Python+Requests+Unittest接口测试框架

1 Requests从入门到精通

1.1 Requests介绍

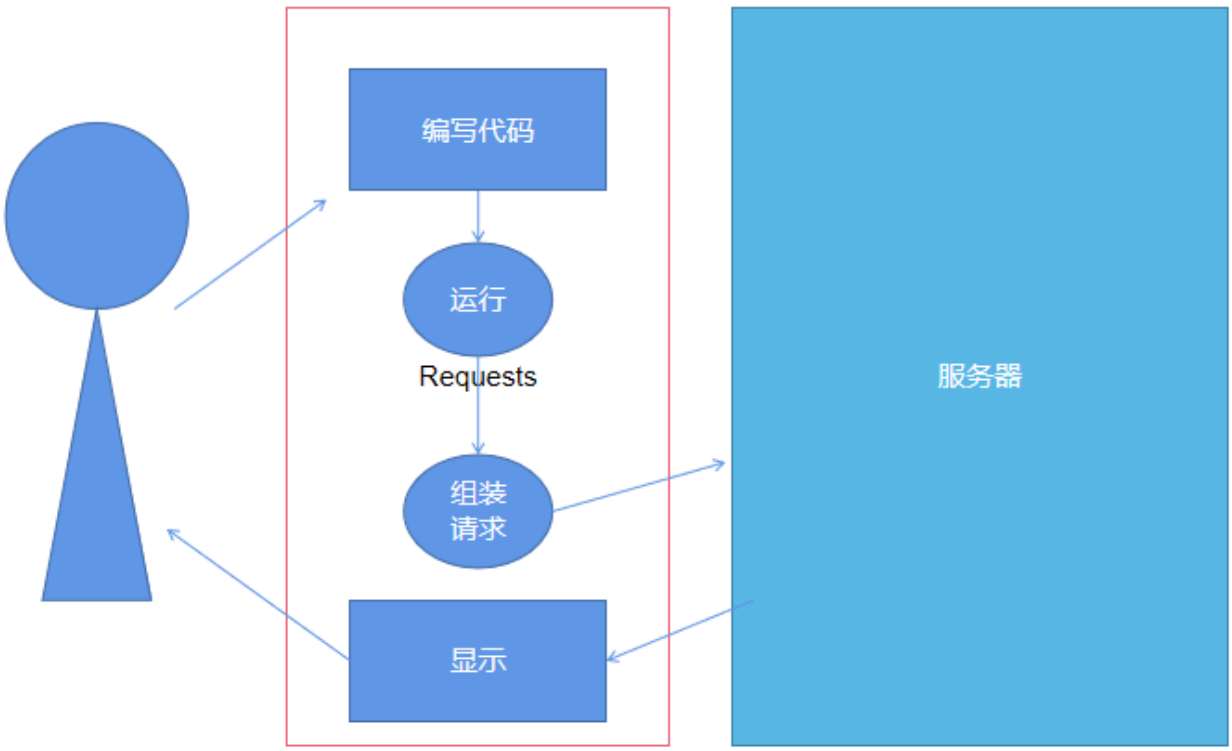
Python能够进行接口测试的库包括：urlib,http,request等

Requests在此基础上进行了封装，优化了其他模块难以学习使用的特点，简单易学并能完全满足互联网接口测试的需求。

1.2 Requests安装

```
pip install requests
```

用户通过Requests模块与服务器进行交互作用说明图



1.3 Requests语法

一个HTTP接口组成部分包括：

- 请求方法

- URL
- 请求头
- 请求体
- 响应数据

Requests可以在内部封装的函数中设置HTTP接口所需要的请求数据：

```
import requests

response = requests.post(url="http://www.xxx.com/api/v1/xx", params={"key":"value"},
data={"key":"val"}, json={"key":"val"}, headers={"Content-Type":"application/json"},
cookies=cookie)
```

- response 接口requests.post(...)方法返回的Response对象，该对象包括了服务器返回的响应数据
- requests.post(...) 设置请求方法为POST，其他设置方法有：requests.get(),requests.put(),requests.delete()等等
- url="http://www.xxx.com/api/v1/xx" 设置接口的URL是http://www.xxx.com/api/v1/xx
- params={"key":"value"} 设置接口的查询参数为key=value
- data={"key":"val"} 设置表单格式请求体数据为key=value
- json={"key":"val"} 设置json结构请求体数据为{"key":"val"}
- headers={"Content-Type":"application/json"} 设置请求头为{"Content-Type":"application/json"}
- cookies=cookie 设置cookies，在这里右边的cookie是一个变量

1.4 Requests入门案例和处理响应数据

百度搜索案例：

```
import requests

# 向服务器发送【百度搜索接口】请求
response = requests.get(url="http://www.baidu.com/s?wd=python")
# 打印结果
print(response.text)
```

处理响应数据

- 打印请求的URL：response.url
- 打印响应状态码：response.status_code
- 打印Cookie：response.cookies
- 打印响应头：response.headers
- 打印响应正文（响应体）
  - 字节码打印:response.content
  - 文本打印:response.text
  - Json打印:response.json()

1.5 Requests实战案例之Cookie运用技巧

使用Cookie完成拉勾商城后台管理系统登录

```
import requests

# 获取验证码
response = requests.get(url="http://localhost/index.php?m=Home&c=User&a=verify")
# 提取cookie
cookie = response.cookies
# 登录
response = requests.post(url="http://localhost/index.php?m=Home&c=User&a=do_login",
    data="username=13800138006&password=123456&verify_code=8888",
    headers={"Content-Type":"application/x-www-form-urlencoded"})
# 打印结果
print(response.text)
```

第一步：查看不使用Cookie时的登录

第二步：引入Cookie对比效果

1.6 Requests实战案例之Session运用技巧

requests模块封装了session模块，该模块可以自动管理session会话信息，自动保存cookie到连接中

这样就不需要主动传递cookie了

使用Session完成拉勾商城后台管理系统登录

语法：

```
第一步： 实例化session对象
session = requests.Session()
第二步： 使用session发送接口请求，后续一旦都是同一个session对象发送的请求，那么这个session都会把
所有请求过程产生的会话信息，都保存到session对象当中。我们就不需要主动的管理会话信息了。
特别是Cookie就不需要管理了。
```

```
import requests

# 实例化session
session = requests.Sessino()
# 获取验证码
response = session.get(url="http://localhost/index.php?m=Home&c=User&a=verify")

# 登录
```

```
response = session.post(url="http://localhost/index.php?m=Home&c=User&a=do_login",
    data="username=13800138006&password=123456&verify_code=8888",
    headers={"Content-Type":"application/x-www-form-urlencoded"})
# 打印结果
print(response.text)
```

1.7 Requests实战案例之上传文件

上传文件语法

```
data = None
files = {"file":open(path, 'rb')}
requests.post(url="http://xxx.com/",data=None, files=files)

其中files就是要上传的文件
```

案例：拉勾商城上传图片

```
# 需求：上传文件（上传文件接口是什么）
# 导包
import requests
# 实例化session对象
session = requests.Session()
# 使用session对象发送验证码接口请求(session发送请求的方式和requests一模一样，区别就是我们是使用
session对象发送请求)
response = session.get(url="http://localhost/index.php?
m=Home&c=User&a=verify&r=0.9698092918823165")
print(response.content)
print(response.cookies)
# 使用session对象发送登录接口请求
response = session.post(url="http://localhost/index.php?
m=Home&c=User&a=do_login&t=0.8916582864091132",
                        data={"username":"13800138006", "password":"123456",
"verify_code":"8888"},
                        headers={"Content-Type":"application/x-www-form-urlencoded"})
# 打印登录的结果
print(response.json())

# 上传图片
data = {"id":"WU_FILE_0", "name":"camara.jpg", "type":"image/jpeg",
        "lastModifiedDate":"Mon May 31 2021 15:05:08 GMT+0800 (中国标准时间)",
        "size":"169267"}

# 要上传的文件
file = {"file": open(r"C:\Users\wind\Pictures\first_requests.jpg", 'rb')}

#调用上传图片接口
```

```
response =
session.post(url="http://localhost/index.php/home/Uploadify/imageUp/savepath/head_pic/pic
title/banner/dir/images.html",
                         data=data,
                         files=file)

print("上传文件的结果为：", response.json())
```

