以以**命令行模式**运行：

在 cmd 命令行中进入文件所在文件目录，通过 pytest 文件路径 / 测试文件名对测试用例脚本文件进行执行

```
>pytest -s ./test_demo.py
```

在运行测试脚本时，为了调试或打印一些内容，我们会在代码中加一些 print 内容，但是在运行 pytest 时，这些内容不会显示出来。如果带上-s，就可以显示了。运行模式：

```
>pytest test_demo.py -s
```

获取帮助信息

- 查看 pytest 版本 ： >pytest --version
- 显示可用的内置函数参数 : >pytest --fixtures
- 通过命令行查看帮助信息及配置文件选项 : >pytest --help

pytest 测试用例执行控制

- 执行失败后结束测试：在执行第 N 个用例失败后，结束测试执行；

```
>pytest -x              # 第 1 次失败，就停止测试

>pytest --maxfail=2     # 出现 2 个失败就终止测试
```

- 执行指定测试模块：执行对应的测试模块文件；

```
>pytest test_mod.py
```

- 执行指定测试目录：执行该测试目录下的所有符合条件的文件；

```
>pytest test_case/
```

- 执行通过关键字表达式过滤执行 ：执行符合匹配关键字的用例，包括：文件名、类名、方法名匹配表达式的用例

```
>pytest -k "MyClass and not method"
```

这里这条命令会运行 TestMyClass.testsomething ， 不会执行 TestMyClass.testmethod_simple

- 通过 node id 指定测试用例

nodeid 由模块文件名、分隔符、类名、方法名、参数构成，举例如下：
运行模块中的指定用例

```
>pytest test_mod.py::test_func
```

运行模块中的指定方法

```
>pytest test_mod.py::TestClass::test_method
```

- 通过标记表达式执行：通过给方法添加@pytest.mark.XXX，来指定执行用例，XXX 标记可以自定义设置；

```
>pytest -m slow
```

这条命令会执行被装饰器 @pytest.mark.slow 装饰的所有测试用例,这里只会执行 test_b()方法

- 通过包执行测试: 指定要执行的包

```
>pytest --pyargs test_case
```

这条命令会自动导入包 test_case，并使用该包所在的目录，执行下面的用例。

多进程运行 cases

当 cases 量很多时，运行时间也会变的很长，如果想缩短脚本运行的时长，就可以用多进程来运行。

使用多进程运行需要先安装 pytest-xdist： pip install -U pytest-xdist

运行模式：

```
>pytest test_se.py -n NUM
```

其中 NUM 填写并发的进程数。

重试运行 cases

在做接口测试时，有事会遇到 503 或短时的网络波动，导致 case 运行失败，而这并非是我们期望的结果，此时可以就可以通过重试运行 cases 的方式来解决。

使用重试运行 case 时需先安装 pytest-rerunfailures： pip install -U pytest-rerunfailures

运行模式：

```
>pytest test_se.py --reruns NUM
```

NUM 填写重试的次数。

- **显示 print 内容**

在运行测试脚本时，为了调试或打印一些内容，我们会在代码中加一些 print 内容，但是在运行 pytest 时，这些内容不会显示出来。如果带上-s，就可以显示了。运行模式：

```
pytest test_demo.py -s
```

另外，pytest 的多种运行模式是可以叠加执行的，比如，若同时运行 4 个进程，同时又打印出 print 的内容。可以用：

```
pytest test_demo.py -s -n 4
```

pytest 命令参数：

- -s ： 输出详细信息
- -x ： 失败 1 次停止
- -k ： 根据关键词执行
- -m: 根据标签执行用例
- -n: 多进程执行
- --reruns 3 :失败重新执行 3 遍
- --maxfail 3: 3 条用例失败停止执行
- --pyargs ： 执行指定包内的用例

3.2 Pytest 的 setup 和 teardown

在 unittest 中，setup 和 teardown 可以在每个用例前后执行，也可以在所有的用例集执行前后执行。

在 pytest 中有四种 setup 和 teardown，主要分为：模块级，类级，功能级，函数级。

1、setupmodule 和 teardownmodule 在整个测试用例所在的文件中所有的方法运行前和运行后运行，只会运行一次；

2、setupclass 和 setupclass 则在整个文件中的一个 class 中所有用例的前后运行，

3、setupmethod 和 teardownmethod 在 class 内的每个方法运行前后运行，

4、setupfunction、teardownfunction 则是在非 class 下属的每个测试方法的前后运行；

3.2.1 模块级别

模块基本就是在整个.py 测试脚本文件中的用例集开始前后，对应的是：

- setup_module ： 模块级别的 setup，在该脚本内所有用例集执行之前触发执行
- teardown_module ： 模块级别的 teardown，在该脚本内所有用例集执行之后触发执行

```
import pytest

def setup_module():
    print("模块开始")

def test_a():
    print('--> test a')
```

```
    assert 1 == 1

def test_b():
    print(" --> test b")
    assert 2 > 1

def teardown_module():
    print("模块结束")

class TestDemo():
    def test_demo(self):
        print("-->类中方法执行 test demo")
        assert 2==2

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

执行结果是：类中和非类中的方法 都被执行一遍，且 setupmodule() 和 teardownmodule() 只被执行一次;

3.2.2 类级别

类级别，必须使用在类里面，是在类中的所有用例集执行前后，对应的是：

- setup_class ：类级别的 setup，在该类中内用例集执行之前触发执行
- teardown_class ： 类级别的 teardown，在该类中内用例集执行之后触发执行

```
import pytest
class TestDemo():
    def setup_class(self):
        print("---- 类中用例集合执行开始")

    def test_a(self):
        print('--> test a')
        assert 1 == 1

    def test_b(self):
        print(" --> test b")
        assert 2 > 1
    def teardown_class(self):
        print("---- 类中用例集合执行结束")

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

执行结果是：setupclass() 和 teardownclass() 只被执行了一次；

3.2.3 类中方法级

方法级别，必须在类中对应的方法，在每一个方法执行前后，对应的是：

- setup_method：类中方法级别的 setup，在该类中内每个用例执行之前触发执行
- teardown_method：类中方法级别的 teardown，在该类中内每个用例执行之后触发执行

```
import pytest

class TestDemo():
    def setup_method(self):
        print(" \n 类中每个方法-----用例执行开始 - setup_method")

    def teardown_method(self):
        print(" \n 类中每个方法 -----用例执行结束- teardown_method")

    # @pytest.mark.skip(condition = "跳过此用例")
    def test_one(self):
        print('--> test one 111')
        assert 1 == 1

    def test_two(self):
        print(" --> test two 222")
        assert 2

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

执行结果是：类中的每个方法执行的时候 setupmethod()和 teardownmethod() 都会被执行一遍；

3.2.4 函数级别-非类中

函数级别，只对函数用例生效，而且不在类中使用，对应的是：

- setup_function：函数级别的 setup，在该脚本内每个用例函数执行之前触发执行
- teardown_function：函数级别的 teaddown，在该脚本内每个用例函数执行之后触发执行

```
import pytest

def setup_function():
    print("===非类中函数执行 setup_function")

def teardown_function():
    print("===非类中函数执行 teardown_function")

def test_func():
    print("===非类中函数执行")
```

```
class TestDemo():
    def test_demo(self):
        print("===类中方法执行 test demo")


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

执行结果是：只执行了 testfunc() 方法，类中的方法 testdemo()方法未被执行；

3.3 Pytest 配置文件

pytest 脚本有多种运行方式，如果处于 PyCharm 环境，可以使用右键或者点击运行按钮运行，也就是在 **pytest 中的主函数中运行：**

```
if __name__ == '__main__':
    pytest.main(["-s", "demo1.py"])   # 就是调用的 pytest 的 main 函数
```

也可以**在命令行中运行：**

```
>python demo1.py
```

这种方式，跟使用 Python 解释器执行 Python 脚本没有什么两样。也可以如下面这么执行：

```
>pytest -s demo1.py
```

当然，还有一种是使用配置文件运行，**通过配置便于命令行运行所用符合要求的测试用例。**

在项目的根目录下，我们可以建立一个 pytest.ini 文件，在这个文件中可以实现相关的配置：

```
[pytest]
addopts = -s -v
testpaths = ./scripts
python_files = test_*.py
python_classes = Test*
python_functions = test_*
```

注意:配置文件中不许有中文，pytest.ini 文件必须位于项目的根目录，而且也必须叫做 pytest.ini。

配置参数参数：

- addopts 可以搭配相关的参数，比如-s。多个参数以空格分割，其他参数后续用到再说。
  - -s，显示详细的 print 打印信息，没有-s 则 print 信息不会显示。
  - -v，使输出结果更加详细。
- testpaths 配置测试用例的目录，
  - 配置测试用例所在的文件目录，这个 scripts 就是我们所有文件或者目录的顶层目录。
  - 其内的子文件或者子目录都要以 test_开头，pytest 才能识别到。

- 另外，上面这么写，是从一个总目录下寻找所有的符合条件的文件或者脚本，那么我们想要在这个总目录下执行其中某个具体的脚本文件怎么办？

[pytest]
testpaths = ./scripts/
python_files = test_case_01.py
这么写就是执行 scripts 目录下面的 test_case_01.py 这个文件。

- python_classes 则是说明脚本内的所有用例类名的规则
  - 所有用例类名必须是以 Test 开头，也可以自定义为以 Test_开头
- python_functions 则是说脚本内的所有用例函数的命名规则
  - 所用测试用例方法必须以 test_开头才能识别。

3.4 Pytest 常用插件

插件列表网址：
包含很多插件包，大家可依据工作的需求选择使用。

前置条件

1.文件路径：

- Test_App
- - test_abc.py
- - pytest.ini

2.pyetst.ini 配置文件内容：

```
[pytest]
# 命令行参数
addopts = -s
# 搜索文件名
python_files = test_*.py
# 搜索的类名
python_classes = Test_*
#搜索的函数名
    python_functions = test_*
```

3.4.1 pytest-html 测试报告插件

pytest-HTML 是一个插件，pytest 用于生成测试结果的 HTML 报告。兼容 Python 2.7,3.6

安装方式：

```
pip install pytest-html
```

通过命令行方式，生成 xml/html 格式的测试报告，存储于用户指定路径。插件名称：pytest-html

使用方法： 命令行格式：pytest --html=用户路径/report.html

示例：

```
import pytest
class TestDemo():
```

```
    def setup_class(self):
        print("------->setup_class")
    def teardown_class(self):
        print("------->teardown_class")
    def test_a(self):
        print("------->test_a")
        assert 1
    def test_b(self):
            print("------->test_b")
            assert 0 # 断言失败```
```

运行方式：

1.修改 Test_App/pytest.ini 文件，添加报告参数，即：addopts = -s --html=./report.html

- -s:输出程序运行信息
- --html=./report.html 在当前目录下生成 report.html 文件
  若要生成 xml 文件，可将 --html=./report.html 改成 --html=./report.xml

2.命令行进入 Test_App 目录

3.执行命令： pytest

执行结果：

1.在当前目录会生成 assets 文件夹和 report.html 文件

3.4.2 pytest-rerunfailures 失败重试

失败重试意思是指定某个用例执行失败可以重新运行。

下载安装

```
pip install -U pytest-rerunfailures
```

使用

需要在 pytest.ini 文件中，给 addopts 字段新增（其他原有保持不变）--reruns=3 字段，这样如果有用例执行失败，则再次执行，尝试 3 次。 配置：

```
[pytest]
addopts = -s --html=report/report.html --reruns=3
```

代码实例如下：

```
import pytest

def test_case01():
    print('执行用例 01.......')
    assert 1  # 断言成功

def test_case02():
    print('执行用例 02.......')
```

```
    assert 0  # 断言失败，需要重新执行

class TestCaseClass(object):

    def test_case_03(self):
        print('执行用例 03.......')
        assert 1
```

我们也可以从用例报告中看出重试的结果。

失败重试包括两种情况：一种情况：用例失败了，然后重新执行多少次都没有成功。另一种情况，那就是用例执行失败，重新执行次数内通过了，那么剩余的重新执行的次数将不再执行。

3.4.3 pytest-ordering 控制用例执行顺序

如何手动控制多个用例的执行顺序，这里也依赖一个插件。

下载安装

```
>pip install pytest-ordering
```

使用实例

手动控制用例执行顺序的方法是在给各用例添加一个装饰器：

```
@pytest.mark.run(order=x)   # x 是一个整数
```

代码如下：

```
import pytest

class TestCaseClass(object):
    @pytest.mark.run(order=3)
    def test_case_03(self):
        print('执行用例 03.......')
        assert 1

    @pytest.mark.run(order=2)
    def test_case01():
        print('执行用例 01.......')
        assert 1  # 断言成功

    @pytest.mark.run(order=1)
    def test_case02():
        print('执行用例 02.......')
        assert 1  # 断言成功
```

那么， 现在的执行顺序是 2 1 3，按照 order 指定的排序执行的。

如果传个 0 或者负数啥的，那么它们的排序关系应该是这样的：

0 > 正数 > 没有参与的用例 > 负数
正数和负数就是按照大小关系排列的

3.4.4 pytest-xdist 并发执行

一条一条用例的执行，肯定会很慢，来看如何并发的执行测试用例，当然这需要相应的插件

下载安装

```
pip install -U pytest-xdist
```

使用

在 pytest.ini 配置文件中 addopts 添加 -n=auto , 修改配置如下：

```
[pytest]
addopts =  -v -s --html=report/report.html -n=auto
```

就是这个-n=auto：

- -n=auto，自动侦测系统里的 CPU 数目。
- -n=numprocesses，也就是自己指定运行测试用例的进程数。

并发的配置可以写在配置文件中，然后其他正常的执行用例脚本即可

```
import pytest

def test_case01():
    print('执行用例 01.......')
    assert 1  # 断言成功

@pytest.mark.skipif(condition= 2 > 1, reason='跳过用例')
def test_case02():
    print('执行用例 02.......')
    assert 0  # 断言失败

class TestCaseClass(object):

    def test_case_03(self):
        print('执行用例 03.......')
        assert 1

    def test_case_04(self):
        print('执行用例 04.......')
        assert 1
```

