

我们是在项目内落地自动化测试体系的

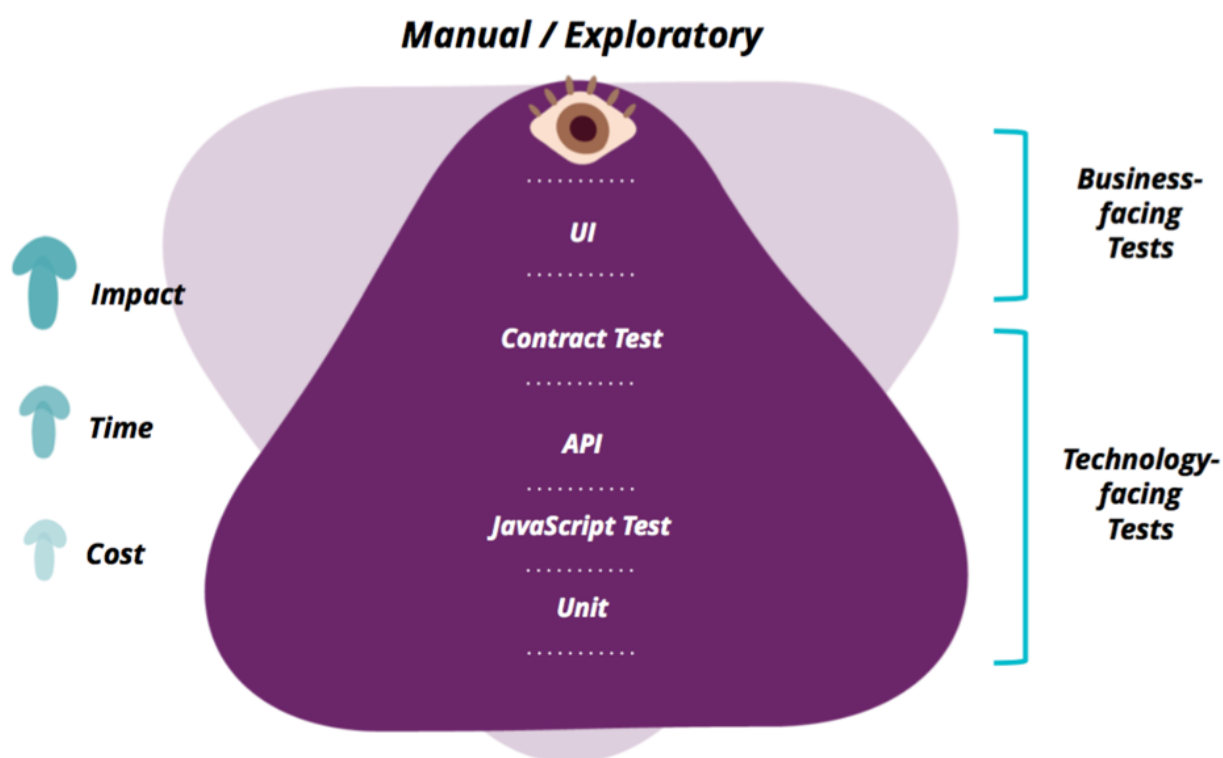
相信绝大多数从事测试行业的同志们对自动化测试有抱有一个美好的幻想，可是到底该怎么实现和落地呢？接下来我将结合分层测试金字塔和实际案例为大家分享：

项目背景： 案例项目是ThoughtWorks的内部招聘看板系统，主要服务于ThoughtWorks在19个国家49家办公室的招聘团队。核心功能是为HR们提供一个展示候选人的信息的面板，生成和招聘数据有关的报表。

一、了解自动化测试金字塔

说起自动化测试不得不提及测试金字塔，这种三角形的结构主要为我们展示了一个健康的自动化测试体系应该是什么样子的。如图所示，金字塔的从上往下依次是UI测试，接口测试，单元测试，越在高层影响就越大，花费的时间和精力就越多。图示的测试金字塔只是一种形态示例，不同项目的金字塔的实现内容可能略有区别。

在《google软件测试之道》曾写过：对于google产品，70%的投入为单元测试，20%为集成、接口测试，10%为UI层的自动化测试。



二、分层实现

如何实现UI层的自动化测试

在测试金字塔中可以看到，UI层面的自动化影响大、变化多、可维护成本高，所占比例也最少。所以在我们看来，UI层面的自动化应该是一些从高层次上验证一些happy pass，确保我们的核心功能可用。

拿案例项目来举例，我们识别出最核心的功能就是通过过滤器筛选数据来进行展示或计算，所以我们的用例主要会去验证操作这些过滤器能否筛选出正确的数据。在技术选型上，我们主要用到BDD的思想，选用cucumber + capybara这套体系去做UI层的自动化。

```
Scenario: test office filter in China region
  Given I login to GoHire website
  When I select "China" region
  And I select "Chengdu" option
  Then I can only see all the application cards in chengdu
office
```

UI层自动化的小经验

- UI层自动化的适用场景是做核心功能的回归测试和冒烟测试，所以在实施过程中，要注意不要把所有的用例都堆砌在UI层，而是尽可能放到接口测试和单元测试中去做。
- 在代码层面，我们可以遵循page object的设计模式，避免在测试代码中直接操作html元素。这样就可以减少重复的代码并提高代码的可维护性。