上传文件接口测试方法：

- 测试是否上传成功
- 安全相关测试：
    - 测试上传可执行代码
    - 测试上传其他需求规定以外的文件

2 Requests+Unittest实现接口测试

2.1 引入Unittest的目的

Requests模块主要是用来构造接口请求和接收响应数据。而进行接口测试时，还需要管理接口、断言、

参数化、生成测试报告等等。

- 管理接口测试用例
- 断言
- 生成测试报告

2.2 Requests + Unittest实现拉勾商城登录接口测试

拉勾商城登录测试点

使用代码实现接口测试点：

```
import unittest, requests


class TestLgShopLogin(unittest.TestCase):

    def setUp(self) -> None:
        self.session = requests.Session()

    def test01_login_success(self):
        # 使用session对象发送验证码接口请求(session发送请求的方式和requests一模一样，区别就是我们是使用session对象发送请求)
        response = self.session.get(url="http://localhost/index.php?m=Home&c=User&a=verify&r=0.9698092918823165")
        print(response.content)
        print(response.cookies)
        # 使用session对象发送登录接口请求
        response = self.session.post(url="http://localhost/index.php?m=Home&c=User&a=do_login&t=0.8916582864091132",
                                     data={"username": "13800138006", "password": "123456", "verify_code": "8888"},
                                     headers={"Content-Type": "application/x-www-form-urlencoded"})
        # 打印登录的结果
        print(response.json())
```

3 Python操作数据库

进行接口测试时，我们需要连接到数据库中，对数据源进行备份、还原、验证等操作。

3.1 Python连接数据库常见模块

- MysqlDB

  python2时代最火的驱动库。基于C开发，对windows平台不友好。

  现在已经进入python3时代，基本不再使用

- MysqlClient

  mysqldb的衍生版本，完全兼容python3.

  它是重量级Web开发框架Django中ORM功能依赖工具

- Pymysql

  纯Python实现的驱动，性能比MysqlDb差，但是安装简单，容易使用。

- SQLAlchemy

  即支持原生SQL也支持ORM的库。

3.2 精通Pymysql使用方法

安装方法：pip install pymysql

3.2.1 Pymsyql使用流程

- 获取连接
- 获取游标
- 执行SQL语句
- 关闭游标
- 关闭连接

3.2.2 Pymysql入门示例

案例：查询Mysql的版本号

```
建立连接之后，查询mysql的版本是：select version();
```

```
建立连接的语法：
# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")

conn:我们定义的变量，接收Pymysql.connect方法返回的对象（建立的连接对象）
pymysql.connect:建立的连接的方法
 host="192.168.85.139" ：要连接的数据库的服务器域名
```

```
port=3306：要连接的数据库的端口号
database="ssm_lagou_edu"：连接的数据库库名
user="root"：用户名
password="Lagou@1234"：密码
charset="utf8"：建立连接的编码
```

```
import pymysql

# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 建立游标
cursor = conn.cursor()
# 执行核心SQL语句
cursor.execute("select version();")
# 关闭游标
cursor.close()
# 关闭连接
conn.close()
```

3.2.3 Pymysql实现数据库CRUD

CRUD(增删改查)：Created、Retrieve、Update、Delete

查询

案例：查询拉勾课程表中id为27的数据

```
import pymysql

# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 建立游标
cursor = conn.cursor()
# 执行核心SQL语句
cursor.execute("select id,course_name from course where id=27")
# 打印一条记录
print("只打印一条记录：", cursor.fetchone());
```

```
print("打印全部记录：", cursor.fetchall());
print("打印执行SQL后影响的行数：", cursor.rownumber)
# 重置游标指向的数据位置
cursor.rownumber=0
print("重置游标指向的数据位置再次打印全部记录：", cursor.fetchall());


# 关闭游标
cursor.close()
# 关闭连接
conn.close()
```

插入：插入一本名为【测试开发从入门到精通】的课程

插入、修改、删除数据时，需要提交事务

```
conn.commit() #提交事务
```

```
import pymysql

# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 建立游标
cursor = conn.cursor()
# 执行核心SQL语句
cursor.execute("INSERT INTO `ssm_lagou_edu`.`course`(`id`, `course_name`, `brief`,
`price`, `price_tag`, `discounts`, `discounts_tag`, `course_description_mark_down`,
`course_description`, `course_img_url`, `is_new`, `is_new_des`, `last_operator_id`,
`auto_online_time`, `create_time`, `update_time`, `is_del`, `total_duration`,
`course_list_img`, `status`, `sort_num`, `preview_first_field`, `preview_second_field`,
`sales`) VALUES (7, '文案高手的18项修炼', '手把手教你写出实用的高转化文案', 263.00, '', 100.00,
'成就自己', '', 'https://edu-lagou.oss-cn-
beijing.aliyuncs.com/images/2020/07/10/15943482627237468.jpg', NULL, NULL, NULL, NULL,
'2020-07-10 10:33:56', '2020-08-04 17:34:58', 0, NULL, 'https://edu-lagou.oss-cn-
beijing.aliyuncs.com/images/2020/07/10/1594348262748358.jpg', 1, 1, '100讲', '50课时',
1314);
")

# 执行后，使用navicat连接数据库查看插入结果


# 关闭游标
cursor.close()
# 关闭连接
```

```
conn.close()
```

修改：修改之前插入的【测试开发从入门到精通】的课修改为【测试开发技术巅峰之路】

```
import pymysql

# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 建立游标
cursor = conn.cursor()
# 执行核心SQL语句
cursor.execute("update course set course_name='学透pymysql' where id = 28")
# 执行后，使用navicat连接数据库查看修改结果


# 关闭游标
cursor.close()
# 关闭连接
conn.close()
```

删除：删除刚才添加或修改的课程

```
import pymysql

# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 建立游标
cursor = conn.cursor()
# 执行核心SQL语句
cursor.execute("delete from course where id=28")
# 执行后，使用navicat连接数据库查看删除结果

# 关闭游标
cursor.close()
# 关闭连接
conn.close()
```

3.3 Pymysql对数据库事务的操作

概念：对一组操作序列进行控制，这组操作序列要么全部都成功，要么全部都失败。

例如：银行转账就需要通过数据库事务技术来实现。

在接口测试中，任何接口请求都是一个事务，所以也有人称接口请求数为事务请求数。

数据库事务四大特性(ACID):

- 原子性

  事务是最小单位，不可以被分割。

- 隔离性

  事务与事务之间，事务的中间状态不被其他事务所见。

- 一致性

  数据事务提交前后，数据需要完全一样。

- 永久性

  事务一旦提交，那么数据就会永久的保存下来，不会因为物理因素而遭到破坏。

3.3.1 Pymysql操作事务的方法

- 自动操作方法

  建立连接时，通过关键字参数autocommit=False设置自动提交事务的开关为True/False

  建立连接后，通过conn.autocommit(False)设置自动提交事务的开关为True/False

- 手动操作

  不开启自动提交事务的前提下，才能手动操作

  提交事务：conn.commit()

  回滚事务：conn.rollback()