3.4.5 pytest-sugar

pytest-sugar 改变了 pytest 的默认外观，添加了一个进度条，并立即显示失败的测试。它不需要配置，只需 下载插件即可，用 pytest 运行测试，来享受更漂亮、更有用的输出。

安装下载

```
pip install -U pytest-sugar
```

其他照旧执行用例即可。

3.4.6 pytest-cov

pytest-cov 在 pytest 中增加了覆盖率支持，来显示哪些代码行已经测试过，哪些还没有。它还将包括项目的测试覆盖率。

下载

```
pip install -U pytest-cov
```

使用

在配置文件 pytest.ini 中配置--cov=./scripts,这样，它就会统计所有 scripts 目录下所有符合规则的脚本的测试覆盖率。

配置修改如下：

```
[pytest]
addopts =  -v -s --html=report/report.html -n=auto --cov=./scripts
```

执行的话，就照常执行就行。

3.5 Pytest 的高阶用法

3.5.1.Pytest 跳过用例 @pytest.mark.skip()：

跳过用例，我们使用@pytest.mark.skipif(condition, reason)：

- condition 表示跳过用例的条件。
- reason 表示跳过用例的原因。

然后将它装饰在需要被跳过用例的的函数上面。

```
import pytest

@pytest.mark.skip(condition='我就是要跳过这个用例啦')
def test_case_01():
    assert 1

@pytest.mark.skipif(condition=1 < 2, reason='如果条件为 true 就跳过用例')
def test_case_02():
    assert 1
```

3.5.4 Pytest 固件 @pytest.fixture()

固件（Fixture）是一些函数，pytest 会在执行测试函数之前（或之后）加载运行它们，也称测试夹具。

我们可以利用固件做任何事情，其中最常见的可能就是数据库的初始连接和最后关闭操作

fixture 修饰器来标记固定的工厂函数,在其他函数，模块，类或整个工程调用它时会被激活并优先执行,通常会被用于完成预置处理和重复操作。

方法：fixture(scope="function", params=None, autouse=False, ids=None, name=None)
常用参数:

- scope：被标记方法的作用域
  - “function" (default)：作用于每个测试方法，每个 test 都运行一次
  - "class"：作用于整个类，每个 class 的所有 test 只运行一次
  - "module"：作用于整个模块，每个 module 的所有 test 只运行一次
  - "session：作用于整个 session(慎用)，每个 session 只运行一次
- params：(list 类型)提供参数数据，供调用标记方法的函数使用
- autouse：是否自动运行,默认为 False 不运行，设置为 True 自动运行
- **简单的例子**

```
import pytest

@pytest.fixture()
def login():
    print("用户登录")

def test_home(login):
    print("主页")
```

- **fixture 第一个例子(通过参数引用)**

```
class Test_ABC():
    @pytest.fixture()
    def before(self):
        print("------->before")

    def test_a(self,before): #    test_a 方法传入了被 fixture 标识的函数，已变量的形式
        print("------->test_a")
        assert 1

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

```
.
```

- **fixture 第二个例子(通过函数引用)**

```
import pytest
```

```
@pytest.fixture() # fixture 标记的函数可以应用于测试类外部
def before():
    print("------->before")

@pytest.mark.usefixtures("before")
class Test_ABC():

    def setup(self):
        print("------->setup")

    def test_a(self):
        print("------->test_a")
        assert 1

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

      .
```

3.5.2 Pytest 标记预期失败 @pytest.mark.xfail()

所谓的预期失败，就是希望用例执行失败。这里用到了**@pytest.mark.xfail** 装饰器：

@pytest.mark.xfail 的含义是：期望测试用例是失败的，但是不会影响测试用例的的执行。如果测试用例执行失败的则结果是 xfail（不会额外显示出错误信息）；如果测试用例执行成功的则结果是 xpass。

@pytest.mark.xfail(condition, reason, [raises=None, run=True, strict=False])
需要掌握的必传参数的是：

- condition，预期失败的条件，当条件为真的时候，预期失败。
- reason，失败的原因。

那么关于预期失败的几种情况需要了解一下：

- 预期失败，执行成功：期望标记失败，实际执行成功，则输出 XPASS (reason)
- 预期失败，执行失败：期望标记失败，实际执行失败，则输出 XFAIL (reason)
- 预期成功，执行成功：期望标记成功，实际执行成功，则输出 PASSED
- 预期成功，执行失败：期望标记成功，实际执行失败，则输出 FAILED

```
import pytest


class TestCase(object):

    @pytest.mark.xfail(1 < 2, reason='预期失败， 执行失败')
    def test_case_01(self):
        """ 预期失败， 执行也是失败的 """
        print('预期失败， 执行失败')
        assert 0 > 1
```

```
    @pytest.mark.xfail(1 < 2, reason='预期失败， 执行成功')
    def test_case_02(self):
        """ 预期失败， 但实际执行结果却成功了 """
        print('预期失败， 执行成功')
        assert 0 < 1

    @pytest.mark.xfail(1 > 2, reason='预期成功， 执行成功')
    def test_case_03(self):
        """ 预期成功， 实际执行结果成功 """
        print('预期成功， 执行成功')
        assert 1==1

    @pytest.mark.xfail(1 > 2, reason='预期成功， 执行失败')
    def test_case_04(self):
        """ 预期成功， 但实际执行结果却失败了 """
        print('预期成功， 执行失败')
        assert 0 > 1

    def test_case_05(self):
        """ 普通的测试用例 """
        print('执行成功的普通用例')
        assert 1

    def test_case_06(self):
        """ 普通的测试用例 """
        print('执行失败的普通用例')
        assert 0
```

而在预期失败的两种情况中，我们不希望出现预期失败，结果却执行成功了的情况出现，因为跟我们想的不一样。预期这条用例失败，那这条用例就应该执行失败才对，你虽然执行成功了，但跟我想的不一样，你照样是失败的！

所以，我们需要将预期失败，结果却执行成功了的用例标记为执行失败，可以在 pytest.ini 文件中，加入：

```
[pytest]
xfail_strict=true
```

这样就就把上述的情况标记为执行失败了。

3.5.3 Pytest 参数化 @pytest.mark.parametrize()

pytest 身为强大的测试单元测试框架，那么同样支持 DDT 数据驱动测试的概念。在 pytest 中参数化测试，使用的工具就是 @pytest.mark.parametrize(argnames, argvalues)，即每组参数都独立执行一次测试。

- argnames 表示参数名。
- argvalues 表示列表形式的参数值。
- **单个参数**

```
import pytest

mobile_list = ['10010', '10086']

@pytest.mark.parametrize('mobile', mobile_list)
def test_register(mobile):
    """ 通过手机号注册 """
    print('注册手机号是: {}'.format(mobile))
```

多个参数的参数化

在多参数情况下，多个参数名是以,分割的字符串。参数值是列表嵌套的形式组成的。

多个参数时需要一一对应的，如我们注册的时候希望注册的手机号与验证码一一对应组合，可以如下实现：

```
mobile_list = ['15855667788', '18911223344']
code_list = ['7788', '3344']
@pytest.mark.parametrize('mobile, code', zip(mobile_list, code_list))
def test_register_mobile(mobile, code):
    """ 通过手机号注册 """
    print('注册手机号是: {} 验证码是: {}'.format(mobile, code))
```

函数返回值类型示例：

```
import pytest

def return_test_data():
    return [(1,2),(0,3)]

class Test_ABC():
    def setup_class(self):
        print("------->setup_class")

    def teardown_class(self):
         print("------->teardown_class")
@pytest.mark.parametrize("a,b",return_test_data()) # 使用函数返回值的形式传入参数值
def test_a(self,a,b):
    print("test data:a=%d,b=%d"%(a,b))
    assert a+b == 3
```

3.5.4 Fixture 作用域设置

Fixture 默认设置为运行

```
import pytest

@pytest.fixture(autouse=True) # 设置为默认运行
def before():
    print("------->before")

class TestABC():

    def setup(self):
        print("------->setup")

    def test_a(self):
        print("------->test_a")
        assert 1

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

```
.
```

fixture 设置作用域为 function

```
import pytest

@pytest.fixture(scope='function',autouse=True) # 作用域设置为 function，自动运行
def before():
    print("------->before")

class Test_ABC():
    def setup(self):
        print("------->setup")

    def test_a(self):
        print("------->test_a")
        assert 1

    def test_b(self):
        print("------->test_b")
        assert 1

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

Fixture 设置作用域为 class

```
    import pytest

    @pytest.fixture(scope='class',autouse=True) # 作用域设置为 class，自动运行
    def before():
        print("------->before")

    class Test_ABC():

        def setup(self):
            print("------->setup")

        def test_a(self):
            print("------->test_a")
            assert 1

        def test_b(self):
            print("------->test_b")
            assert 1


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

Fixture 返回值

```
import pytest

@pytest.fixture(params=[1, 2, 3])
def need_data(request): # 传入参数 request 系统封装参数
    return request.param # 取列表中单个值，默认的取值方式


class Test_ABC():
    def test_a(self,need_data):
        print("------->test_a")
        assert need_data != 3 # 断言 need_data 不等于 3

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

4.5.5 Pytest 预处理和后处理 yield

很多时候需要在测试前进行预处理（如新建数据库连接），并在测试完成进行清理（关闭数据库连接）。

当有大量重复的这类操作，最佳实践是使用固件来自动化所有预处理和后处理。

Pytest 使用 yield 关键词将固件分为两部分，yield 之前的代码属于预处理，会在测试前执行；yield 之后的代码属于后处理，将在测试完成后执行。

以下测试模拟数据库查询，使用固件来模拟数据库的连接关闭：

```
import pytest

@pytest.fixture()
def db():
    print('Connection successful')

    yield

    print('Connection closed')

def search_user(user_id):
    d = {
        '001': 'xiaoming',
        '002': 'xiaohua'
    }
    return d[user_id]

def test_case_01(db):
    assert search_user('001') == 'xiaoming'

def test_case_02(db):
    assert search_user('002') == 'xiaohua'
```

可以看到在两个测试用例执行前后都有预处理和后处理。

3.6 Pytest 与 Unittest 的区别

3.6.1 用例编写规则

1.unittest 提供了 test cases、test suites、test fixtures、test runner 相关的类,让测试更加明确、方便、可控。

使用 unittest 编写用例,必须遵守以下规则:

（1）测试文件必须先 import unittest

（2）测试类必须继承 unittest.TestCase

（3）测试方法必须以“test_”开头

（4）测试类必须要有 unittest.main()方法

2.pytest 是 python 的第三方测试框架,是基于 unittest 的扩展框架,比 unittest 更简洁,更高效。

使用 pytest 编写用例,必须遵守以下规则:

1）测试文件名必须以“test”开头或者"test"结尾(如:test_ab.py)

2）测试方法必须以“test_”开头

3）测试类命名以"Test"开头

总结: pytest 可以执行 unittest 风格的测试用例,无须修改 unittest 用例的任何代码,有较好的兼容性。 pytest 插件丰富,比如 flask 插件,可用于用例出错重跑;还有 xdist 插件,可用于设备并行执行。

3.6.2 用例前置和后置

1.unittest 前置和后置

（1）通过 setUp 每个用例执行前执行，tearDown 每个用例执行后执行；

（2）通过 setUpClass 类里面所有用例执行前执行，tearDownClass 类里面所有用例执行后执行；

2.pytest 前置和后置

pytest 提供了模块级、函数级、类级、方法级的 setup/teardown，比 unittest 的 setUp/tearDown 更灵活。

（1）模块级别：setupmodule/teardownmodule，整个.py 全部用例开始前执行/全部用例执行完后执行

（2）函数级别：setupfunction/teardownfunction，只对函数级别生效，每个用例开始前和结束后执行一次

（3）类级别：setupclass/teardownfunction,只对类级别生效，类里面所有用例开始前执行一次，所有用例执行完执行一次

（4）方法级别：setupmethod/teardownmethod,只是类里面方法级别生效，方法开始前执行一致，方法结束后执行一次

（5）方法级别：setup/teardown，这个与 setupmethod/teardownmethod 用法很类似，但是级别比 method 级别要低，也就是说在同一个方法中会先执行 setupmethod 再执行 setup，方法结束后先执行 teardown 再执行 teardownmethod

通过 firture 可以自定义 pytest 的前置和后置，格式 fixture(scope=”function”, params=None, autouse=False, ids=None, name=Noe)

- scope:有四个级别，function（默认）,class,module,session
- params:参数列表
- autouse:False 为默认值，意思代表需要根据设置的条件（scope 级别）来激活 fixture，如果为 Ture，则表示所有 function 级别的都被激活 fixture
- ids：每个字符串 id 的列表，感觉没啥实质性作用
- name：fixture 的名字

firture 相对于 setup 和 teardown 来说有以下几点优势：

- 命名方式灵活，不局限于 setup 和 teardown 这几个命名
- conftest.py 配置里可以实现数据共享，不需要 import 就能自动找到一些配置，可供多个 py 文件调用。
- scope="module" 可以实现多个.py 跨文件共享前置
- scope="session" 以实现多个.py 跨文件使用一个 session 来完成多个用例
- 用 yield 来唤醒 teardown 的执行

3.6.3 测试结果断言

1.unittest 提供了 assertEqual、assertIn、assertTrue、assertFalse。

- assertEqual(a, b) # 判断 a 和 b 是否相等
- assertNotEqual(a, b) # 判断 a 不等于 b
- assertTrue(a) # 判断 a 是否为 Ture
- assertFalse(a) #判断 a 是否为 False
- assertIn(a, b) # a 包含在 b 里面
- asserNotIn(a, b) # a 不包含在 b 里面

2.pytest 直接使用 assert 表达式。

pytest 只需要用 assert 来断言就行，assert 后面加需要断言的条件就可以了，例如：

- assert a = = b # 判断 a 是否等于 b
- assert a != b # 判断 a 不等于 b
- assert a in b # 判断 b 包含 a

总结：从断言上面来看，pytest 的断言比 unittest 要简单些，unittest 断言需要记很多断言格式，pytest 只有 assert 一个表达式，用起来比较方便

3.6.4 测试报告

1.unittest 使用 HTMLTestRunnerNew 库。

2.pytest 有 pytest-HTML、allure 插件。

3.6.5 用例失败重跑

1、unittest 无此功能。

2、pytest 支持用例执行失败重跑，pytest-rerunfailures 插件。

四、接口自动化框架搭建

在框架搭建之前，我们先梳理一下框架搭建的思路：

1. 使用 Python 的第三方 requests 库来处理接口的请求构建与响应结果获取；
2. 使用 pytest 来组建测试用例脚本；

3. 使用 YAML 文件来编写测试用例，这里的测试用例就是构建的请求数据和响应结果检查数据；
4. 使用 Python 的第三方 ddt 库作为测试用例的加载引擎；
5. 根据 YAML 编写的测试用例自动生成测试脚本；
6. 使用 Python 第三方 Allure 库作为结果的收集与展示。

YAML 安装：

