自动化测试

* 接口自动化httprunner框架

charles抓包保存返回结果为har格式文件—>

转化成yaml文件格式: har2case test.har -2y

-config:

name: testcase name

全局变量设置

-test

name:

request:

data:

headers:

method:

url:

validate:

- eq:

- status\_code

- 200

- gt/

运行与调用：用jenkins 持续集成自动化测试

jenkins创建job，给job配置构建步骤（执行shell），

定时执行jenkins job: jenkins上配置中选择触发机制，线上定时触发，提测是git提交代码后触发

* UI自动化appium + python

1. appium原理：appium启动时会创建一个服务端，指定端口和地址[http://127.0.0.1:4723/wd/hub](http://127.0.0.1:4723/wd/hub%E6%9C%8D%E5%8A%A1%E7%AB%AF(%E7%9B%B8%E5%BD%93%E4%BA%8E%E4%B8%80%E4%B8%AA%E4%B8%AD%E8%BD%AC%E7%AB%99)%EF%BC%8C%E8%84%9A%E6%9C%AC%E4%BC%9A%E5%91%8A%E8%AF%89%E6%9C%8D%E5%8A%A1%E5%99%A8%E6%88%91%E8%A6%81%E5%81%9A%E4%BB%80%E4%B9%88%EF%BC%8C%E6%9C%8D%E5%8A%A1%E7%AB%AF%E5%86%8D%E5%8E%BB%E8%B7%9F%E8%AE%BE%E5%A4%87%E6%89%93%E4%BA%A4%E9%81%93%E3%80%82)

，运行测试用例的服务器先初始化一个appiumserver连接，向appium发送case执行的请求，appium server默认端口监听到请求后再去和设备通信，会给设备丢个bootstrap.jar包，该包启动后创建一个socket和appium server进行通信，接受传过来的命令，然后bootstrap.jar通过操作uiautomator来在设备上执行最终的操作，操作结果原路返回给appium，返回给case服务端，再进行之后的case

1. **在 UI 自动化用例中如何切换 Android 原生页面和 H5 页面**

查看句柄：print driver. contexts//需要app开webview

切换：driver.switch\_to.contexts(contexts[1]) //context[0]=”NATIVE\_APP”

mitmproxy获取接口数据+执行自动化case:

mitmdump -s mproxy.py -p 8081

mproxy.py:

def response(flow: mitmproxy.http.HttpFlow):

searchurl= flow.request.host+”接口路径”

if flow.request.url.startswith(searchuri):

data=json.loads(flow.response.text)

with open(‘res.txt’, ‘w’) as f:

逻辑处理

f.truncate()//截断文件，默认当前位置，则清空文件内容

desired\_capabilities 参数：

platformName

deviceName--adb device

apppackage--

appactivity--adb shell; logcat |grep cmp=

f.write(data)

f.close

1. appuim case: unittest框架+selenium.webdriver

class test(unittest.TestCase):

def setUp(self):

初始化工作

初始化appium配置

初始化webdriver连接

self.driver=webdriver.Remote(‘http://localhost:4723/wd/hub’, desired\_cacabilities)

可通过工具sdk UIAutomateViewer

或者 appium desktop inspector

driver.find\_element\_by\_xpath()path

1. 通过id定位，resrouce-id

find\_element\_by\_id(id)

2.通过ClassName定位： classname

3.通过Accessiblityld定位：content-desc

def testcases(self):

执行case

定位元素

操作元素

def tearDown(self):

退出driver,关闭被测应用的所有的关联窗口

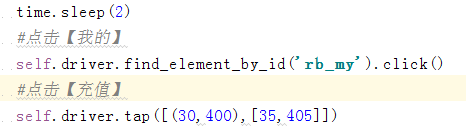
self.driver.quit()

操作元素

1点击:

**click():** 直接对页面元素对象进行点击操作

**tap([(x,y)(x2,y2].....)**：点击屏幕上的坐标位置，可放多组坐标，实现多点触摸

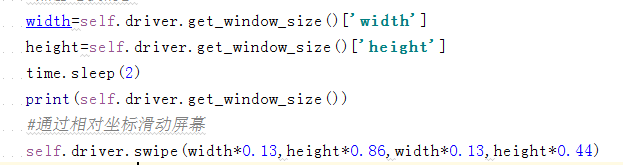


2.滑动/翻页：

**swipe（start\_X,start\_y,end\_x,end\_y,duration=n）**：在n毫秒内从（start\_X,start\_y）滑动到（end\_x,end\_y），通过设置n的大小实现快速或慢速滑动的效果。

**flick(start\_X,start\_y,end\_x,end\_y)**：快速滑动,通过调整x,y的坐标值可实现上滑、下滑、左滑、右滑的操作。

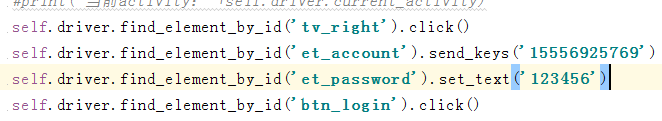
**考虑适配：先获取屏幕大小，再采用相对坐标值进行滑动操作，如下：**



3.输入：

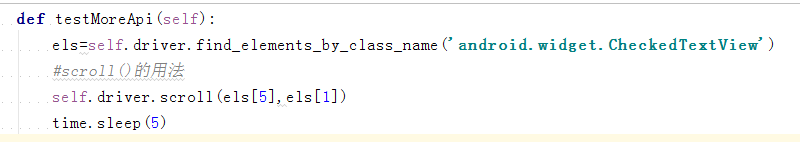
**send\_keys(‘要输入的文字’)**：调用设备当前系统输入法键盘，向文本框中输入数据