代码案例演示：拉勾教育插入课程操作

```
# 1 导包
import pymysql
# 2 建立连接
# 获取连接
conn = pymysql.connect(
    host="192.168.85.139",
    port=3306,database="ssm_lagou_edu",
    user="root",
    password="Lagou@1234",
    charset="utf8")
# 3 获取游标
cursor = conn.cursor()
```

```
try:
    # 4 执行SQL语句
    cursor.execute("") # 插入Python编程
    cursor.execute("") # 插入Java高薪

    # 提交事务
    conn.commit()
except Exception as e:
    # 回滚
    conn.rollback()

finally:
    # 5 关闭游标
    cursor.close()
    # 6 关闭连接
    conn.close()
```

3.4 封装Pymysql的CRUD操作

Pymysql使用起来比较繁琐，我们可以对它进行优化。

实现：

- 分离数据库配置文件
- 简化代码
- 自动关闭游标

示例代码

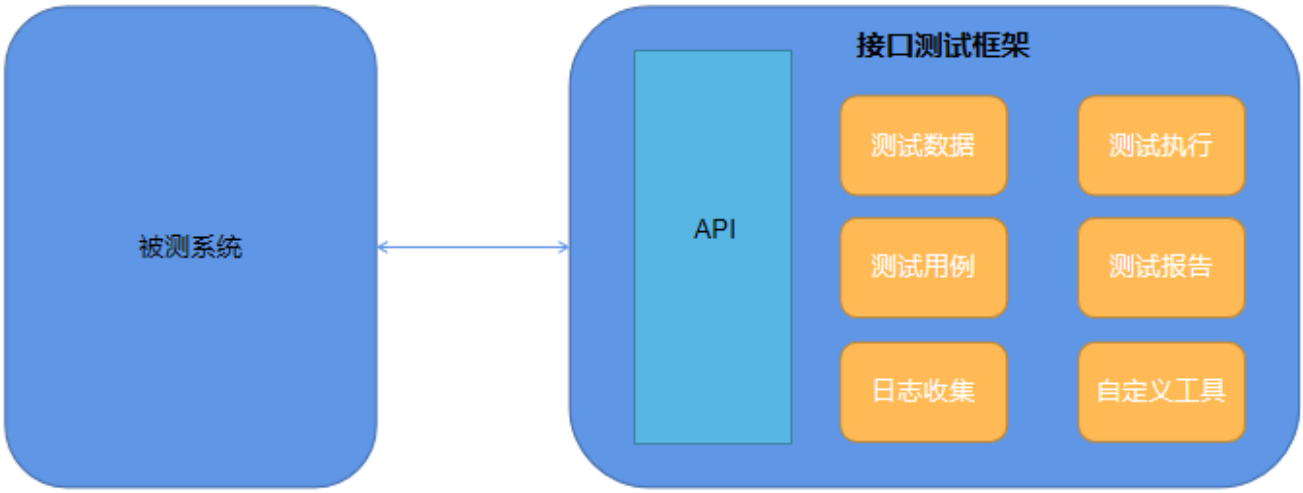
```
import pymysql

class DataBaseHandle(object):
    def __init__(self,host,username,password,database,port):
        '''初始化数据库信息并创建数据库连接'''
        # 下面的赋值其实可以省略，connect 时 直接使用形参即可
        self.host = host
        self.username = username
        self.password = password
        self.database = database
        self.port = port
        self.db =
pymysql.connect(self.host,self.username,self.password,self.database,self.port,charset='utf
8')
    def insertDB(self,sql):
        ''' 插入数据库操作 '''
        self.cursor = self.db.cursor()
        try:
            # 执行sql
            self.cursor.execute(sql)
            # tt = self.cursor.execute(sql)  # 返回 插入数据 条数 可以根据 返回值 判定处理结果
```

```
            # print(tt)
            self.db.commit()
        except:
            # 发生错误时回滚
            self.db.rollback()
        finally:
            self.cursor.close()

    def deleteDB(self,sql):
        ''' 操作数据库数据删除 '''
        self.cursor = self.db.cursor()
        try:
            # 执行sql
            self.cursor.execute(sql)
            # tt = self.cursor.execute(sql) # 返回 删除数据 条数 可以根据 返回值 判定处理结果
            # print(tt)
            self.db.commit()
        except:
            # 发生错误时回滚
            self.db.rollback()
        finally:
            self.cursor.close()

    def updateDb(self,sql):
        ''' 更新数据库操作 '''
        self.cursor = self.db.cursor()
        try:
            # 执行sql
            self.cursor.execute(sql)
            # tt = self.cursor.execute(sql) # 返回 更新数据 条数 可以根据 返回值 判定处理结果
            # print(tt)
            self.db.commit()
        except:
            # 发生错误时回滚
            self.db.rollback()
        finally:
            self.cursor.close()

    def selectDb(self,sql):
        ''' 数据库查询 '''
        self.cursor = self.db.cursor()
        try:
            self.cursor.execute(sql) # 返回 查询数据 条数 可以根据 返回值 判定处理结果
            data = self.cursor.fetchall() # 返回所有记录列表
            # 结果遍历
            for row in data:
                sid = row[0]
                name = row[1]
                # 遍历打印结果
                print('sid = %s,  name = %s'%(sid,name))
        except:
            print('Error: unable to fecth data')
        finally:
```

```
            self.cursor.close()

    def closeDb(self):
        ''' 数据库连接关闭 '''
        self.db.close()
```