```
pip install pyyaml
```

YAML 数据格式：

- 对象——字典格式 key: value
  {'name':'张三','age':18, 'mobile': '13112345678'}
  name: 张三
  age: 18
  mobile: 13112345678

- 数组 —— list 列表
  name = ['zhangsan'、'lisi' ,'wangwu']
  - zhangsan
  - lisi
  - wangwu

- 纯量 —— 字符串、数字、空
  "helloworld" , 19,
  string: helloworld
  number: 19
  ~

DDT : Data Driver Test 数据驱动测试

ddt 安装

```
pip install ddt
```

ddt 使用：

- @ddt : 使用在测试类上方
- @ddt.data() ：使用在具体测试方法上方，可传入单个测试数据，或多个测试数据（列表、元组、字典）；
- @ddt.unpack ： 使用在具体测试方法上方，data 传入多个数据时，需要进行解构

```
import ddt
from ddt import ddt
```

```
@ddt.ddt
class TestCase()
    @ddt.data(['zhangsan','lisi','wangwu'])
    @ddt.unpack
    def test_a(self,username):
        pass
```

- @ddt.file_data() ：使用在具体测试方法上方，传入文件数据

```
import ddt
@ddt.ddt
class TestCase()
    @ddt.file_data("c:/data/aa.yaml")
    @ddt.unpack
    def test_a(self,*case_data):
        data = case_data.get('data')
```

设计规范

pytest 的规范

- 文件规范 ：用例文件 必须以 test 开头：test_api_name.py
  - main 函数规范：每个用例文件里面需要有个
    if __name__ == '__main__':
        pytest.main(['-s','test_register.py'])
- 测试类规范：类名必须以 Test 开头的驼峰命名格式：class TestApiName()
- 用例方法规范：方法名称必须以 test 开头：`def testlogin():`
- ddt 的规范：
  - ddt 引入 import ddt ,使用时 @ddt.ddt ,@ddt.data() ,@ddt.file_data
  - ddt 引入 from ddt import ddt ,使用时 @ddt , @data, @file_data
  - @unpack 在 unittest 测试框架中解构生效，在 pytest 框架中 不生效；

规划框架结构

搭建框架根据分层的设计思想将整个项目分成不同的目录层级，便于团队成员协同工作，代码的分类放置，提高协同开发的效率。因此大概将项目分为以下目录层级：

```
-- AutoITF # 项目名，"简单的接口测试框架"
   -- bin   # bin 目录，用于存放启动文件，如 run.py
   -- cases # cases 目录，存放测试脚本， 也就是 unittest 的测试类文件
   -- data  # data 目录，存放测试用例，也就是 YAML 文件
   -- lib       # lib 目录，存放各种附加的代码文件，比如加密、连接数据库、生成测
```

试脚本等

-- report # report 目录，用于存放测试报告，各种日志文件等

-- setting.py # setting.py 文件，用于存放各种路径配置、服务器接口配置等等

-- case_template.txt # case_template.txt 文件，作为模板文件，生成测试脚本的模板

按照以上建好框架目录。

静态资源配置

我们先在 setting.py 文件中加入服务相关配置，如 APIURL、APPKEY、 APP_SECRET 等，其他文件直接引用即可；

```
### /setting.py
import os
# 服务相关配置
API_URL =  'http://hn216.api.okayapi.com/'
APP_KEY =  'DA8ED0D9F1D522934AFCB6552A45AD02'
# 不同的账号，key 不同，需要根据自己的修改
APP_SECRET =  'xWyS2Hh5ewZvSLC7PV0uJyo2uGZkKgOVOFxsbR'
# 不同的账号，secret 不同，需要根据自己的修改
# 设置目录的绝对路径
BASE_PATH =  os.path.dirname(os.path.abspath(__file__))
# yaml 测试用例存放位置
DATA_PATH =  os.path.join(BASE_PATH, 'data')
# unittest 测试类文件存放位置
CASE_PATH =  os.path.join(BASE_PATH, 'cases')
# 测试报告路径
REPORT_PATH =  os.path.join(BASE_PATH, 'report')
```

注意事项

注：BASE_PATH 这个变量主要用户获取当前项目的绝对路径，其他的目录可以通过绝对路径去运算。至于为什么要用这个绝对路径，这是为了保证代码在任何地方都能运行成功。

我们引入的时候，同一目录下可直接引入，在环境变量中的目录可直接引入，除此之外的其他目录引入都会出错。比如我们当前我们在 cases/test_login.py 文件中引入 setting.py，那么必须 setting.py 在 cases 目录或者 setting.py 所在的目录在环境变量中，否则无法引入成功。

对于这种情况我们就需要将项目所在的目录加入环境变量，因此这里我们在用路径的时候都用这个绝对路径去拼接。后面将这个绝对路径加入环境变量即可。

之所以我们在用 Pycharm 工具时不会出现这样的问题，是因为 Pycharm 会自动将项目目录加入环境变量。

附加资源文件

由于密码需要加密，且需要生成签名，我们先在 lib 目录下新建一个 utils.py 文件，用于处理签名和加密

在 cases 目录下新建 test_login.py

```
### /cases/test_login.py
import pytest
import requests
from lib.utils import  *
from setting import  *
class  Login():
    def test_login(self):
        # 构建参数
        data = {
        's': 'App.User.Login',
        'username': 'first',
        'password': 'asdf1234'
        }         # 加密密码
        if 'password' in data:
            data['password'] =  set_md5(data['password'])
            # 生成签名后重新组装 data
            data = set_request_data(data)
            # 根据方法构造请求
            res = requests.post(API_URL,  data=data)
            # res.text 比 res.json 的成功率高
            resp = res.text
            # 根据构造的结果，匹配结果
        check = ['ret=200', 'err_code=0',  '"err_msg=""]
        for c in check:
             assert c in resp
```

这里有个问题了，我们的预期结果 check = ['ret=200', 'errcode=0', '"errmsg=""]，逐一与实际结果匹配。而实际结果如下：

{ "ret":200, "data":{ "err_code":0, "err_msg":"", "uuid":"F774893AEF48CD0DBF77ACF8CEF4232C", "token":"9FDD3382419114E43616BF6D6E931851D0A8C66CAECF4C0C12CB0D9EF00928F2" }, "msg":"当前请求接口：App.User.Login" }

我们用 assert In 去断言，必定断言失败。但是这种断言方式又是比较简单，且通用性强的断言。我们只有将结果改造一下让结果的 JSON 格式数据变为：

{"ret=200,"data={"err_code=0,"err_msg=","uuid=f77...

这样我们就会很容易断言了。

在 lib 目录 utils.py 中**追加**一个函数：

```
### /lib/utils.py
    ...
# 前面的代码省略    # 返回的报文是 json 格式的，利用字符替换让数据变成"a=1,b=2"的格式
def  set_res_data(res):
    if res:
        return res.lower().replace('":"',  '=').replace('":', '=')
```

好了，现在修改一下测试用例文件 cases/test_login.py 将结果处理一下：

```
  ### /cases/test_login.py
    class  Login(unittest.TestCase):
        def test_login(self):
            ...
          # 前面的代码省略
        # 根据方法构造请求
        res = requests.post(API_URL, data=data)
        # res.text 比 res.json 的成功率高
        resp = res.text        ### 加了下面这一句 ###
        results = set_res_data(resp)
        # 根据构造的结果，匹配结果
        check = ['ret=200', 'err_code=0',  '"err_msg="']
        for c in check:
             assert  c In results
```

好，现在是不是就执行通过了？断言数据 check 变量中的三条预期都循环使用 assertIn 做了断言并且都通过了。

这只是一个单独测试的例子，我们思考一下，对于不同的用例，其实我们不同的是什么呢？

1. 数据不同，也就是 test_login.py 中的 data 变量不同，比如后面几条用例要设计错误的密码、错误的账号等，其实都是 data 里面封装的请求数据不同；
2. 断言结果不同,对于不同的测试用例,需要断言的结果不同。比如成功的 errcode=0，而失败的 errcode=1；
3. 请求方法可能不同，比如查询用 get， 增删改用 post 等；
4. url 也可能不同，很多接口用路由来定位接口，不同的接口 url 可能不同。小白接口的 url 都一样，这属于特例。
5. 用例的描述不同，我们学测试用例的时候知道，用例标题必须唯一，能够标识我们测试目的的。

构造测试数据

那么接下来，我们就根据不同的用例的数据不同，来构造测试用例的 YAML 文件。我们的最终目的是要测试用例与测试脚本分离。在 data 目录下新建 login.yaml:

```
### /data/login.yaml
- # 测试登录成功
    url:  http://hn2.api.okayapi.com/
    method: post
    data:
        s: App.User.Login
        username: first
        password: asdf1234
    check:
        - err_code=0
        - ret=200
        - "err_msg="
- # 测试用户名正确密码错误，登录失败
    url:  http://hn2.api.okayapi.com/
    method: get
    data:
        s: App.User.Login
        username: first001
        password: asdf1234
    check:
       - ret=200
       - err_code=1
       - err_msg=登录失败，账号不存在
```

根据我们前面对 YAML 的讲解，我们可以得出 login.yaml 转化成 Python 后，类似如下：

```
[      {
'url':'http://hn2.api.okayapi.com/'  ,
'method':'post',
'data': {
    's': 'App.User.Login',
    'username': 'first',
    'password': 'asdf1234'
    },
'check': [
        'ret=200',
        'err_code=0',
        'err_msg=',         ]
} ,
{
'url':'http://hn2.api.okayapi.com/',
```

```
'method':'get',
'data': {
    's': 'App.User.Login',
    'username': 'first001',
    'password': 'asdf1234'        },
        'check': [
        'ret=200',
        'err_code=1',
        'err_msg=登录失败，账号不存在'        ]      }      ]
```

也就是一个列表中多个字典，每个字典相当于是一条测试用例。

现在我们通过 ddt ，加载 YAML 文件中的用例数据，然后根据用例的条数（列表的元素个数）逐一执行测试用例。修改一下 cases/test_login.py(**把原来的文件重命名，新建一个文件来敲如下代码，改动较多)：

```
### /cases/test_login.py
import pytest
import requests
from lib.utils import *
import os
import ddt
@ddt.ddt
class Login():
    # 使用 ddt 加载 yaml 文件
    @ddt.file_data(os.path.join(DATA_PATH, 'login.yaml'))
    def test_login(self,  **case):  # 星号用于接收解包后的用例内容
        url =  case.get('url')
        method = case.get('method')
        data =  case.get('data')
        check =  case.get('check')        # 加密密码
        if 'password' in  data:
            data['password'] =  set_md5(data['password'])
            # 生成签名后重新组装 data
        data =  set_request_data(data)          # 根据方法构造请求
        try:
            if method.lower() ==  'post':
                res =  requests.post(url, data=data)
                resp =  res.text
            else:
                res =  requests.get(url, params=data)
                resp =  res.text
        except Exception as  e:
                print('接口请求出错！')
```

```
            resp = e            # 处理结果数据，方便比较
        results =  set_res_data(resp)           # 根据构造的结果,匹配结果
        for c in check:
            assert c in results
```

我们从 YAML 中获取到的数据类似[{}, {}...]这样的结构，通过 ddt 的处理，会变成 ddt.data({}, {}...)，参数中的一个字典就是一条用例。**参数中的字典元素必须与测试用例的参数个数一一匹配。我们这里直接用**case**参数来接收字典解包后的元素，相当于最终形成一个** case**字典传入测试用例方法内部。**

然后使用字典的取值方法 dict.get(key)，字典的取值方式有 dick.get(key)和 dict[key]两种方式。这两种方式的主要区别在于 dict[key]如果 key 不存在，就会抛出 KeyError 的异常，导致程序中断；而 dick.get(key)key 不存在的话会返回 None，不会抛出任何异常。一般根据情况选择一种方式即可。

为了避免因各种原因导致请求错误，把请求的异常捕获下来。

综上，就是一个示例用例了。接下来我们就要将示例用例作为一个模板，用它来根据不同的测试用例 YAML 文件生成不同的测试类文件。

接口模板处理

现在我们根据上一小节写的 cases/testlogin.py，用它作为模板，我们在根目录下建一个文件 casetemplate.txt 作为模板。内容直接从 cases/test_login.py 拷贝过来修改一下：