**set\_text(‘要输入的文字’)**：不调用键盘，直接向文本框中输入数据



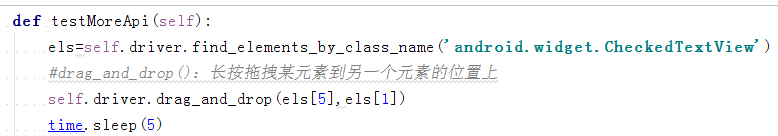
4.滚动

scroll()



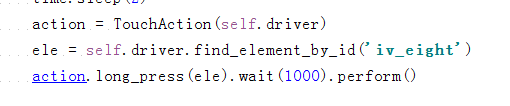
5..长按拖拽某元素到另一个元素的位置上

drag\_and\_drop()



6.长按：

long\_press()

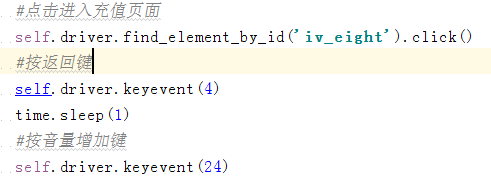


7.放大和缩小：

zoom()、pinch()：对元素进行放大一倍或缩小一倍的操作

8.发送按键码

keyevent( 按键码 )：不同的按键码对应不同的手机上不同的键



appium截图对比

select.screenshot(screenPath)//截图

对比：

imagehash.average\_hash(Image.open(tpath))

photohash.hash\_distance(str(hash1),str(hash2)))<=5即可

**import imagehash**

**import photohash**

**import Image  
  
with** open(image1, 'rb') **as** fp:  
 hash\_1 = imagehash.average\_hash(Image.open(fp))  
**with** open(image2, 'rb') **as** fp:  
 hash\_2 = imagehash.average\_hash(Image.open(fp))  
distance = photohash.hash\_distance(str(hash\_1), str(hash\_2)) #汉明距离 小于等于5时我们认为这两张图非常相似，大于10 则认为这两张图完全不一样  
# distance = 0:image1和image2完全不像；distance=1：image1和image2完全一样；distance=2：image1和image2部分重叠  
**if** distance < 0:  
 distance = -distance  
**if** distance <= max\_dif:  
 **return** 1  
**else**:  
 **return** 0

汉明距离是一个概念，它表示两个（相同长度）字对应位不同的数量，我们以d（x,y）表示两个字x,y之间的汉明距离。对两个字符串进行异或运算，并统计结果为1的个数，那么这个数就是汉明距离。

* webUI自动化 selenium+groovy
* **unitTest python的单元测试框架**

unittest的工作原理：

testLoader 将Testcase 加载到testsuite中，然后textTestRunner来运行testsuite, 运行结果放到textTestReault中，整个过程集成在unittest.main模块中。

**TestCase**：一个TestCase的实例就是一个测试用例。测试用例就是指一个完整的测试流程，包括测试前准备环境的搭建（setUp），执行测试代码（run），以及测试后环境的还原（tearDown）。

**TestSuite**：一个TestCase的实例就是一个测试用例。测试用例就是指一个完整的测试流程，包括测试前准备环境的搭建（setUp），执行测试代码（run），以及测试后环境的还原（tearDown）。

**TestRunner**

**TestFixture**

一个testcase必须是unittest.TestCase子类,测试方法应以test 开头

import unittest

class TestStringMethods(unittest.TestCase):

def **test**\_upper(self):

self.assertEqual('foo'.upper(), 'FOO')

运行单元测试

在文件最后加入

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

python -m unittest 文件名

unittest的用法

TestCase

* 常用的断言

assertEqual(a, b) a == b

assertNotEqual(a, b) a != b

assertTrue(x) bool(x) is True

assertFalse(x) bool(x) is False

assertIsNone(x) x is None

assertIsNotNone(x) x is not None

assertIn(a, b) a in b

assertNotIn(a, b) a not in b

* 跳过测试用例

使用 skipXxx 装饰器来跳过测试用例。unittest 一共提供了 3 个装饰器，分别是 ＠unittest.skip(reason)、@unittest.skipIf(condition, reason) 和 ＠unittest.skipUnless(condition, reason)。其中 skip 代表无条件跳过，skipIf 代表当 condition 为 True 时跳过；skipUnless 代表当 condition 为 False 时跳过。

使用TestCase 的 skipTest() 方法来跳过测试用例。

* TestFixture

setUp()&tearDown()：在每个测试方法（用例）运行时被调用一次

setUpClass()&tearDownClass(): 每个类只调用一次。必须使用@classmethod 装饰器

setUpModule()&tearDownModule(): 整个文件级别上只调用一次 setUp/tearDown

* TestSuite

unittest.main()是将模块下的测试用例全部执行，而且是按ascii码顺序执行测试用例,使用TestSuit可以自定义测试组件 可以使用TestLoader来加载TestCase到TestSuite中。

unittest.TestLoader().loadTestsFromTestCase(testCaseClass)

unittest.TestLoader().loadTestsFromModule(module)

unittest.TestLoader().loadTestsFromName(name,module=None)

unittest.TestLoader().loadTestsFromNames(names,module=None)

unittest.TestLoader().discover(start\_dir，pattern ='test \* .py'，top\_level\_dir = None)

* TestRunner

TextTestRunner.run(suit)开始执行测试用例,TextTestResult用来保存测试用例结果