4 通过Python+Requests+Unittest接口测试框架实现接口测试

4.1 接口测试框架设计思想



被测系统：要测试的系统

API：封装被测系统的接口

测试数据：分离测试用例和数据

测试用例：调用封装的API接口，实现接口测试用例的执行

日志收集：采集接口测试框架过程中产生的日志

测试执行：设置执行策略，采用多线程运行测试用例

测试报告：生成HTML测试保安高

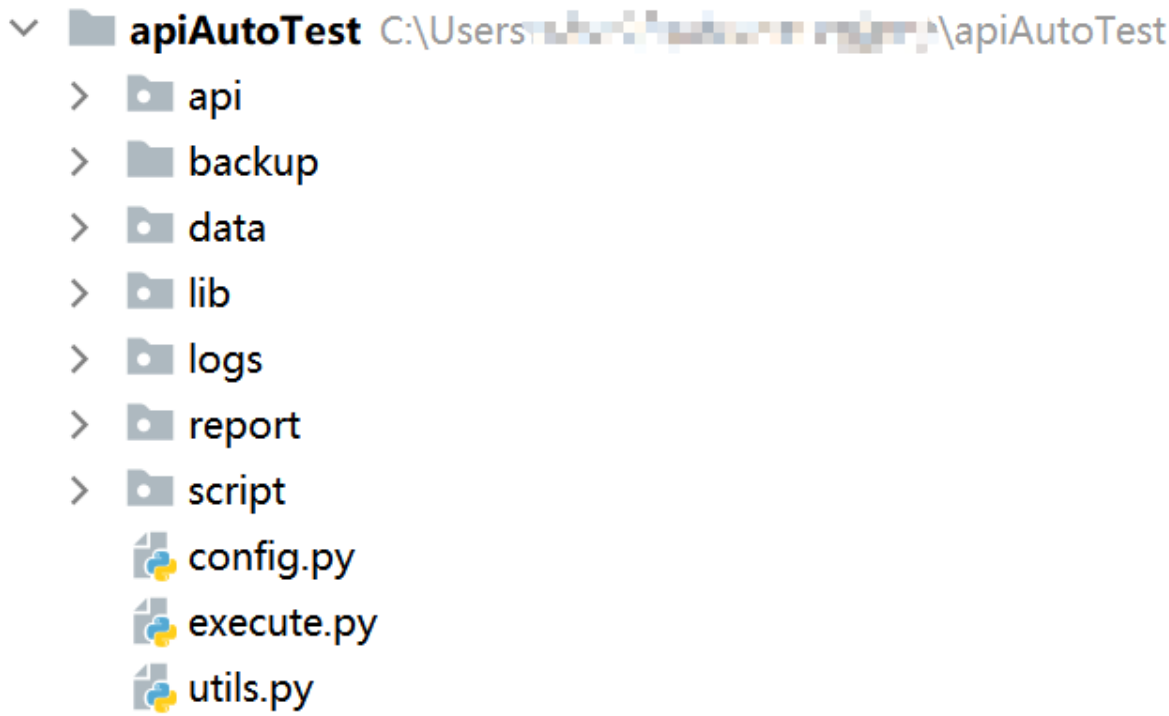
自定义工具：扩展更多功能

使用接口测试框架的优点：

- 把接口测试模式化，统一化
- 减少代码重复
- 增强维护性
- 提高测试效率

- 测试过程可溯

4.2 接口测试框架目录结构



- API：编写封装被测系统API接口的模块
- backup：存放要备份/还原的数据服务器数据
- data：存放要设计的接口测试数据
- lib：第三方工具包
- logs：存放生成的日志
- report：存放测试报告
- script：存放编写的接口测试用例脚本
- config.py：公共配置py文件，存放配置信息
- execute.py：项目入口文件，执行execute.py可以运行整个接口测试框架的全部代码，运行已经写在框架中的所有测试用例。
- utils.py：自定义工具的模块

4.3 接口测试框架接口封装

4.3.1 接口封装思想

接口的本质就是输入和输出，而接口的封装，主要是对输入数据进行封装，

输入数据中有**固定**的数据和**动态**的数据，其中**动态**变化的数据是我们重点测试的对象。

所以把固定的数据，封装在API模块中，动态变化的数据用形式参数从外界接收。

4.3.2 拉勾商城登录接口封装

拉勾商城登录信息：

请求方法：POST

URL：http://localhost/index.php?m=Home&c=User&a=do_login

HEADERS：

```
Content-Type:application/x-www-form-urlencoded
Cookie: PHPSESSID=vpi4htcq7ghhdt7jv3ilf74qq7;
```

Body：

```
username=13800138006&password=1234561&verify_code=8888
```

Response:


```
{"status":1,"msg":"\u767b\u9646\u6210\u529f","result":
{"user_id":8,"email":"","password":"519475228fe35ad067744465c42a19b2","paypwd":"519475228f
e35ad067744465c42a19b2","sex":0,"birthday":0,"user_money":"100000.00","frozen_money":"0.00
","distribut_money":"0.00","underling_number":0,"pay_points":100000,"address_id":0,"reg_ti
me":1523857661,"last_login":1622109286,"last_ip":"","qq":"","mobile":"13800138006","mobile
_validated":1,"oauth":"","openid":null,"unionid":null,"head_pic":"http:\/\/thirdwx.qlogo.c
n\/mmopen\/vi_32\/c58Iiaib1aPodvKHMMGR9ZYmq7XGFUgppvhxgQKrJxdlZTAauZ8dTucEguiamsncVDR3h32T
MO4YzppDmSuHIGI9w\/132","province":0,"city":0,"district":0,"email_validated":0,"nickname":
"summer","level":2,"discount":"1.00","total_amount":"55.00","is_lock":0,"is_distribut":0,"
first_leader":3,"second_leader":0,"third_leader":0,"token":"","message_mask":63,"push_id":
"190e35f7e07c8658ec6","distribut_level":0,"is_vip":0,"xcx_qrcode":null,"poster":null,"leve
l_name":"\u5014\u5f3a\u9752\u94dc"},"url":""}
```


分析：该接口中，请求方法、URL、固定写法，一般不会变。

而Headers中，Cookie中的sessionid是变化的，所以需要传递headers

请求体会动态变化

此外，拉勾商城登录接口，依赖获取验证码返回的sessionid，所以我们还需要封装获取验证码接口

代码演示：

```
# 封装登陆接口固定的内容
# 请求方法、URL、是固定的
# 请求体和请求头是动态的 ： 请求体是我们需要设计的请求数据，请求头中有Cookie，有令牌，令牌是会随机的

# 导包
import requests

# 创建封装的接口类
class LgShopLogin:
```

```
    def __init__(self):
        # 定义要用到的URL，把URL固定到封装的接口类当中
        self.url_verify = "http://localhost/index.php?m=Home&c=User&a=verify" # 验证码的URL
        self.url_login = "http://localhost/index.php?m=Home&c=User&a=do_login" # 登陆的URL


    # 封装获取验证码接口
    def get_verify(self):
        response = requests.get(url=self.url_verify)
        return response

    # 封装登陆接口
    def login(self, data, headers):
        return requests.post(url=self.url_login, data=data, headers=headers)



if __name__ == '__main__':
    lg_api = LgShopLogin() # 实例化封装的类
    response = lg_api.get_verify()
    print(response.content)

    # 获取cookie
    print(response.cookies)
    # 组装请求头
    headers={"Content-Type":"application/x-www-form-urlencoded",
"Cookie":"PHPSESSID=2jp22sf2d7dev2s401ujn6vll6;"}
    # 设置请求体数据
    data = "username=13800138006&password=123456&verify_code=8888"
    # 使用封装的登陆方法来发送请求
    respinse = lg_api.login(data, headers)
    print(respinse.json())
```

4.3.3 拉勾商城注册接口封装

在API中编写封装接口的代码

拉勾商城注册信息：

请求方法：POST

URL：http://localhost/index.php/Home/User/reg.html

HEADERS：

```
Content-Type:application/x-www-form-urlencoded
Cookie: PHPSESSID=vpi4htcq7ghhdt7jv3ilf74qq7;
```

Body:

```
auth_code=TPSHOP&scene=1&username=13800000002&verify_code=8888&password=519475228fe35ad067
744465c42a19b2&password2=519475228fe35ad067744465c42a19b2
```

Response:

```
{"status":1,"msg":"\u6ce8\u518c\u6210\u529f","result":
{"user_id":45,"email":"","password":"519475228fe35ad067744465c42a19b2","paypwd":null,"sex"
:0,"birthday":0,"user_money":"0.00","frozen_money":"0.00","distribut_money":"0.00","underl
ing_number":0,"pay_points":0,"address_id":0,"reg_time":1622109855,"last_login":1622109855,
"last_ip":"","qq":"","mobile":"13800000002","mobile_validated":1,"oauth":"","openid":null,
"unionid":null,"head_pic":"\/public\/images\/icon_goods_thumb_empty_300.png","province":0,
"city":0,"district":0,"email_validated":0,"nickname":"13800000002","level":1,"discount":"1
.00","total_amount":"0.00","is_lock":0,"is_distribut":1,"first_leader":0,"second_leader":0
,"third_leader":0,"token":"da9f757edfb9c0f52cb51c7996dee3a0","message_mask":63,"push_id":"
0","distribut_level":0,"is_vip":0,"xcx_qrcode":null,"poster":null}}
```