```
### /case_template.txt
import pytest
import requests
from lib.utils import  *
import os   import ddt
@ddt.ddt
class %(class_name)s():     # 使用 ddt 加载 yaml 文件
    @ddt.file_data(os.path.join(DATA_PATH, '  %(data_file)s.yaml'))
    def test_ %(method_name)s(self,  **case):  # 星号用于接收解包后的用例内容

        url = case.get('url')
        method = case.get('method')
        data = case.get('data')           # 加密密码
        if 'password' in data:
            data['password'] =  set_md5(data['password'])           # 生成签名后
重新组装 data
        data = set_request_data(data)           # 根据方法构造请求
        try:
            if method.lower() == 'post':
                res = requests.post(url,  data=data)
            resp = res.text
```

```
            else:
                res = requests.get(url,  params=data)
                resp = res.text
        except Exception as e:
            print('接口请求出错！')
            resp = e          # 处理结果数据，方便比较
            results = set_res_data(resp)            # 根据构造的结果,匹配结果

        check = case.get('check')
        for c in check:
            assert c In results
```

改的部分集中在类名、文件名和方法名的部分：

```
class  %(class_name)sTest():       # 使用 ddt 加载 yaml 文件          @ddt.file_da
ta(os.path.join(CASE_PATH, ' %(data_file)s.yaml'))
    def test_ %(method_name)s(self,  **case):
```

星号用于接收解包后的用例内容

- %(class_name)s， 用来替换类名，首字母大写
- %(data_file)s，用来替换文件名，全小写
- %(method_name)s，用来替换方法名，全小写

接下来我们在 lib 目录下新建一个 setcaseflie.py：

```
### /lib/set_case_file.py
from setting import  *
def  create_case_file():
"""
从 data 目录中找到所有的 yaml 文件
使用 case_template.txt 作为模板，生成测试用例文件
"""
    file_lists = os.listdir(DATA_PATH) # 取出 data 目录下的所有文件
    template_file = os.path.join(BASE_PATH,  'case_template.txt')
    for fList in file_lists:
        if fList.endswith('.yaml') or  fList.endswith('.yml'):
            # 测试用例文件名和 yaml 文件名
            data_file = fList.replace('.yaml',  '').replace('.yml', '')

            # 测试用例方法名
            test_method_name =  data_file.lower()  #方法名全小写
            # 测试用例类名
            test_class_name =  test_method_name.capitalize() #首字母大写

            with open(template_file, 'r',  encoding='utf-8') as temp:
```

```
# 从模板文件中取出内容
content = temp.read() % {
    'class_name':  test_class_name,
    'method_name':  test_method_name,
    'data_file':  data_file
}
test_case_file = 'test_%s.py' %  data_file
# 根据模板生成测试用例文件
with open(os.path.join(CASE_PATH, test_case_file),
          'w', encoding='utf-8') as f:
    f.write(content)
```

我们来解释一下 createcasefile()函数， 该函数的主要目的是从 data 目录下获取所有文件，筛选出其中的.yaml,.yml 文件，这两种文件格式都是 YAML 文件的后缀；用 base.txt 文件作为模板，替换其中的类名、文件名、方法名等；然后新建.py 文件，将模板内容写入.py 文件形成测试用例类文件。

大家可能要疑惑，为什么我们只需要改一下类名、文件名、方法名就能形成不同的测试用例呢？这就像我们在写功能测试用例的时候，我们很多时候操作步骤都是一样的，只是用例不同。而接口更是这样，无非就是调用接口，除了数据不同，其他的操作都一样。我们把不同的地方都抽象成了变量，并且放在了测试用例文件中，因此代码写成一样就行了。

接口测试报告

现在只需要再写一个 run.py 文件，用于查找用例、运行用例和生成报告，根据我们的结构设计，当然要放在 bin 目录下了。

```
>pytest
```

如此，在 run.py 中运行，即可执行用例并在 report 目录中生成测试报告。

到此，一个简单的接口测试框架就搭建好了。

五、接口自动化框架设计

1. 自动化测试框架分析

项目技术栈分析，需要使用到以下技术栈：

基础库	技能点
requests	requests 请求与应答、requests header、requests 封装
pytest	pytest 组织执行用例、pytest 参数化、pytest 断言
Allure	Allure 报告安装配置、Allure 应用（title ， description 知识点）、Allure 参数化断言

基础库　　技能点

Jenkins　Jenkins 安装配置、Jenkins 项目构建、Jenkins 邮件发送

其他处理　Excel 数据处理、log 日志处理、yaml DDT 数据驱动、PyMySQL 处理

首先考虑把 pytest 脚本跑起来，在这个基础上，在增加其他加功能：

- **Requests+Pytest 操作**
  - 构造基本的接口请求及相应：post/get 请求，相应结果 result.text ,result.json()
  - 数据的参数化：ddt + yaml 数据驱动，测试用例数据分离
  - 测试结果断言:对测试结果 assert 断言,测试断言数据分离,断言数据 yaml 参数化并遍历
- **使用 allure 生成测试报告**
- 测试报告 Allure 平台集成，进行 BDD 行为驱动测试；
  - 测试用例优化，为每一个请求用例添加 title 和 description
- 测试报告 allure 进行用例标识，指定执行对应用例
- **添加 log 日志**
  - 封装调用 logging
  - 在关键点添加上 log 日志，如：①. 请求的时候　②. 断言的时候　③. 可选打包的时候　④. 读 Excel 的时候
- **Jenkins 持续集成**
  - 对自动化用例持续构建，调用终端命令执行用例，对生成的 allure 报告进行邮件发送。
  - 执行终端命令：　①. subprocess ②. os.popen
  - 将测试报告发邮件：压缩文件、发送邮件、删除 report 文件夹

  注意：发邮件的时候，是不能发文件夹的， 解决办法是将 allure 报告文件夹打包成 zip， 发送 zip 文件。

  为了解耦合，需要遵循软件开发规范

  ①. 数据文件夹 ②. 配置文件夹 ③. 脚本文件夹

2. 自动化测试框架设计

建立目录结构

根据框架结构分析，依据**分层设计**的理念、**封装重构**的设计思想、**数据分离**的设计方法，搭建自动化测试的级别框架结构，对应的目录结构如下：

APISAutoTest

api: 主程序目录

comm：公共函数，包括：接口请求基类、请求及相应数据操作基类等

intf_handle：接口操作层，包含：接口初始化、断言等

business：业务实现部分

utils：工具类，包括：读取文件、发送邮件、excel 操作、数据库操作、日期时

间格式化等

config：配置文件目录，包含 yaml 配置文件、以及路径配置

data：测试数据目录，用于存放测试数据

temp：临时文件目录，用于存放临时文件

result：结果目录

report：测试报告目录，用于存放生成的 html 报告

details：测试结果详情目录，用于存放生成的测试用例执行结果 excel 文件

log：日志文件目录

test：测试用例、测试集相关目录，启动 test_suite 执行用例文件存放在此

test_case：测试用例存放路径

test_suite：测试模块集，按模块组装用例

上面就是整个接口自动化的框架示意图,当然具体的框架结构要结合具体的项目进行分析和设计，并不是一味的贪大求全，一切以适合为宜。

框架基本配置

配置 ini 文件
pytest.ini 文件中不要有中文，哪怕是注释的部分有中文也不行

```
[pytest]

addopts = -s -v -p no:warnings --alluredir=./report/allure_json
testpaths = ./test
python_files = test_*.py
python_classes = Test*
python_functions = test_*
```

基本路径的配置
创建 setting.py 文件，将项目的基本路径进行配置，便于后续使用；

```
import os

base_url = 'http://hn216.api.yesapi.cn/'
app_key = '5227C53B83002D99A28D874326F07BB6'

# 项目根路径
BASE_PATH = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# 测试用例路径
CASE_PATH = os.path.join(BASE_PATH, 'test_case')
```

```
# 测试数据路径
DATA_PATH = os.path.join(BASE_PATH, 'data')
# 测试模板路径
TEMPLATE_PATH = os.path.join(BASE_PATH, 'template.txt')
# 测试日志路径
LOG_PATH = os.path.join(BASE_PATH, 'log')
# 测试报告路径
REPORT_PATH = os.path.join(BASE_PATH, 'report')
# print(DATA_PATH)
```

建立入口文件
在项目根目录创建一个 run.py 文件，作为执行的入口文件

```
import pytest

if __name__ == '__main__':
    pytest.main()
```

测试用例优化
测用例的前置处理和后置处理的封装：

```
import pytest
class TestBase():
    def setup_class(self):
        pass
    def teardown_class(self):
        pass
    def setup_method(self):
        pass
    def teardown_method(self):
        pass
```

在测试用例类里面进行继承：

```
from test_case.test_api.test_Base import TestBase
class TestLogin(TestBase):
    def test_login(self):
        pass
```

测试用例的参数化封装：

```
import pytest
import requests
```

```
import ddt
from test_case.test_api.test_Base import TestBase

@ddt.ddt
class TestLogin(TestBase):
    @ddt.file_data(os.path.join(DATA_PATH, 'login.yaml'))
    def test_login(self, **case_data):
        # 获取测试数据
        data = case_data.get('data')
        expect = case_data.get('expected')
        method = case_data.get('method')
```

封装后的测试用例：

```
import pytest
import requests
import ddt
import os
from config.setting import DATA_PATH, base_url
from test_case.test_api.test_Base import TestBase
from utils.utils import result_set

@ddt.ddt
class TestLogin(TestBase):
    @ddt.file_data(os.path.join(DATA_PATH, 'login.yaml'))
    def test_login(self, **case_data):
        # 获取测试数据
        data = case_data.get('data')
        expect = case_data.get('expected')
        method = case_data.get('method')

        # 接口请求
        response = None
        if method == 'post':
            response = requests.post(url=base_url, json=data)
        elif method == 'get':
            response = requests.get(url=base_url, params=data)
        result = response.text
        #结果处理
        result = result_set(result)
        print(result)
        for ex in expect:
            assert ex in result

if __name__ == '__main__':
```

```
    pytest.main(['test_login.py'])
```

3. 自动化测试框架集成

log 日志功能集成

设置 logging 基本配置

```
# ---------------- 日志相关 -------------------
LOG_LEVEL = 'debug'     # 日志级别：
LOG_STREAM_LEVEL = 'debug'  # 屏幕输出流
LOG_FILE_LEVEL = 'info'   # 文件输出流
# 日志文件夹命名：
LOG_FILE_NAME = os.path.join(LOG_PATH, datetime.datetime.now().strftime('%Y-%m-%d') + '.log')
```

封装 logging 日志：

```
import logging
import os
import time
from config.setting import LOG_FILE_NAME

class Log(logging.Logger):
    def __init__(self, name):
        super().__init__(name=name)
        self.log_file_name = LOG_FILE_NAME
        self.f_handler = logging.FileHandler(filename=self.log_file_name, encoding='utf-8')
        self.s_handler = logging.StreamHandler()
        self.setLevel(logging.INFO)  # 日志级别
        self.log_fmt = "%(asctime)s, Module:%(name)s - [File:%(filename)s - Lines:%(lineno)d] - %(levelname)s ： %(message)s"
        self.formatter = logging.Formatter(self.log_fmt)

        if self.log_name:
            # 输出到 log 文件
            self.f_handler.setFormatter(self.formatter)
            self.f_handler.setLevel(logging.INFO)
            self.addHandler(self.f_handler)
            # 输出到控制台
            self.s_handler.setFormatter(self.formatter)
            self.s_handler.setLevel(logging.WARNING)
            self.addHandler(self.s_handler)
```

log 日志调用

测试用例中关键位置添加日志信息：

每个测试用例中都需要引入 log，且在每条测试用例执行的前后都需要调用，那么可以引入 pytest 的前置和后置处理，并将前置和后置处理封装为一个公共的 case 类，便于每个用例继承调用 log：

```
from utils.log import Log

class TestBase():
    def setup_class(self):
        pass
    def teardown_class(self):
        pass
    def setup_method(self):
        self.logger = Log('Api Test')
        self.logger.info('--- 测试执行开始 ---')

    def teardown_method(self):
        self.logger.info("--- 测试执行结束 -- ")
```

具体的每个测试用例继承封装好的公共的测试用例：

```
import pytest
import requests
import ddt
import os
from config.setting import DATA_PATH, base_url
from test_case.test_api.test_Base import TestBase
from utils.utils import result_set

@ddt.ddt
class TestLogin(TestBase):
    @ddt.file_data(os.path.join(DATA_PATH, 'login.yaml'))
    def test_login(self, **case_data):
        title = case_data.get('title')
        allure.dynamic.title(title)

        # 获取测试数据
        data = case_data.get('data')
        expect = case_data.get('expected')
        method = case_data.get('method')

        self.logger.info('接口名称：{}, --> 请求方式{}'.format(data['s'], metho
d))
        self.logger.info('测试用例标题: {}'.format(title))
```

```
        self.logger.info('测试数据：{}'.format(data))

        # 接口请求
        response = None
        if method == 'post':
            response = requests.post(url=base_url, json=data)
        elif method == 'get':
            response = requests.get(url=base_url, params=data)
        result = response.text
        #结果处理
        result = result_set(result)
        print(result)
        for ex in expect:
            assert ex in result

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

执行后日志截图：

六、Allure 测试报告集成

1 Allure 框架的简介

Allure 框架是一个灵活的轻量级多语言测试报告工具，它不仅以 web 的方式展示了简介的测试结果，而且允许参与开发过程的每个人从日常执行的测试中最大限度的提取有用信息。它支持绝大多数测试框架， 例如 TestNG、Pytest、JUint 等。它简单易用，易于集成。

pytest 中 allure 插件下载

```
pip install allure-pytest
```

但由于这个 allure-pytest 插件生成的测试报告不是 html 类型的，我们还需要使用 allure 工具再“加工”一下。所以说，我们还需要下载这个 allure 工具。

PS：allure 依赖 Java 环境，我们还需要先配置 Java 环境。

如果你的电脑已经有了 Java 环境，就无需重新配置了。配置完了 Java 环境，我们再来下载 allure 工具

Allure 下载地址：

https://github.com/allure-framework/allure2

https://bintray.com/qameta/maven/allure2

安装后打开你的终端验证 Allure 的版本如下：

```
C:\Users\Lenovo\Desktop>allure --version
2.10.0
```

Allure 的主要命令

- pytest --alluredir=./report/result （指定 allure 报告数据生成路径）
- allure serve ./report/html （生成 HTML 报告，这个会直接在线打开报告）
- allure generate ./report/result -o ./report/html --clean（指定生成报告的路径）
- allure open -h 192.168.1.5 -p 8888 ./report/html（启动本地服务生成链接查看报告）

```
allure generate report/results -o report/report_html --clean
```

```
allure serve report/results  report/html_report
```

Allure 安装注意事项

1、allure 依赖 Java 环境：需安装配置 Java 环境变量；

2、用 pip 与 pycharm 的集成 ：需要注意选择 python 本地解析解；

3、使用下载的 allure 包：需要配置 allure 环境变量；

2 Allure 的使用过程

一般 allure 使用要经历几个步骤：

1.配置 pytest.ini 文件。

2.编写用例并执行。

3.使用 allure 工具生成 html 报告。

6. **配置 pytest.ini 文件：**

在 addopts 选修上添加 --alluredir ./report/result ，如下：

```
[pytest]
addopts =  -v -s --html=report/report.html --alluredir=./report/result
```

7. **编写用例并执行用例**

在 cmd 命令行终端中输入 pytest 正常执行测试用例：

```
D:\WorkSpaces\pytestDemo>pytest
```

执行完毕后，在项目的根目下，会自动生成一个 report 目录，这个目录下有：

- report.html 是我们的之前的 pytest-html 插件生成的 HTML 报告，跟 allure 无关。
- result 和 assets 目录是 allure 插件生成的测试报告文件，但此时该目录内还没有什么 HTML 报告，只有一些相关数据。

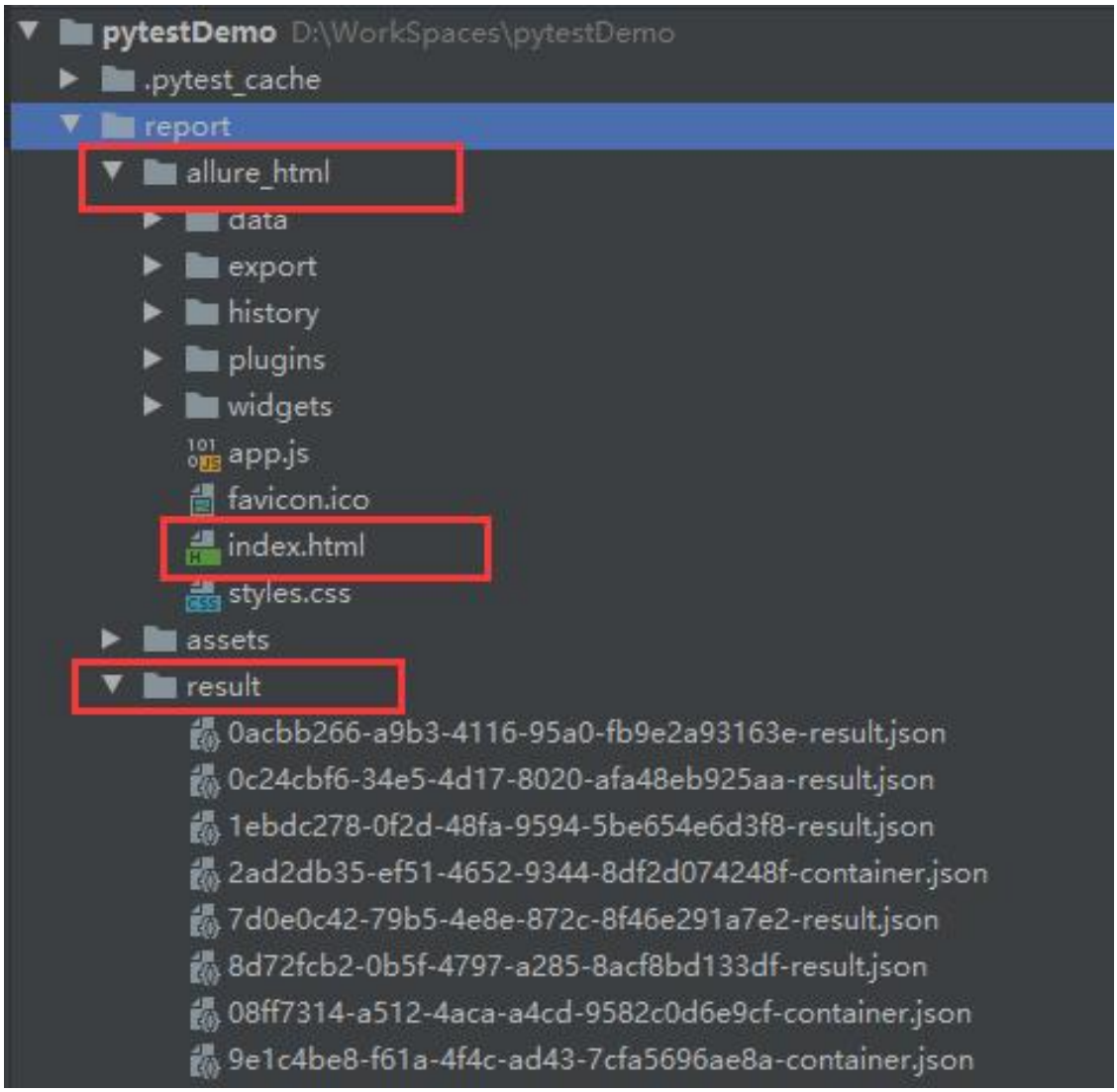
1. **使用 allure 工具生成 html 报告**

接下来需要使用 allure 工具来生成 HTML 报告。

此时我们在 cmd 终端，路径是项目的根目录，执行下面的命令:

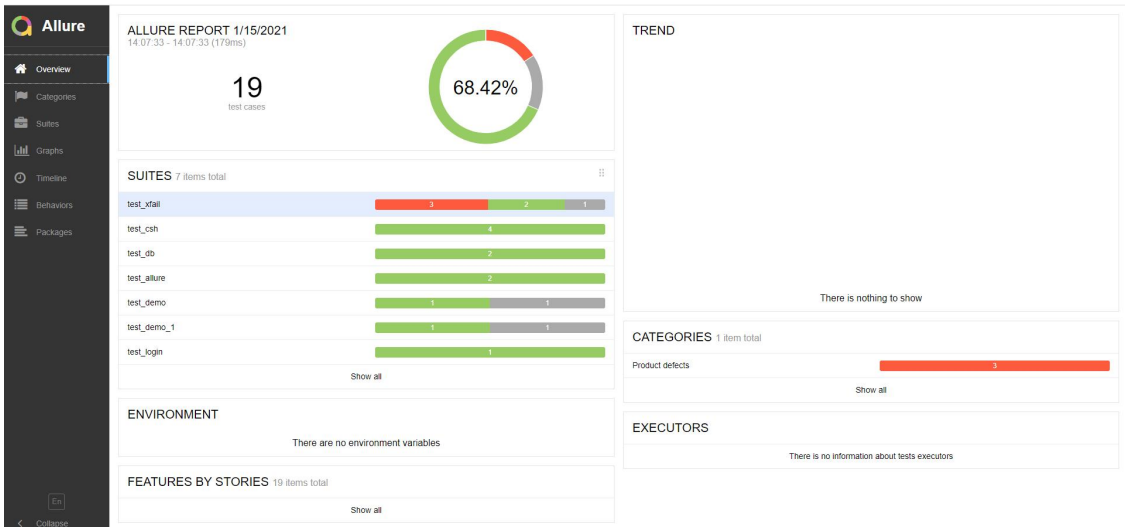
```
D:\WorkSpaces\pytestDemo> allure generate report/result -o report/allure_html --clean
```

执行后项目 report 目录下生成 allure_html 文件夹，里面就是生成的 Allure 报告：



命令的意思是，根据 report\result 目录中的数据（这些数据是运行 pytest 后产生的）。在 report 目录下新建一个 allure_html 目录，而这个目录内有 index.html 才是最终的 allure 版本的 HTML 报告；如果重复执行的话，使用--clean 清除之前的报告。

结果很漂亮：



3 Allure 的常用特性

在使用 allure 生成报告的时候，在编写用例阶段，还可以有一些特性参数可以使用：

- title，自定义用例标题，标题默认是用例名。
- description，测试用例的详细说明。
- feature 和 story 被称为行为驱动标记，因为使用这个两个标记，通过报告可以更加清楚的掌握每个测试用例的功能和每个测试用例的测试场景。或者你可以理解为 feature 是模块，而 story 是该模块下的子模块。
- allure 中对 severity 级别的定义：
  - Blocker 级别：中断缺陷（客户端程序无响应，无法执行下一步操作）
  - Critical 级别：临界缺陷（ 功能点缺失）
  - Normal 级别：普通缺陷（数值计算错误）
  - Minor 级别：次要缺陷（界面错误与 UI 需求不符）
  - Trivial 级别：轻微缺陷（必输项无提示，或者提示不规范）
- dynamic，动态设置相关参数。

allure.title 与 allure.description
在测试用例方法上方添加 @llure.title()和@allure.description() 装饰器描述测试用例的标题和详情，具体代码如下：

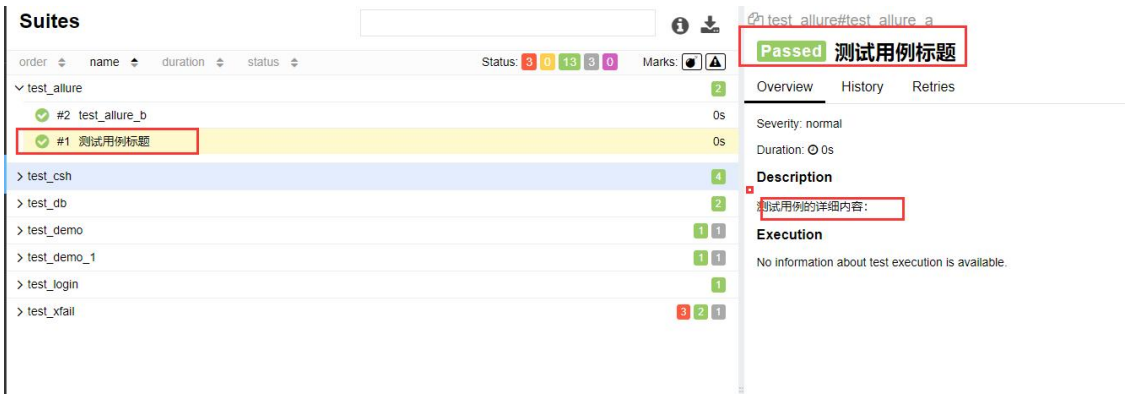
```
import allure
import pytest

@allure.title('测试用例标题')
@allure.description("测试用例的详细内容：")
def test_allure_a():
    print("allure aaa")
    assert 0 in [0, 21, 20]
```

```
def test_allure_b():
    print("allure bbb")
    assert '0' in '20208'
```

重新执行用例并生成 Allure 报告

```
D:\WorkSpaces\pytestDemo>pytest
D:\WorkSpaces\pytestDemo>allure generate report/result -o report/allure_html --clean
```

生成的报告如下：



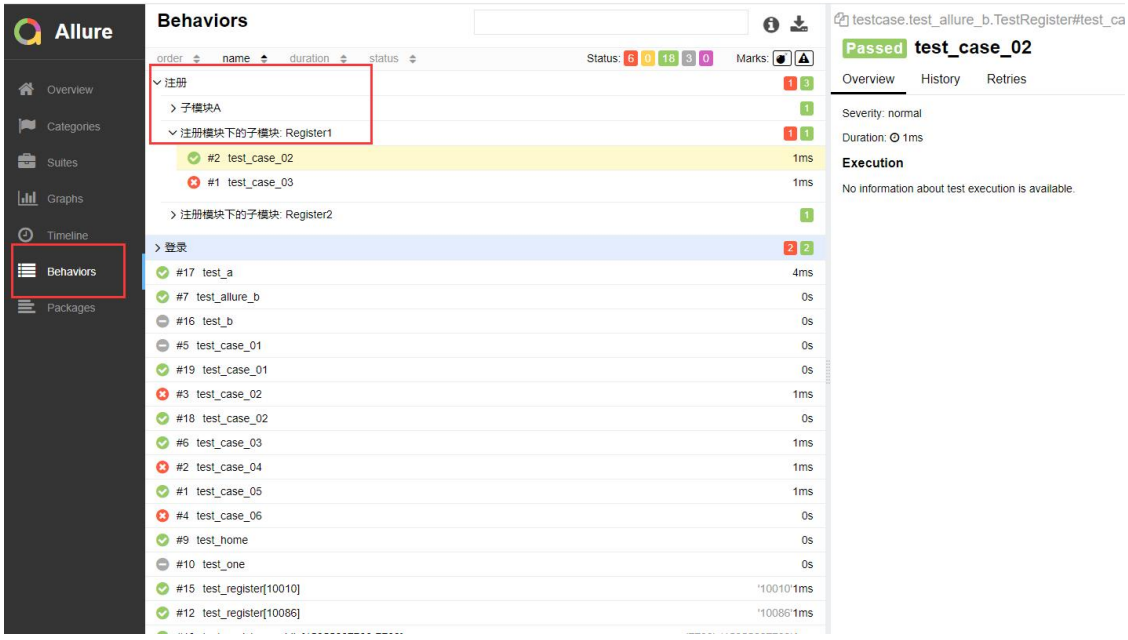
feature 和 story

在测试用例方法上添加@allure.feature() 和@allure.story() 装饰器，用来标识用例所属模块或子模块

```
import pytest
import allure

@allure.feature("登录")
# @allure.feature('登录模块')
class TestLogin():

    @allure.story('登录模块下的子模块: Login1')
    def test_case_01(self):
        assert 1>0

    @allure.story('登录模块下的子模块: Login1')
    def test_case_02(self):
        assert 1<0

    @allure.story('登录模块下的子模块: Login2')
    def test_case_03(self):
        assert 3>=3

    @allure.story('登录模块下的子模块: Login3')
```

```
    def test_case_04(self):
        assert 1==2

@allure.feature('注册')
# @allure.feature('注册模块')
class TestRegister():
    @allure.story("子模块 A")
    # @allure.story('注册模块下的子模块: Register1')
    def test_case_01(self):
        assert 1 in [1,3,4]

    @allure.story('注册模块下的子模块: Register1')
    def test_case_02(self):
        assert 'e' in 'hello'

    @allure.story('注册模块下的子模块: Register1')
    def test_case_03(self):
        assert '' is None

    @allure.story('注册模块下的子模块: Register2')
    def test_case_04(self):
        assert 16 > 10
```