分析：该接口中，请求方法、URL、固定写法，一般不会变。

而Headers中，Cookie中的sessionid是变化的，所以需要传递headers

请求体中的scene,username、password、password2、verify_code会动态变化

此外，拉勾商城注册接口，依赖获取验证码返回的sessionid，所以我们还需要封装获取验证码接口

代码演示：

```
# 封装登陆接口固定的内容
# 请求方法、URL、是固定的
# 请求体和请求头是动态的 ： 请求体是我们需要设计的请求数据，请求头中有Cookie，有令牌，令牌是会随机的

# 导包
import requests


# 创建封装的接口类
class LgShopLogin:

    def __init__(self):
        # 定义要用到的URL，把URL固定到封装的接口类当中
        self.url_verify = "http://localhost/index.php?m=Home&c=User&a=verify"  # 验证码的URL
        self.url_login = "http://localhost/index.php?m=Home&c=User&a=do_login"  # 登陆的URL

    # 封装获取验证码接口
```

```
    def get_verify(self):
        response = requests.get(url=self.url_verify)
        return response

    # 封装登陆接口
    def login(self, data, headers):
        return requests.post(url=self.url_login, data=data, headers=headers)

    # 使用session封装获取验证码接口
    def get_verify_session(self, session):
        return session.get(url=self.url_verify)

    # 使用session封装登陆接口
    def login_session(self, session, data, headers):
        return session.post(url=self.url_login, data=data, headers=headers)


if __name__ == '__main__':
    lg_api = LgShopLogin()  # 实例化封装的类
    response = lg_api.get_verify()
    print(response.content)

    # 获取cookie
    print(response.cookies)
    # 组装请求头
    headers = {"Content-Type": "application/x-www-form-urlencoded", "Cookie":
"PHPSESSID=2jp22sf2d7dev2s401ujn6vll6;"}
    # 设置请求体数据
    data = "username=13800138006&password=123456&verify_code=8888"
    # 使用封装的登陆方法来发送请求
    respinse = lg_api.login(data, headers)
    print(respinse.json())
```

4.4 实现接口测试用例

在script中编写实现接口测试用例的代码

4.4.1 拉勾商城登录接口测试

测试点：

- 登陆成功
- 密码错误
- 验证码错误
- 手机号码没有注册

```
# 导包
import unittest, requests
from api.lgshop_api_login import LgShopLogin
# 创建测试类
class TestLgshopLogin(unittest.TestCase):

    def setUp(self):
        # 实例化封装的登陆API
        self.login_api = LgShopLogin()
        # 实例化session对象
        self.session = requests.Session()
    # 实现登陆的测试用例
    def test001_login_success(self):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = "username=13800138006&password=123456&verify_code=8888"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("登陆的结果为: ", response.json())

        # 断言
        self.assertEqual(200, response.status_code) #断言响应状态码
        self.assertEqual(1, response.json().get("status")) # 断言登陆成功之后的status的值
        self.assertEqual("登陆成功", response.json().get("msg")) # 断言登陆成功之后的msg

    def test002_password_is_error(self):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = "username=13800138006&password=1234567&verify_code=8888"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("登陆的结果为: ", response.json())
        # 断言
        self.assertEqual(200, response.status_code) #断言响应状态码
        self.assertEqual(-2, response.json().get("status")) # 断言登陆成功之后的status的值
        self.assertEqual("密码错误!", response.json().get("msg")) # 断言登陆成功之后的msg

    def test003_verify_is_error(self):
        # 获取验证码
```

```
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = "username=13800138006&password=1234567&verify_code=1234"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("登陆的结果为：", response.json())
        # 断言
        self.assertEqual(200, response.status_code) #断言响应状态码
        self.assertEqual(0, response.json().get("status")) # 断言登陆成功之后的status的值
        self.assertEqual("验证码错误", response.json().get("msg")) # 断言登陆成功之后的msg

    def test004_username_is_not_regist(self):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = "username=14388888888&password=1234567&verify_code=8888"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("登陆的结果为：", response.json())
        # 断言
        self.assertEqual(200, response.status_code) #断言响应状态码
        self.assertEqual(-1, response.json().get("status")) # 断言登陆成功之后的status的值
        self.assertIn("账号不存在", response.json().get("msg")) # 断言登陆成功之后的msg
```

4.4.2 拉勾商城注册接口测试

```
# 导包
import unittest, requests
from api.lgshop_api_regist import LgshopRegist
# 创建测试类
class TestLgshopLogin(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        # 实例化封装的注册的API
        cls.regist_api = LgshopRegist()

    def setUp(self) -> None:
        # 实例化session对象
        self.session = requests.Session()
```

```
    # 注册成功
    def test001_regist_success(self):
        # 注册的验证码
        response = self.regist_api.get_verify_session(self.session)
        print("response的结果为：", response.content)

        # 注册
        data =
"auth_code=TPSHOP&scene=1&username=13800000129&verify_code=8888&password=519475228fe35ad06
7744465c42a19b2&password2=519475228fe35ad067744465c42a19b2"
        headers = {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.regist_api.regist_session(self.session, data ,headers)
        print("注册的结果为：", response.json())


    # 注册时密码不一致
    # 注册成功
    def test002_regist_password_is_error(self):
        # 注册的验证码
        response = self.regist_api.get_verify_session(self.session)
        print("response的结果为：", response.content)

        # 注册
        data =
"auth_code=TPSHOP&scene=1&username=13800000127&verify_code=8888&password=519475228fe35ad06
7744465&password2=519475228fe35ad067744465c42a19b2"
        headers = {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.regist_api.regist_session(self.session, data ,headers)
        print("注册的结果为：", response.json())
```