重新执行后在 Behaviors 下看的模块及子模块的名称



allure.severity

allure.severity 用来标识测试用例或者测试类的级别，分为 blocker，critical，normal，

minor，trivial5 个级别。

```
import pytest
import allure

@allure.feature('商品模块')
class TestGoods(object):
    @allure.story("添加商品")
    @allure.severity(allure.severity_level.BLOCKER)
    def test_case_01(self):
        assert 1==0

    @allure.story("修改商品")
    @allure.severity(allure.severity_level.CRITICAL)
    def test_case_02(self):
        assert 1>0

    @allure.story('商品上架')
    @allure.severity(allure.severity_level.MINOR)
    def test_case_03(self):
        assert 1

    @allure.story("商品下架")
    @allure.severity(allure.severity_level.TRIVIAL)
    def test_case_04(self):
        assert 1 < 0

    def test_case_05(self):
        assert 1
```

severity 的默认级别是 normal，所以上面的用例 5 可以不添加装饰器了。

```
allure.dynamic
@pytest.mark.parametrize('name', ['动态名称 1', '动态名称 2'])
def test_case_05(self, name):
    allure.dynamic.title(name)
```

4 Allure 报告集成

前面介绍了 Allure 的基本使用，接下来要把它集成到接口自动化的项目中来

Allure 基本配置

```
# ---------------- Allure 报告相关 ----------------------
# Allure 的 json 文件路径：
ALLURE_JSON_DIR_NAME = 'allure_json'  # Allure 的 json 文件夹命名：
ALLURE_JSON_DIR_PATH = os.path.join(REPORT_PATH, ALLURE_JSON_DIR_NAME)
# allure 的 html 报告路径
ALLURE_REPORT_DIR_NAME = 'allure_html'   # allure 的 html 文件夹命名：
ALLURE_REPORT_DIR_PATH = os.path.join(REPORT_PATH, ALLURE_REPORT_DIR_NAME)
# 压缩包路径：
ZIP_FILE_PATH = os.path.join(REPORT_PATH, 'report.zip')
# 生成报告命令：
ALLURE_COMMAND = "allure generate {} -o {}".format(ALLURE_JSON_DIR_PATH, ALLURE
_REPORT_DIR_PATH)
```

Allure 报告执行集成

在接口自动化项目中要用命令行模式调用并执行 Allure 生成测试报告

```
# 执行终端命令、subprocess 要代替一些老旧的模块命令：
from subprocess import Popen, call
from conf import settings
from util.LogHandler import logger
class AllureHandler(object):
    # 读取 json 文件，生成 allure 报告：
    def execute_command(self):
        # Python 执行终端命令：shell=True：将['allure', 'generate', '-o', 'xxxx']
替换为'allure generate -o'
        try:
            call(settings.ALLURE_COMMAND, shell=True)
            logger().info('执行 allure 命令成功')
        except Exception as e:
            logger().error("执行 allure 命令失败,详情参考: {}".format(e))
```

七、Jenkins 持续集成

在自定义好 Python 接口自动化测试框架，实现接口自动化后，还需要 GitLab 对 python 代码进行管理、版本控制、最后再通过 Jenkins 去手动构建触发、定时任务触发、代码上传触发接口自动化测试用例的执行，以达到持续集成的目的。

1 Jenkins 安装

Jenkins 是一个开源的软件项目，是基于 java 开发的一种持续集成工具，用于监控持续重复的工作，旨在提供一个开放易用的软件平台，使软件的持续集成变成可能。由于是基于 java 开发因此它也依赖 java 环境，安装之前需要先安装 jdk，建议 jdk1.8+，安装后配置 java 环境变量。安装 jdk 成功后，在 cmd 使用 java -version 可查看 jdk 版本信息。

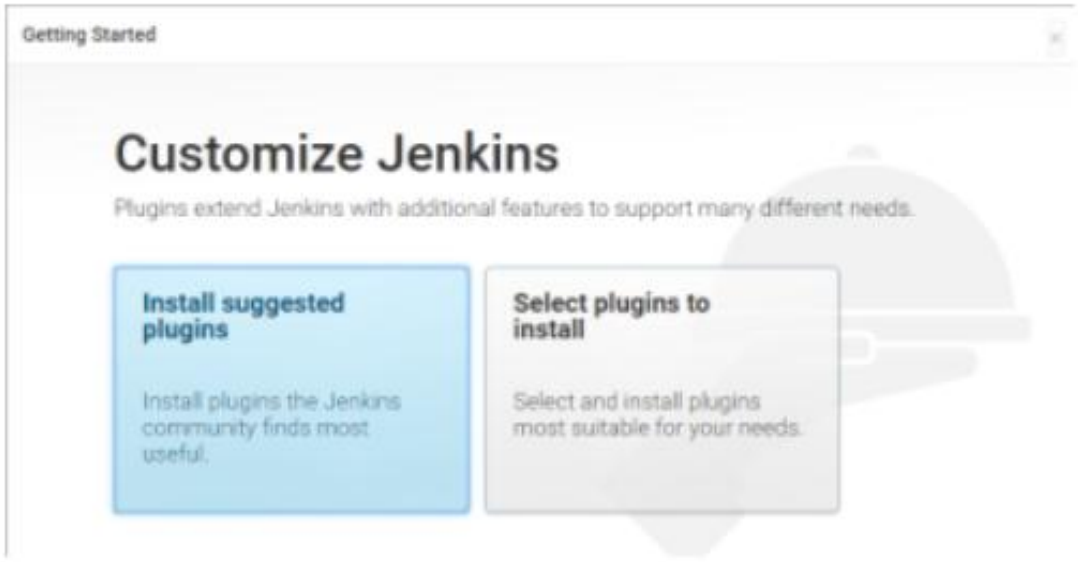
官网下载地址：https://www.jenkins.io/zh/ ，或者这个网址直接下载：http://mirrors.jenkins.io/war-stable/latest/jenkins.war ，下载的包可以是 Jenkins.war，也可以是 Jenkins.msi。

如果是 Jenkins.war，那么将这个文件放到一个目录下，打开 cmd，进入到此文件所在目录，执行：java -jar jenkins.war --httpPort=8080 即可；如果是 Jenkins.msi，直接双击打开运行安装。

安装好之后，打开浏览器输入：localhost:8080，会出现下面这个图：

稍等一会，会提示需要输入管理员密码，按照提示到对应的路径找到 initialAdminPassword 这个文件，用记事本打开，里面会有一个密码，把这个密码粘贴到文本框中点确定即可。

到下图这一步时，选择左边安装推荐的插件，安装插件的过程会比较漫长，如果网络比较差，很多安装失败的话也可以点击 continue 跳过安装，就会跳到设置用户页面，设置一个用户密码，就进入主页面了。



2 Jenkins 部署项目

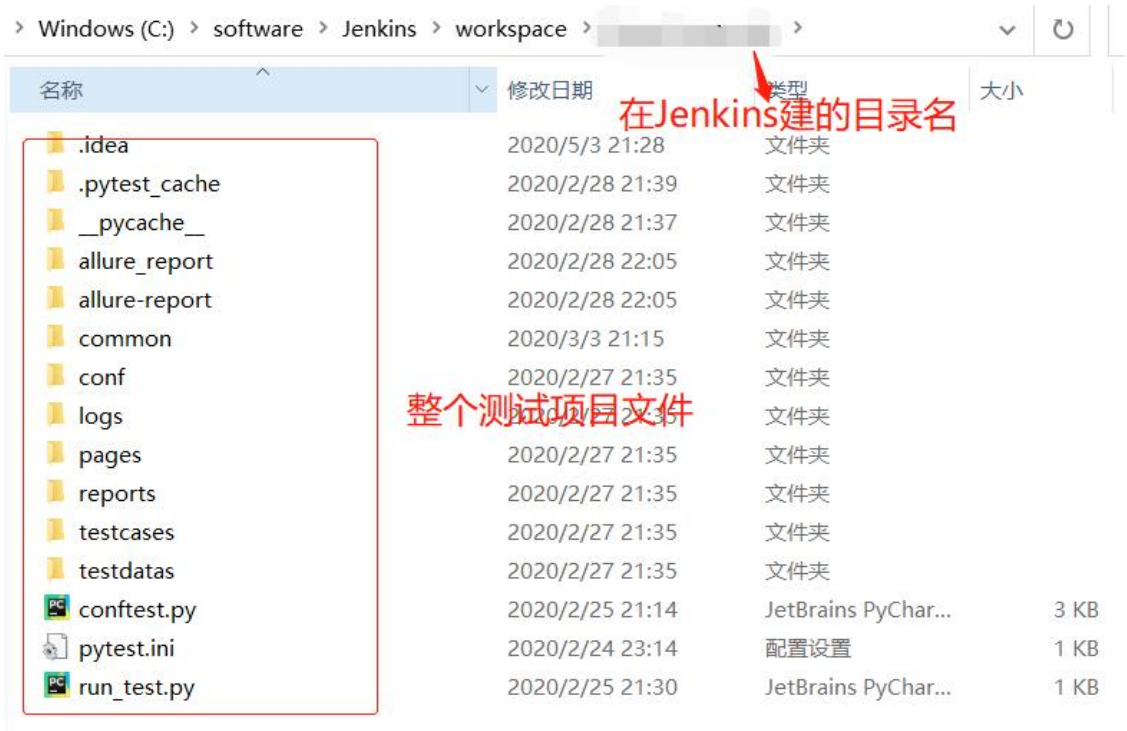
如果在没有 pycharm 编译器的情况下我们怎么运行项目呢？可以在 windows 的命令行下直接运行，但运行时需要先切换到项目路径下，然后在 cmd 输入 python 文件名.py，或者不切换路径，直接输入文件的完整路径，如：python c:\test\main.py。

当然，更方便的还是在 Jenkins 上面 - > 一键运行。

首先要创建一个工作项目，类型选第一个 Freestyle project 就行。

每一个项目建立后，都需要先执行一遍构建才会有工作区间，否则进入工作区间会显示如下：

构建一遍后再点进工作区间会显示“空目录”，这是因为我们还没有把测试项目放到 Jenkins。先找到 Jenkins 安装目录下的 workplace，然后在对 -> 应目录下把我们的测试项目整个拷过来即可，如下：



如果源码不在本地，使用的是 git 或 svn，那么配置好项目的 git/svn，执行构建后会自动

把服务器上的代码拉取下来。下面是源码管理及构建的步骤：

2.1 源码管理

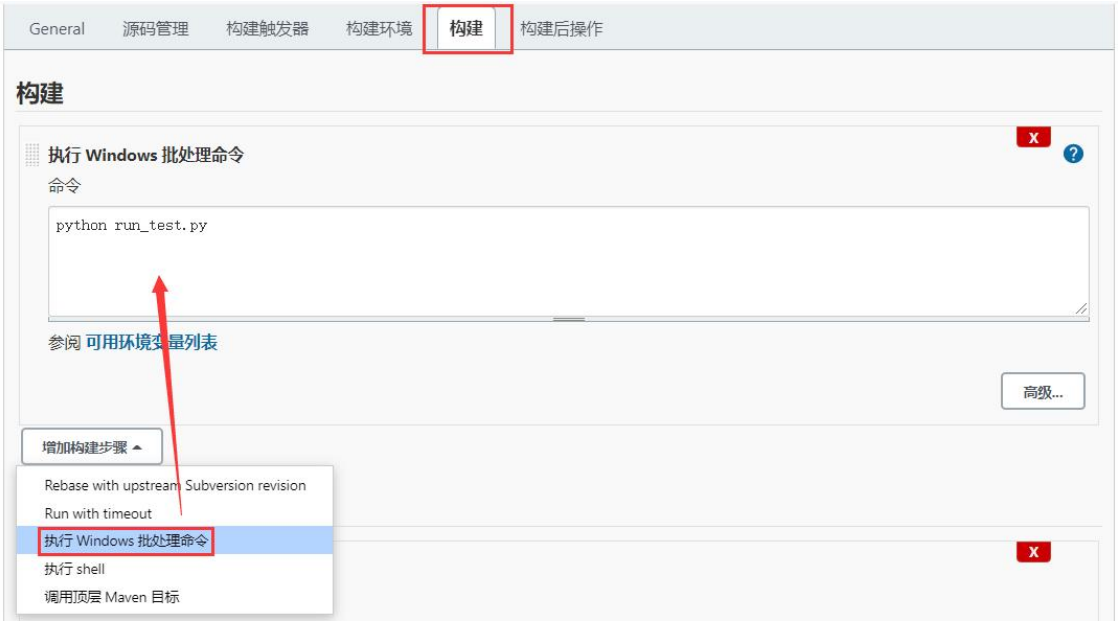
源码管理(source code management)：需要安装插件，如：git/svn，没有插件时显选择无，直接使用本地代码，需要拷贝到工作区间。

这里以 git 为例，参数说明：

- repository url：项目的 git 地址(svn 则填 svn 地址)
- credentails：点击添加 Jenkins 凭据，填写 git/svn 的账号及密码》保存；保存后在左侧选择账号

2.2 构建

构建，即如何运行项目。添加构建步骤，windows 选择 execute windows batch command，linux 或 mac 选择 execute shell，然后输入终端的执行命令:python 文件名.py，保存后点击构建(build now)就可以运行项目了，注意这里运行的文件需要存放在工作区间的根目录下（因为这里默认是根目录）。



配置好后，就可以运行项目了，在对应项目里点击 build now(立即构建)就可以运行项目了，在左下角可查看构建进度、构建历史，还可以查看控制台的输出。

2.3 构建触发器

实际有可能我们运行项目不是一定要定时去执行的，可能开发每次一发版我们就要执行构建一遍来观察开发的新代码是否会有问题，那我们就可以配置一个构建触发器。

配置位置：对应项目 - > 配置 - > 构建触发器 - > 其他工程构建后触发(Build after other projects are build)，输入对应的项目名称，选择“构建稳定时触发”，意思就是我的测试脚本项目在另一个项目（test 项目，这里指开发的项目）部署构建完成后没有出现问题了才触发构建我的测试脚本项目。

也就是说，每一次开发发版成功后就会构建一遍我的测试脚本项目。

定时触发器， 顾名思义，就是定时去执行项目。

配置位置：对应项目 -> 配置 -> 构建触发器 -> 定时构建（Build Periodically）。

定时任务规则分为五个部分规则：

第一个* ：表示分钟，取值 0~59
第二个* ：表示时间，取值 0,23
第三个* ：表示日期，取值 1~31
第四个* ： 表示月份，取值 1~12
第五个* ： 表示星期，取值 0~7，其中 0 和 7 代表的都是周日

每次输入定时规则时，下方会有提示说明，以及上一次运行时间、下一次运行的时间。

常见其他的规则：

- 中横线（-） ：表示指定范围，如，每周 1 到 5：1-5
- 斜划线（/） ：表示指定时间间隔 ,如，每隔分钟： H/15 ；
- 逗号（,） ：表示指定变量取值， 如每天的 8 点,12 点,18 点 ： 8,12,18

示例说明	分钟	小时	日期	月份	星期
	*	*	*	*	*
每隔 15 分钟构建一次	H/15	*	*	*	*
每隔 2 个小时执行一次	H	H/2	*	*	*
每月 1-15 号内每隔 3 天执行一次	H	*	1-15/3	*	*
每天中午 12 点定时构建一次	H	12	*	*	*
每天 8/18/22 点定时构建一次	H	8,18,22	*	*	*
工作日的上午 9-12 点每隔 2 个小时构建一次	H	9-12/2	*	*	1-5
每周 1,3,5 的晚上 11 点半执行一次	30	23	*	*	1,3,5

具体设置：

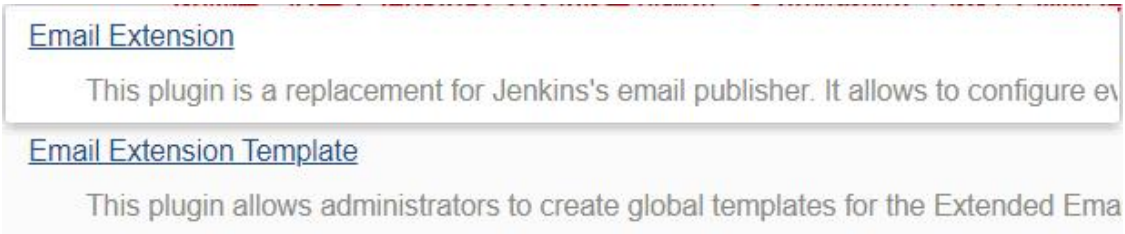
- 每隔 15 分钟构建一次 ： H/15 * * * *
- 每隔 2 小时构建一次 ： H H/2 * * *
- 每天中午 12 点定时构建一次 ： H 12 * * *
- 每天 8/18/22 点定时构建一次 ： H 8,18,22 * * *
- 工作日的上午 9-12 点每隔 2 个小时构建一次： H 9-12/2 * * 1-5
- 在每个小时的前半个小时内的每 10 分钟 ： H(0-29)/10 * * * *
- 每两小时 45 分钟，从上午 9:45 开始，每天下午 3:45 结束 :45 9-16/2 * * 1-5
- 每个工作日上午 9 点到下午 5 点，每隔 2 小时构建一次: H H(9-16)/2 * * 1-5

3 Jenkins 发送邮件

接下来介绍一下如何使用 jenkins 来发送邮件

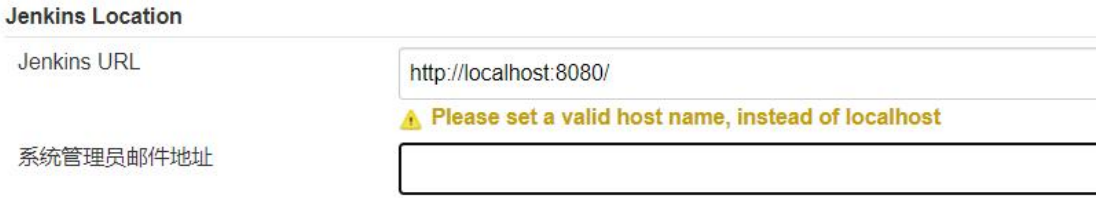
3.1 安装插件

Jenkins 需要先安装以下两个插件



3.2 配置邮件地址

位置：jenkins 管理 -> 系统配置 -> Jenkins Location。在系统管理员邮件地址，输入对应的邮件地址



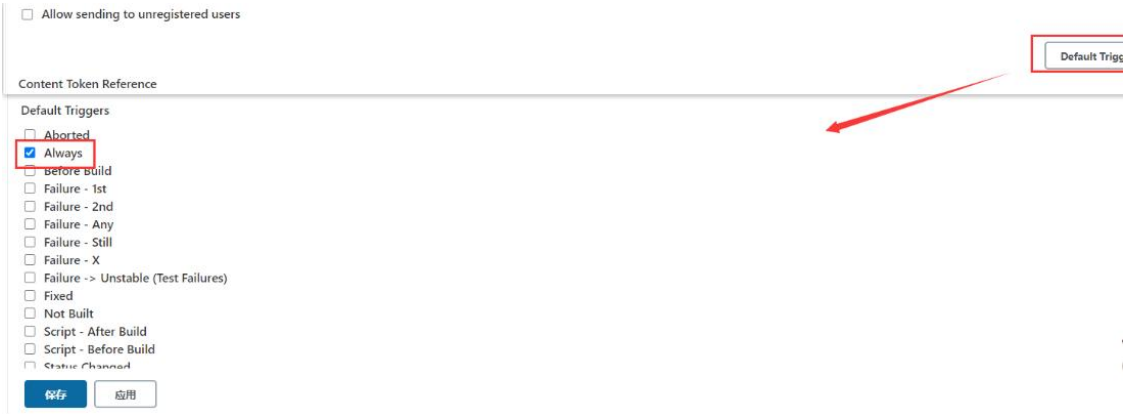
3.3 配置 smtp 服务

位置：jenkins 管理 -> 系统配置 -> Extended E-mail Notification，填写对应的 smtp 服务器相关内容，如下是 qq 邮箱示例

点击高级，输入对应的邮箱地址和 smtp 登录的授权码，勾选 ssl。

3.4 配置邮件触发器

位置：jenkins 管理 -> 系统配置 -> Default Triggers，默认是勾选失败才发，可以配置总是发送（或根据需求勾选），那么每次运行项目时都会发送邮件。



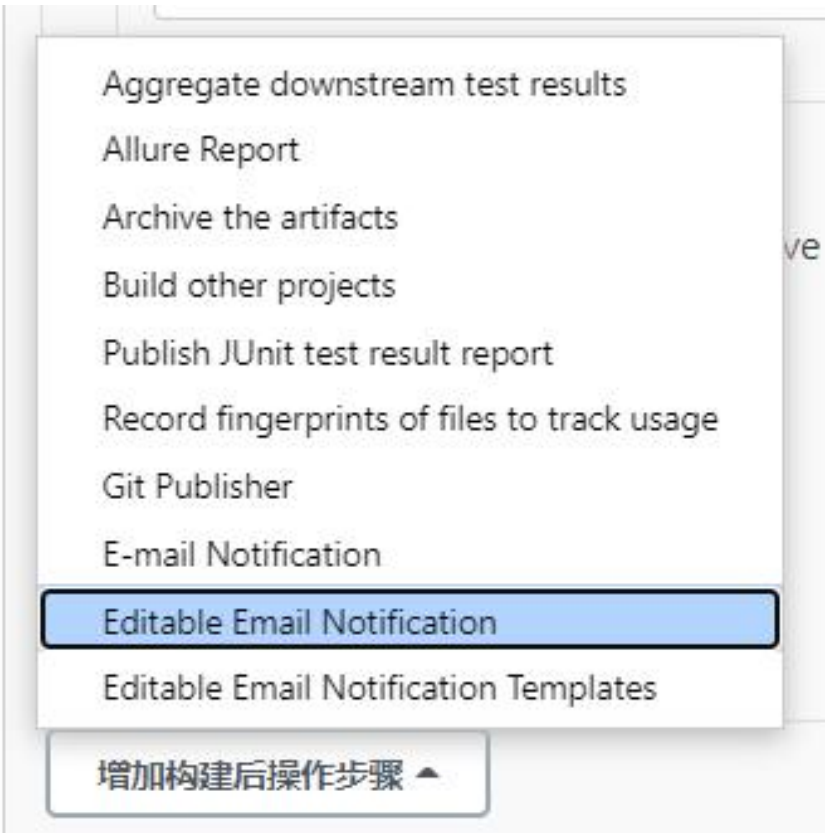
需要特别注意的是，如果项目已经创建了（项目默认是失败发送）才去配系统触发器（配了总是发送），那么项目会读取项目的触发器，如果是先配置了系统触发器再创建的项目，那么项目的触发器是读取系统触发器的。配置项目的触发器见第 5 点。

3.5 配置构建后操作

位置：对应项目 -> 配置 -> 添加构建后操作-Editable Email Notifiction，输入收件人邮件地址(Project Recipient List)，多个邮箱可以用英文逗号隔开；输入附件内容

(Attachments)：reports/report.html，这里输入的是：项目存放测试报告的文件夹名/测试报告文件名。

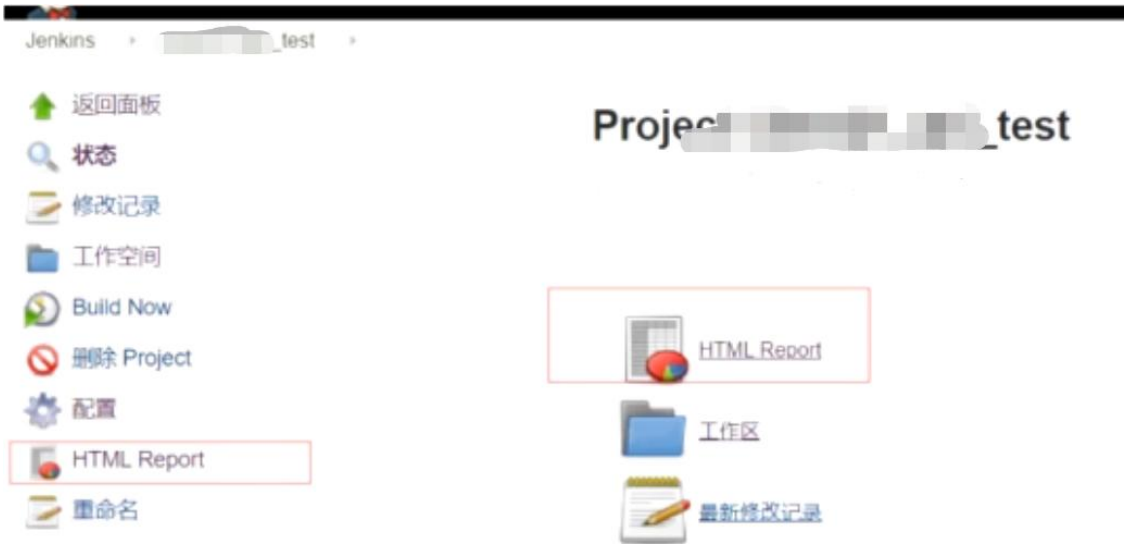
邮件触发器，点击高级 - > 找到 triggers。



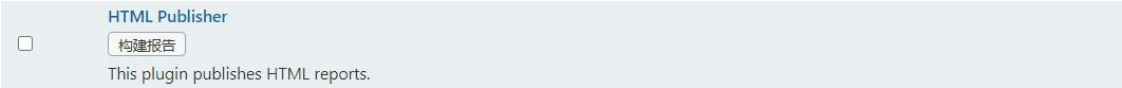
邮件触发器：

3.6 HTML 展示配置

配置后项目生成的 html 报告就可以直接在项目中查看，如下图：



需要安装插件：HTML Publisher



安装后配置项目构建后操作：对应 - > 项目 - > 配置 - > 增加构建后操作。

配置好后重新构建项目就会在项目生成 HTML Report 目录，但是查看这个报告后你会发现 html 报告在 jenkins 上显示没有那么美观，那是因为少了 html 里面的 css 和 js，因为 jenkins 是默认会禁掉 css 和 js。

解决办法：

在 jenkins 管理 -> 命令行终端（Script Console），输入：System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "script-src 'unsafe-inline'")，点击【运行】就好了（记住重启 jenkins 后，就会恢复默认设置，需要重新配置）。