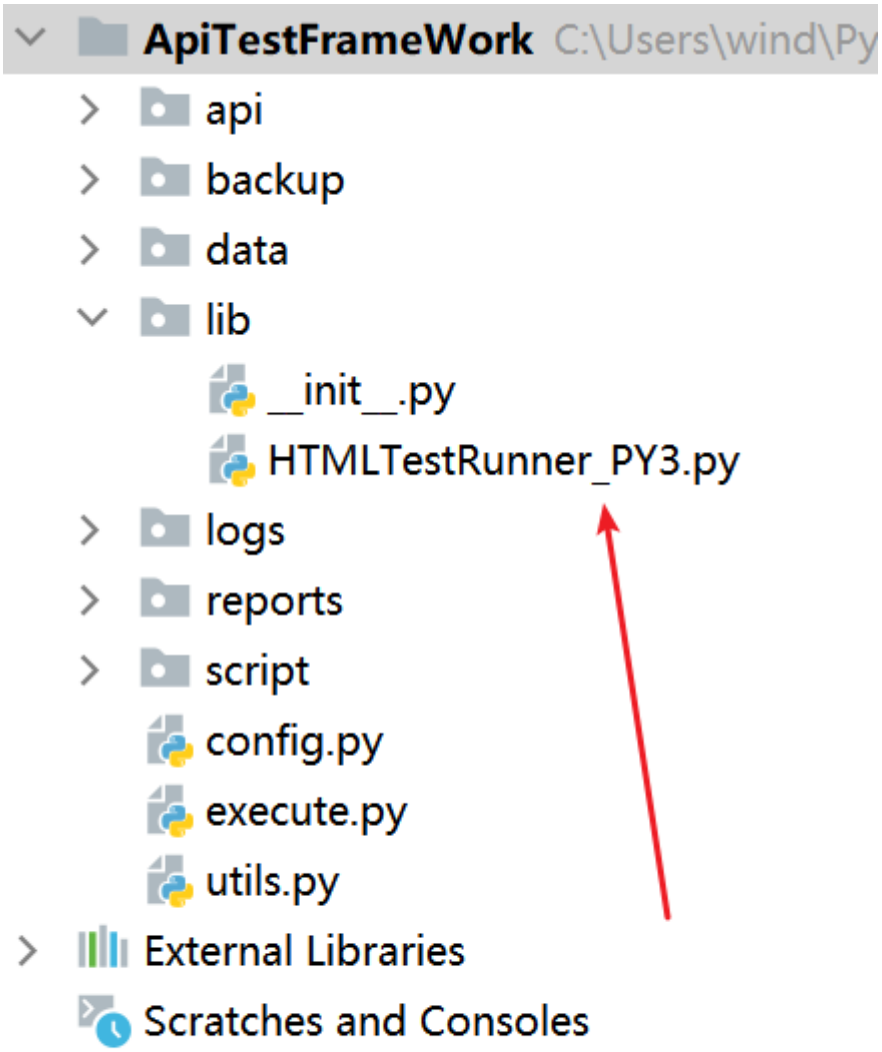
4.5 生成测试报告

进行接口测试之后，我们需要收集所有接口测试用例的执行结果，并生成HTML测试报告。

我们可以使用HTMLTestRunner来采集结果，并生成测试报告，

4.5.1 安装生成测试报告的工具HTMLTestRunner_PY3

课程上已经提供给大家了



4.5.2 编写代码生成测试报告

语法:

```
import lib.HTMLTestRunner_PY3

使用HTMLTestRunner_PY3.HTMLTestRunner(f) 来实例化runner对象
再使用runner对象运行测试用例
最后使用runner对象生成测试报告
```

```
import unittest

import lib.HTMLTestRunner_PY3
from script.test_lgshop_login import TestLgshopLogin
```

```
# 创建测试套件
suite = unittest.TestSuite()
# 将测试用例添加到测试套件
suite.addTests(unittest.makeSuite(TestLgshopLogin))
# 使用HTMLTestRunner运行测试套件生成测试报告
with open("./reports/report.html", mode='wb') as f:

    # 实例化HTMLTestRunner
    runner = lib.HTMLTestRunner_PY3.HTMLTestRunner(f)
    # 运行测试套件
    result = runner.run(suite)
    # 生成测试报告
    runner.generateReport(suite, result)
```

4.6 参数化和数据驱动

4.6.1 Parameterized库介绍

parameterized库是专门用来进行参数化工具

用法：

```
from parameterized import parameterized
@parameterized.expand([("foo", 1, 2)])
def test_add1(name, input, expected):
    actual = add1(input)
    assert_equal(actual, expected)
```

安装方法：

```
pip install parameterized
```

4.6.2 使用parameterized实现登录参数化

```
# 导包
import unittest, requests
from api.lgshop_api_login import LgShopLogin
from parameterized import parameterized

# 创建测试类
class TestLgshopLogin(unittest.TestCase):

    def setUp(self):
        # 实例化封装的登陆API
```

```
        self.login_api = LgShopLogin()
        # 实例化session对象
        self.session = requests.Session()

    # 实现登陆的测试用例
    @parameterized.expand([('13800138006','123456','8888','登陆成功',1),
                           ('13800138006','1234567','8888','密码错误!',-2)])
    def test001_login(self,username,password,verify_code,msg,status):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = f"username={username}&password={password}&verify_code={verify_code}"
        headers = {"Content-Type": "application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("登陆的结果为: ", response.json())

        # 断言
        self.assertEqual(200, response.status_code)  # 断言响应状态码
        self.assertEqual(status, response.json().get("status"))  # 断言登陆成功之后的status的
值
        self.assertEqual(msg, response.json().get("msg"))  # 断言登陆成功之后的msg
```

4.6.3 数据驱动测试

数据驱动就是分离测试数据与测试用例。而测试数据会因为存放介质的不同的而有不同的处理方式:

```
主要包括: json、xml、excel、mysql等等
```

- 通过json文件实现数据驱动测试

  设计Json数据文件

login_data.json

```json
[
  {
    "case_name": "登录成功",
    "username": "13800138006",
    "password": "123456",
    "verify_code": "8888",
    "msg": "登陆成功",
    "status": 1
  },
  {
    "case_name": "密码错误",
    "username": "13800138006",
    "password": "1234567",
    "verify_code": "8888",
    "msg": "密码错误",
    "status": -2
  }
]
```

编写读取Json数据文件为列表元组的数据

```
def read_json_data(filename):
    """
    :param filename:  要传入的外部登陆的json数据文件路径
    :return:
    """
    with open(filename, mode='r', encoding="utf-8") as f:
        jsonData = json.load(f)  # 将读取的数据流f，转化为json数据
        result_list = list()  # 定义空列表存放转化之后的数据
        for login_data in jsonData:
            result_list.append(tuple(login_data.values()))  # 将列表字典数据，转化为列表元组数据，并添加到空列表当中，整合成parameterized所需要的列表元素数据
        print(result_list)
    return result_list
```

将外部数据关联到代码中

```
    @parameterized.expand(read_json_data("../data/login_data.json"))
    def test001_login(self,case_name,username, password, verify_code, msg,status):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = f"username={username}&password={password}&verify_code={verify_code}"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("{case_name} 登陆的结果为: ", response.json())
```

- ElementTree(简称ET)处理xml文件实现数据驱动测试

语法:

```
import xml.etree.ElementTree as ET
f = open(xml_file_path) # 打开xml文件
tree=ET.parse(in_file)  # 使用ET解析xml文件树
root = tree.getroot()  # 获取xml树的根节点

root.findall("Xpath表达式")

XML文档语法结构
<node></node> 节点名称，最上层的是root节点，其他都是root节点的子节点
每个节点都会有tag、attrib、text三大属性
tag:节点名称
attrib:属性名称
text:文本名称

# 获取子节点的方法
for child in root:
    print(child)

# XPATH表达式
* 匹配任意节点
```

设计数据文件:

```
<data>
    <case>
```

```
        <case_name>登陆成功</case_name>
        <username>13800138006</username>
        <password>123456</password>
        <verify_code>8888</verify_code>
        <msg>登陆成功</msg>
        <status>1</status>
    </case>
    <case>
        <case_name>密码错误</case_name>
        <username>13800138006</username>
        <password>1234567</password>
        <verify_code>8888</verify_code>
        <msg>密码错误!</msg>
        <status>1</status>
    </case>
</data>
```

编写读取XML文档的函数：

```
def read_xml_data(filename):
    tree = ET.parse(filename) # 解析XML文件
    root = tree.getroot() # 获取root节点
    parmams_data_list = [] # parameterized工具所需要的列表数据
    for case in root: # 遍历root节点的子节点
        temp_list = []
        for input_data in case.findall("*"): # case.findall("*")把所有子节点都找出来
            temp_list.append(input_data.text) # 把子节点的text属性值添加到临时列表
        parmams_data_list.append(tuple(temp_list)) # 把临时列表的数据都转化为元组数据添加到
parmams_data_list
    print("转化的XML结果为：", parmams_data_list)
    return parmams_data_list # 返回结果
```

实现数据驱动测试

```
    @parameterized.expand(read_xml_data("../data/login_data.xml"))
    def test001_login(self,case_name,username, password, verify_code, msg,status):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = f"username={username}&password={password}&verify_code={verify_code}"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("{case_name} 登陆的结果为：", response.json())
```

- xlrd 处理EXCEL文件实现数据驱动测试

```
安装xlrd
pip install xlrd

语法:
filename="./data/login_data.xls" # 通过右键新建一个xls文档
workbook = xlrd.open_workbook(filename)
sheet = workbook.sheet_by_index(0)


sheet.nrows 获取总行数
sheet.ncols 获取总列数
sheet.row_values(0) 读取第一行数据
sheet.row_values(1) 读取第二行数据
```

设计EXCEL数据文件

case_name	username	password	verify_code	msg	status
登陆成功	13800138006	123456	8888	登陆成功	1
密码错误	13800138006	1234567	8888	密码错误!	-2

编写读取Excel的函数

```
def read_excel_data(self, filename):
    book = xlrd.open_workbook(filename)
    sheet = book.sheet_by_index(0)

    params_data = []
    for i in range(1, sheet.nrows):  # sheet.nrows 获取总行数
        params_data.append(tuple(sheet.row_values(i)))  # sheet.row_values(i) 获取i行的
数据
    return params_data
```