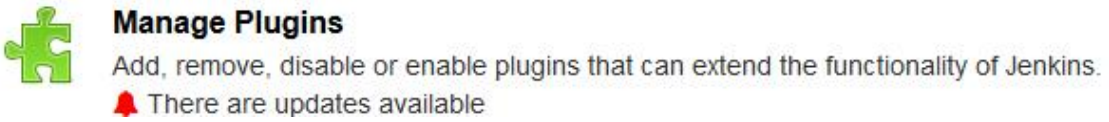
4 Jenkins 与 Allure 集成

4.1 Allure 插件安装

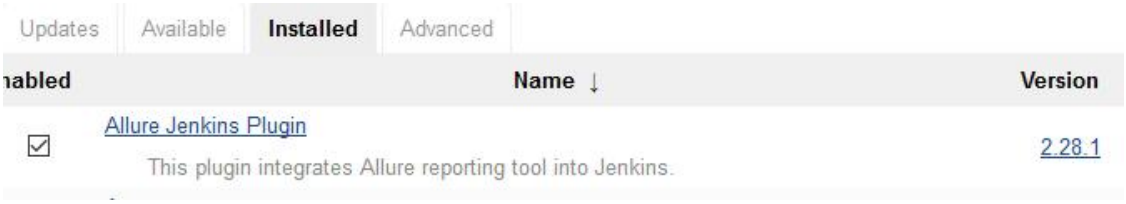
Jenkins 集成 allure 测试报告，需要安装 Allure 插件作为支持

1.打开 Jenkins，首页点击[Manage Jenkins]

2.选择[Manage Plugins]



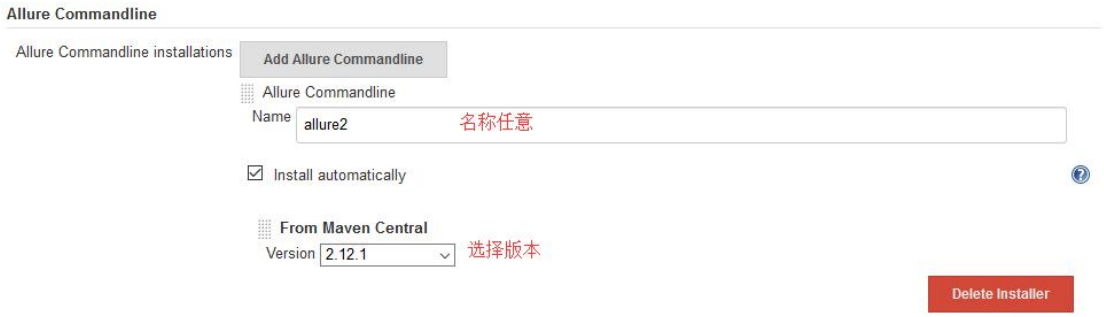
3.选择[Available]选项，搜索输入框输入 Allure，搜索出来的名字就叫 Allure，当安装后名字会变为 Allure Jenkins Plugin



4.2 Allure Commandline 配置

1.jenkins 配置页，选择全局工具配置[Global Tool Configuration]

2.全局宫配置页下拉到最后一项，你会看到“Allure Commandline”项， 按下图配置之后保存即可



4.3 Job 项目配置

- **新建 Job**

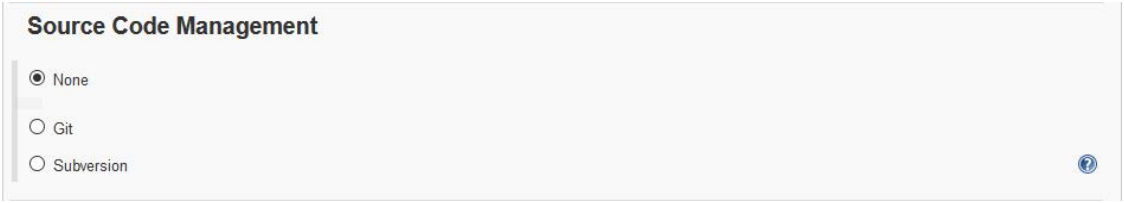
Jenkins 首页点击新建 Job[New Item]

输入 job 名称，选择自由风格的项目，点击[ok]

- **General 配置**

输入描述信息及其他信息，因为我的项目在我的本地，所以我这里未配置其他选项，你可以根据实际情况进行设置

- **源码管理(Source Code Management)**



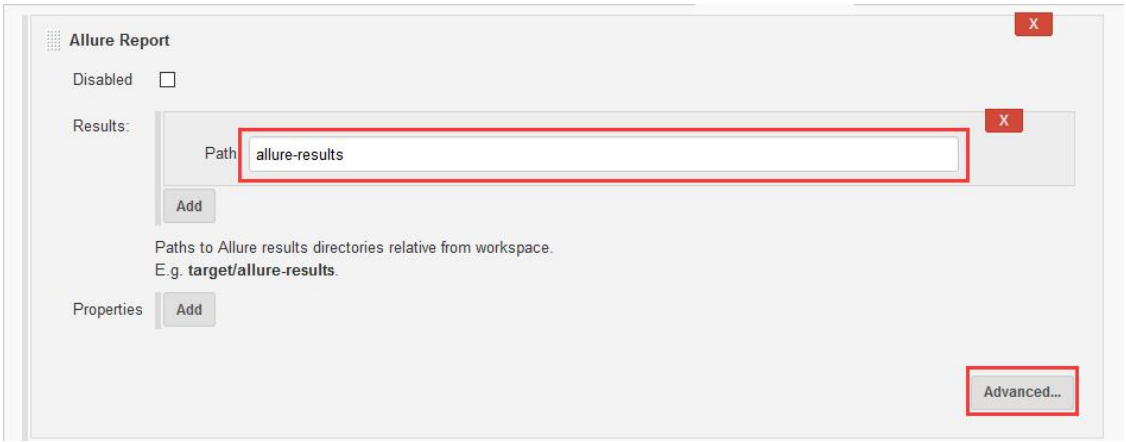
- **构建(Build)**

输入如下命令，我的的测试项目使用的是虚拟环境

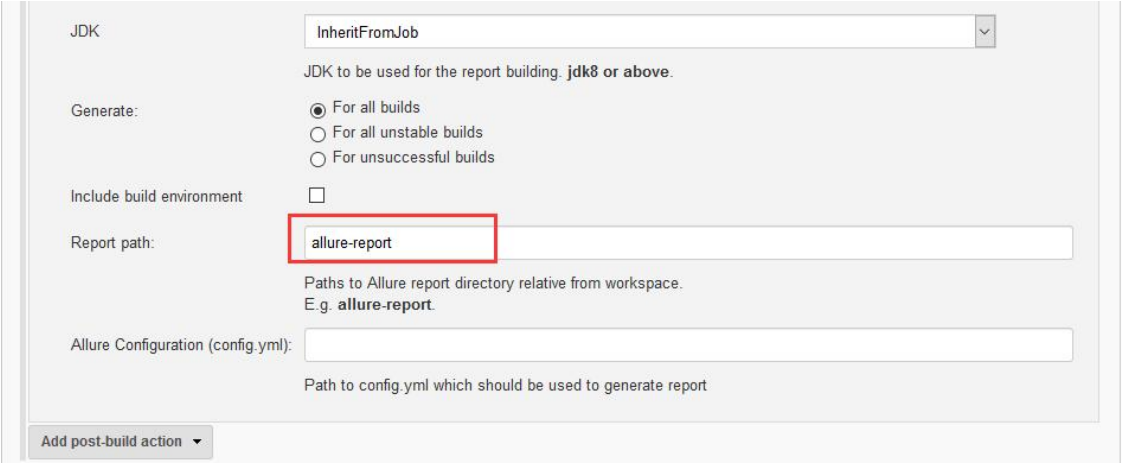

```
cd ./venv/Scripts
activate && cd ..&& cd .. && python.exe run.py
```

- **构建后操作(Post-build Actions)**

输入 Path 名称，这里的 path 名称表示在项目跟目录下生成此文件夹，文件夹用来保存生成 html 报告之前所依赖的 json，xml，txt 等类型文件



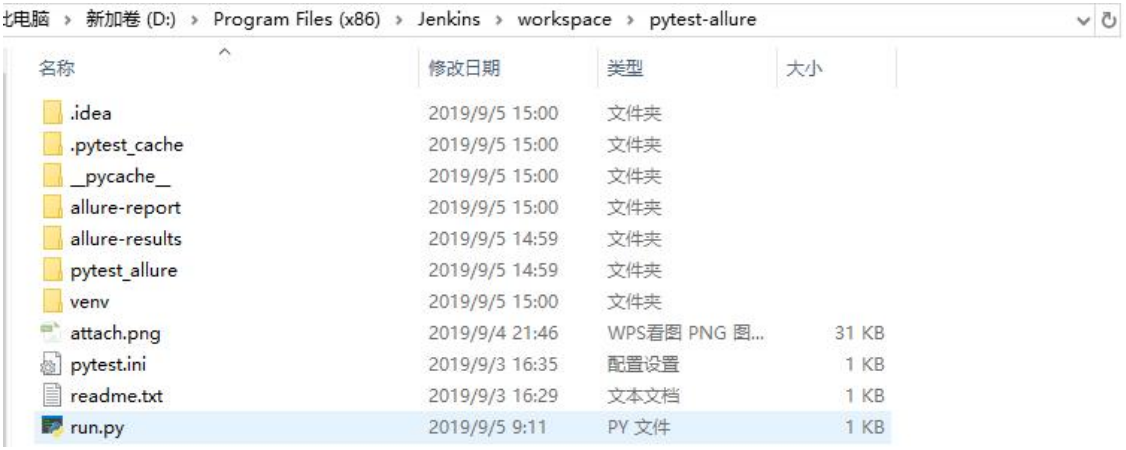
点击[Advanced]，输入 Report path 名称，这里的 path 同样会在项目的跟目录下生成，同时保存最后的 html 报告



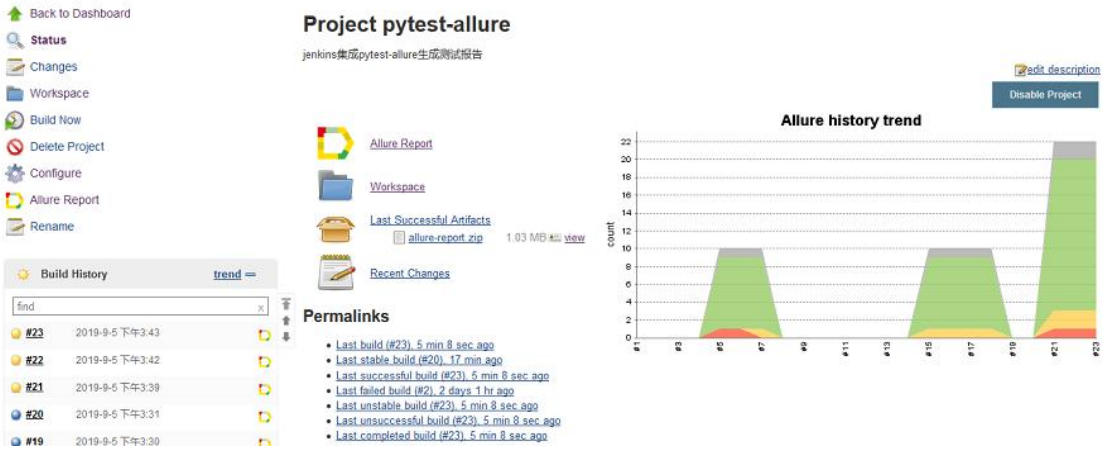
以上所有信息配置完成后保存即可

- **执行构建**

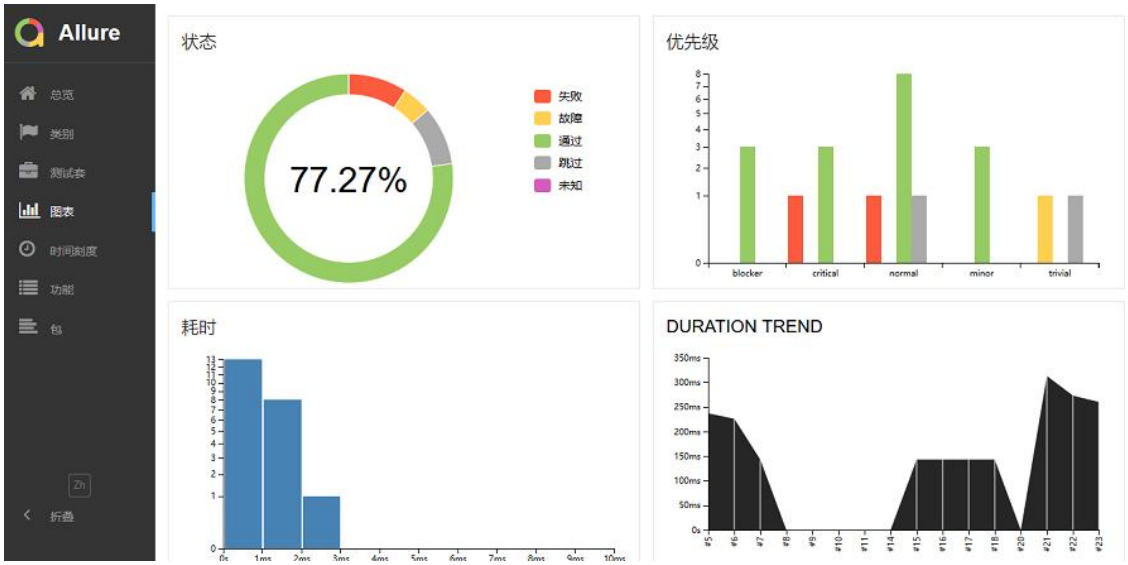
在 jenkins 的工作空间新建名称为 pytest-allure 的目录，并把项目拷贝到此目录(这个工作空间在 General 中可以自定义的，因为我没配置所以使用的默认的工作空间)



接下来点击[Build Now]执行构建



- **查看报告**



至此，我们基于 Python Requests + Pytest + DDT + YAML + Allure + logging + Jenkins + Git +邮件发送 等的自动化持续集成框架基本实现完成。当然还有很多值得优化的地方需要进一步完善，这个可以根据项目的需要进行优化调整。