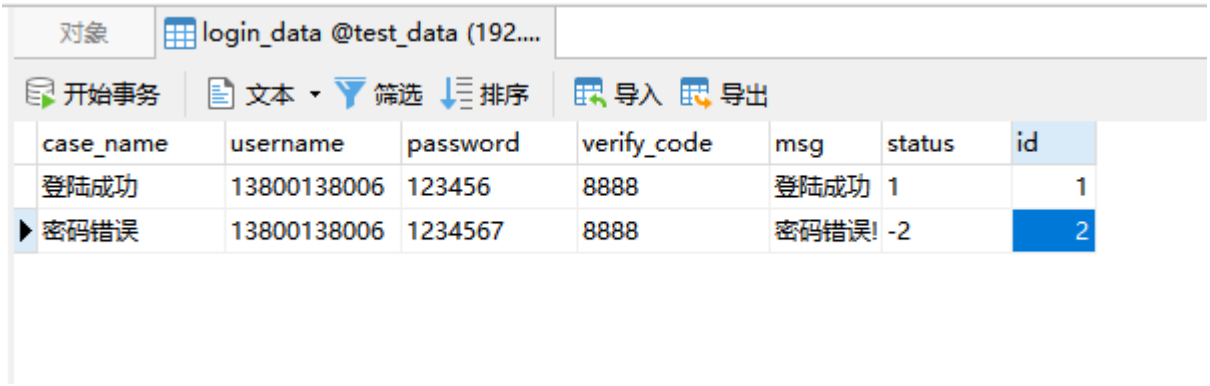
实现数据驱动测试

```
    @parameterized.expand(read_xml_data("../data/login_data.xls"))
    def test001_login(self,case_name,username, password, verify_code, msg,status):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = f"username={username}&password={password}&verify_code={verify_code}"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("{case_name} 登陆的结果为: ", response.json())
```

- pymysql处理mysql数据库实现数据驱动测试

  一般来讲，数据库需要单独创建，不能和服务器的数据库使用同一套数据库

  设计数据库的测试数据



case_name	username	password	verify_code	msg	status	id
登陆成功	13800138006	123456	8888	登陆成功	1	1
密码错误	13800138006	1234567	8888	密码错误!	-2	2

  编写读取数据库数据的函数

```
def read_mysql_data():
    # 建立连接
    conn = pymysql.connect(host="192.168.85.139",
                           port=3306,
                           user="root",
                           password="Lagou@1234",
                           database="test_data",
                           charset="utf8")
    # 获取游标
    cursor = conn.cursor()
    # 执行SQL语句
    cursor.execute('select * from login_data')
    result = cursor.fetchall()
    # 提交事务
    conn.commit()
    # 执行完成之后，在navicat当中，查看结果

    # 关闭游标
    cursor.close()
    # 关闭连接
    conn.close()

    return result
```

  实现数据驱动

```
    @parameterized.expand(read_mysql_data())
    def test001_login(self,case_name,username, password, verify_code, msg,status):
        # 获取验证码
        response = self.login_api.get_verify_session(self.session)
        # 打印验证码
        print(response.content)
```

```
        # 打印cookie
        print(response.cookies)

        # 登陆
        data = f"username={username}&password={password}&verify_code={verify_code}"
        headers= {"Content-Type":"application/x-www-form-urlencoded"}
        response = self.login_api.login_session(self.session, data, headers)
        print("{case_name} 登陆的结果为：", response.json())
```

4.6.4 【扩展】JsonPath处理Json数据

JsonPath 对于Json 相当于 Xpath 对于 XML

在进行json数据处理过程中，有的json数据比较复杂，会出现嵌套列表数据的现象；

这个时候，我们需要指定一些条件来提取目标json数据

例如：要提取下文中id为3的name时，我们会发现通过简单的json数据，很难表达出："id为3的name"

```
[{"id":"1", "name":"顺丰"}, {"id":"2", "name":"菜鸟"}, {"id":"3","name":"申通"}]
```

我们可以使用JsonPath来处理Json响应数据，还能编写条件

JsonPath的Github地址：https://github.com/json-path/JsonPath

JsonPath语法介绍：

Operator	Description
`$`	The root element to query. This starts all path expressions.
`@`	The current node being processed by a filter predicate.
`*`	Wildcard. Available anywhere a name or numeric are required.
`..`	Deep scan. Available anywhere a name is required.
`.<name>`	Dot-notated child
`['<name>' (, '<name>')]`	Bracket-notated child or children
`[<number> (, <number>)]`	Array index or indexes
`[start:end]`	Array slice operator
`[?(<expression>)]`	Filter expression. Expression must evaluate to a boolean value.

JsonPath入门案例


```
{
    "store": {
        "book": [
            {
                "category": "reference",
                "author": "Nigel Rees",
                "title": "Sayings of the Century",
                "price": 8.95
            },
            {
                "category": "fiction",
                "author": "Evelyn Waugh",
                "title": "Sword of Honour",
                "price": 12.99
            },
            {
                "category": "fiction",
                "author": "Herman Melville",
                "title": "Moby Dick",
                "isbn": "0-553-21311-3",
                "price": 8.99
            },
            {
                "category": "fiction",
                "author": "J. R. R. Tolkien",
                "title": "The Lord of the Rings",
                "isbn": "0-395-19395-8",
                "price": 22.99
            }
        ],
        "bicycle": {
            "color": "red",
            "price": 19.95
        }
    },
    "expensive": 10
}
```

JsonPath (click link to try)	Result
$.store.book[*].author	The authors of all books
$..author	All authors
$.store.*	All things, both books and bicycles
$.store..price	The price of everything
$..book[2]	The third book
$..book[-2]	The second to last book
$..book[0,1]	The first two books
$..book[:2]	All books from index 0 (inclusive) until index 2 (exclusive)
$..book[1:2]	All books from index 1 (inclusive) until index 2 (exclusive)
$..book[-2:]	Last two books
$..book[2:]	Book number two from tail
$..book[?(@.isbn)]	All books with an ISBN number
$.store.book[?(@.price < 10)]	All books in store cheaper than 10
$..book[?(@.price <= $['expensive'])]	All books in store that are not "expensive"
$..book[?(@.author =~ /.*REES/i)]	All books matching regex (ignore case)
$..*	Give me every thing
$..book.length()	The number of books

JsonPath实战案例：断言拉勾教育查询所有课程接口中，id为27的课程名称是"一拳超人"

- 拉勾教育项目：192.168.85.138:8080
- 查询所有课程接口 登陆后才能访问，所以需要令牌
- 获取响应数据

```
# 导包
import unittest, requests
import jsonpath


# 创建测试类
class TestLgshopLogin(unittest.TestCase):

    def setUp(self):
        # 实例化session对象
```

```
        self.session = requests.Session()

    def tearDown(self):
        self.session.close()

    def test001_query_all_course(self):
        # 登陆
        response = self.session.post(url="http://192.168.85.139:8080/ssm_web/user/login",
                                     params={"phone": "15321919666", "password":
"123456"})
        print(response.json())
        # 查询所有课程
        response =
self.session.post(url="http://192.168.85.139:8080/ssm_web/course/findAllCourse", json={},
                                     headers={"Content-Type": "application/json"})
        print(response.json())
        # 使用jsonpath提取复杂json语句中的结果
        self.assertEqual("一拳超人", jsonpath.jsonpath(response.json(), "$.content[?
(@.id==27)].courseName")[0])
```

5 引入日志收集功能

logging模块，我们可以对logging模块进行配置，自定义生成的 日志格式内容，

从而帮助我们打印我们所需要的日志。

配置：日志等级、日志格式

5.1 日志的基本概念

- 日志级别

  日志级别是为了控制打印日志的信息程度的

  首先，配置日志模块时，需要先设置日志模块的日志等级

  例如，如果设置为INFO级别，那么打印日志用DEBUG就不会输出日志。

  - DEBUG
  - INFO
  - WARNING
  - ERROR
  - CRITICAL
- 日志输出模式
  - 输出到控制台
  - 输出到文件
- 日志格式

  指定输出的日志的格式和内容

常见的格式有：

- **%(levelno)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

5.2 logging使用流程

第一步：实例化logging模块

第二步：设置日志等级

第三步：配置日志处理器、日志格式；

日志处理器：控制日志的打印模式

第四步：打印日志

5.3 接口测试框架实现日志收集功能

在utils.py中编写日志配置函数代码

这个函数配置了输出日志到控制台和文件，并且设置了日志打印格式

```
def logging_init():
    # 初始化日志器
    logger = logging.getLogger()
    # 设置日志等级
    logger.setLevel(logging.INFO)
    # 添加控制器
    stream_handler = logging.StreamHandler()
    file_handler = logging.handlers.TimedRotatingFileHandler(config.BASE_DIR +
"/logs/lagou_log.log", when='h',
                                                             interval=1, backupCount=3,
encoding="utf-8")
    # 设置日志格式
    fmt = "%(asctime)s %(levelname)s [%(name)s] [ %(filename)s %(funcName)s %(lineno)d ] %
(message)s "
    formatter = logging.Formatter(fmt)
    # 将日志格式添加到控制器
    stream_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)
    # 将控制器添加到日志器
    logger.addHandler(stream_handler)
    logger.addHandler(file_handler)
```

```
    return logger
```

然后在`api.__init__.py`中调用这个函数，完成日志的初始化

```
from utils import logging_init

# 初始化日志配置函数
logging_init()

# 测试打印日志
import logging
logging.info("测试info级别的日志打印")
logging.debug("测试debug级别的日志打印") #不会打印
```

在`api.__init__.py`初始化日志配置的原因：execute执行script中的用例，script中的用例是调用api的接口实现接口测试，按照模块语法，调用模块时，会自动执行模块下的`__init__.py`代码

后续，只需要在需要打印日志的模块，导入logging安装包，就可以输出我们配置好日志格式和日志等级的日志信息了

6 接口测试框架中实现登录态管理

登录态管理的本质就是管理令牌。

而令牌会使用**签名和加密** 技术实现，这部分功能设计到安全，所以正规的公司都不会提供算法给测试。

因此，我们实际在接口测试框架中，只需要实现手动输入令牌的**窗口** 就可以了。

6.1 登陆态管理的两种实现方式

- 手动管理

  在实际工作中，有的公司无法通过接口登陆，拿到登陆成功后服务器返回的令牌。

  这个时候，我们只能手动抓包，获取登陆成功后的令牌，然后录入到接口测试框架中，完成登录态管理。

- 自动管理

  实际工作中，有的公司会通过各种方法，让接口测试人员能够成功调用登陆接口，拿到令牌，这个时候，我们可以使用自动化的方法，自动管理令牌。

阻止调用登陆接口的坑有：

- 验证码

处理方式，任选一种

- 关闭验证码校验
- 设置后门万能验证码（上线时关闭后门）
- 在后台记录验证码，并录入数据库，测试用时通过数据库查询（上线时关闭）
- 白名单

- 密码

  处理方式

  - 发动口才，拿到加密算法，使用这个算法加密密码，得到登陆时所需要的密码。
  - 关闭密码加密

6.2 案例：拉勾商城令牌管理

在config.py中，新增Cookie变量

```
COOKIES = {"JSESSIONID": "c6a0272f-e39c-417a-b696-3785f2543227"}
```

后期调用需要令牌的接口时，都把Cookie传入请求头即可

示例代码：

```
import requests

headers = {"Content-Type":"application/x-www-form-urlencoded", "Cookie":
"PHPSESSID=vjsi7vhui4d3s7tqmvqnsepik0"}

response = requests.post(url="http://localhost/index.php?m=Home&c=User&a=do_login",
              data="username=13800138006&password=123456&verify_code=8888",
              headers=headers)


print(response.json())
```

总结：

登陆态管理方式：

- 在浏览器页面中登陆，获取令牌，然后把该令牌输入到请求头中
- 如果公司安全管理不严格，可以登录成功，那么可以通过session登录，自动管理cookie中的令牌

7 接口加密和加签

接口测试中，密码这类数据通常都会被加密；有的接口整个都会被加签，如果我们不加密密码，不对接口加签，那么我们无法调通接口，进行接口测试。

通用实现思路：

- 获取加密算法
- 使用加密算法对数据进行加密
- 传递加密之后的数据

拉勾商城注册接口中，注册的密码需要使用MD5对密码进行签名后才能传输。

在Python中，我们可以使用hashlib库来实现该功能

语法

```
import hashlib
md5_data = hashlib.md5().update("原文") # 加密原文为md5数据，md5加密不可逆，无法还原
print(md5_data.hexdigest()) # 打印加密后的数据
```

在utils.py中编写如下代码：

```
import hashlib
# 封装MD5加密函数
def MD5_demo(data):
    authcode = "TPSHOP"
    str = authcode + str
    md = hashlib.md5()  # 创建md5对象
    md.update(data.encode(encoding='utf-8'))
    return md.hexdigest()
```

在注册接口中引用

```
import unittest
from unittest import TestCase
from api.regist_api import RegisterApi


class TestLogin(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
```

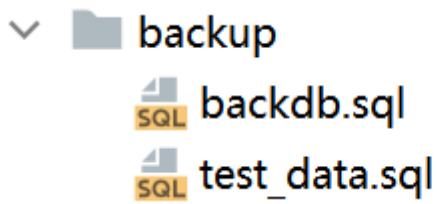
```
        cls.regist_api = RegisterApi()

    def setUp(self):
        self.headers = {"Content-Type": "application/x-www-form-urlencoded"}

    def test01(self):
        # 获取验证码
        response = self.regist_api.get_verify()
        cookies = response.cookies
        self.headers["Cookie"] = cookies
        # 注册
        response = self.regist_api.regist(1, "13800138006", MD5_demo("123456"),
MD5_demo("123456"), "8888", self.headers)
        print("注册的结果为：", response.text)
```

8 服务端数据备份/还原/初始化

多次进行注册接口测试时，你会发现，无法重复注册，这是因为账号只能注册一次。

可是你希望能多次注册怎么办?

我们可以准备一个初始化的数据文件，在注册之前进行初始化，这样就可以保证每次的数据都是没有注册过的。



8.1 备份数据方法介绍

Mysql数据库可以通过mysqldadmin来备份数据

```
mysqldump -h 服务器 -u用户名 -p密码 数据库名 数据库表 > 备份文件.sql
```

然后我们在python中，只需要在执行用例前运行备份的命令即可

```
os.system("mysqldump -u root -h host -proot  tpshop2.0 tp_users > filename.sql")
```

8.2 初始化数据方法

Mysql数据库可以通过mysql命令直接执行SQL文件，从而完成数据的初始化

```
mysql -h host -u root -proot ssm_lagou_edu < 初始化的数据文件
```

然后我们在python中，只需要在执行用例前运行初始化数据的命令即可

```
os.system("mysql -u root -h host -proot  ssm_lagou_edu > 初始化的数据文件")
```

8.3 数据的还原

测试完成之后，我们需要还原数据

Mysql数据库可以通过mysql命令直接执行SQL文件，从而完成数据的初始化

```
mysql -h host -u root -proot ssm_lagou_edu < 要还原的数据文件
```

然后我们在python中，只需要在执行用例前运行初始化数据的命令即可

```
os.system("mysql -u root -h host -proot  ssm_lagou_edu > 要还原的数据文件")
```

###