如何实现接口层的自动化测试

在案例项目中，接口测试主要分为两个部分。

- 在开发过程中，测试人员会和开发合作去写接口的功能测试。
- 在整个功能大概完成，API已经基本确定后，测试和开发一起结对写性能测试。

接口功能测试

在写功能测试的过程中，我们可能会和一些其他的模块或第三方API有依赖，在这种情况下，通常可以通过Mock的方法去解决。

如案例项目中有一个测试场景是：调用一个第三方的API，当这个API出错时，需要接受到API返回的错误码并对错误码进行处理。在实现测试的时候，我们没有办法让第三方API真正的挂掉并返回错误码，所以我们只需要模拟这个请求出错，验证我们代码已经对这种错误进行过处理。

```

def should_return_error_message_when_request_failed(self):
    event = {
        "body": {
            "action": "update_candidate",
            "payload": {
                "candidate": {
                    "id": 101
                }
            }
        }
    }
    with requests_mock.mock() as request_mock:
        request_mock.get("http://api.gh.test/v1/candidates/101",
text=' ', status_code=500)
        with self.assertRaises(Exception) as context:
            webhook.handle(event)
        self.assertTrue(
            '[request error][update_candidate] get candidate 101 from
Harvest API failed' in context.exception)

```

接口性能测试

在功能大致完成后，我们开始做API的性能测试。性能测试在这里的作用主要是获取我们API现有的性能指标，形成一个对比的基线。在便于进行后期的优化的同时也有可能帮助我们发现一些潜在的bug。

小故事：我们在手工测试API的响应速度时，测试结果一切正常。当引入性能测试后就发现，这些接口在前期的响应时间确实很快，可是在请求了一定次数后会突然变得很慢。经过调查我们发现，这是因为我们依赖了一个AWS的服务，这个服务有访问频率的限制，在最初的代码中，每次访问都会请求这些服务并读取这些服务的配置，这也就导致了测试发现的问题。后来我们更改了方案，把读取配置这个操作放在系统初始化的时候去做，顺利的解决了这个问题。

性能测试我们主要选用了Locust这个框架。如我们的一个接口功能是：查询某个国家下的所有的候选人。下面代码用例的意思就是，同时配置50个客户端去访问API，每次随机请求某个国家的所有候选人信息，一共请求1000次，规定每次请求的时间最大不超过5s。

```

def get_applications_in_country(l):
    text = """Australia Brasil Canada China Chile Ecuador Germany
India Italy Singapore Spain Turkey UK USA Uganda Thailand"""

    countries = text.strip().split()
    country = random.choice(countries)
    params = {
        "country": country,

```

```

        "status": "active"
    }

    with l.client.get("/getCandidates", params=params, headers={"x-api-key": API_KEY}, catch_response=True) as response:
        if "errorMessage" in response.text:
            response.failure("Error occurs in response: %s" % response.text)

class UserBehavior(TaskSet):
    tasks = {
        get_applications_in_country: 1
    }

function runLocustDataStoreService {
    validateEnvironment "HOST" "API_KEY"

    setupEnv

    CLIENTS=${CLIENTS:-"50"}
    HATCH_RATE=${HATCH_RATE:-"2"}
    NUM_REQUEST=${NUM_REQUEST:-"1000"}

    locust -f DataStoreService/locustfile.py --host ${HOST} --clients=${CLIENTS} --hatch-rate=${HATCH_RATE} --num-request=${NUM_REQUEST} --no-web --only-summary
}

```

通过Locust跑完用例，我们可以看到在console生成的report：

Name	# reqs	# fails	Avg	Min	Max	Median	req/s
GET /api/candidates?country=Australia	45	0(0.00%)	369	312	502	360	0.40
GET /api/candidates?country=Brasil	55	0(0.00%)	345	290	498	330	0.10
GET /api/candidates?country=Canada	57	0(0.00%)	188	120	989	170	0.40
GET /api/candidates?country=Chile	67	0(0.00%)	241	183	473	230	0.10
GET /api/candidates?country=China	61	0(0.00%)	715	546	2404	630	0.10
GET /api/candidates?country=Ecuador	73	0(0.00%)	335	253	564	320	0.30
GET /api/candidates?country=Germany	68	0(0.00%)	332	254	521	320	0.30
GET /api/candidates?country=India	65	0(0.00%)	1433	1243	2813	1400	0.30
GET /api/candidates?country=Italy	56	0(0.00%)	165	131	281	160	0.30
GET /api/candidates?country=Singapore	49	0(0.00%)	192	146	346	190	0.20
GET /api/candidates?country=South+Africa	54	0(0.00%)	173	121	304	170	0.20
GET /api/candidates?country=Spain	56	0(0.00%)	202	139	308	200	0.30
GET /api/candidates?country=Thailand	61	0(0.00%)	169	120	394	160	0.40
GET /api/candidates?country=Turkey	57	0(0.00%)	224	175	300	220	0.20
GET /api/candidates?country=UK	54	0(0.00%)	364	288	596	350	0.10
GET /api/candidates?country=USA	60	0(0.00%)	426	343	703	400	0.00
GET /api/candidates?country=Uganda	62	0(0.00%)	167	133	347	160	0.10

图片展示的是report的一部分，reqs代表对这个API的请求数目，fails记录了失败的次数，Avg、Min、Max、Mediam分别是每次请求响应时间的平均值、最小值、最大值和中位数。

如何实施单元测试级别的自动化

单元测试是对软件中最小的测试单元进行验证，在这一层上发现问题时解决成本最低，测试用例的维护成本也不高，具有很好的投入产出比。一般情况下，我们是需要开发人员在开发过程中写单元测试。而作为一个QA，我们更多的是一个单元测试的引导者：

- 和团队一起制定单元测试覆盖率的标准。

如果这是一个全新的项目，我们可以把覆盖率设的相对高一点，如85%，这有利于我们在前期就对代码质量做出保证。如果这是一个已经相对成熟的项目，由于前期根本没有单元测试，我们可以先把要求设置的低一点，然后一步步的提升我们的代码覆盖率。

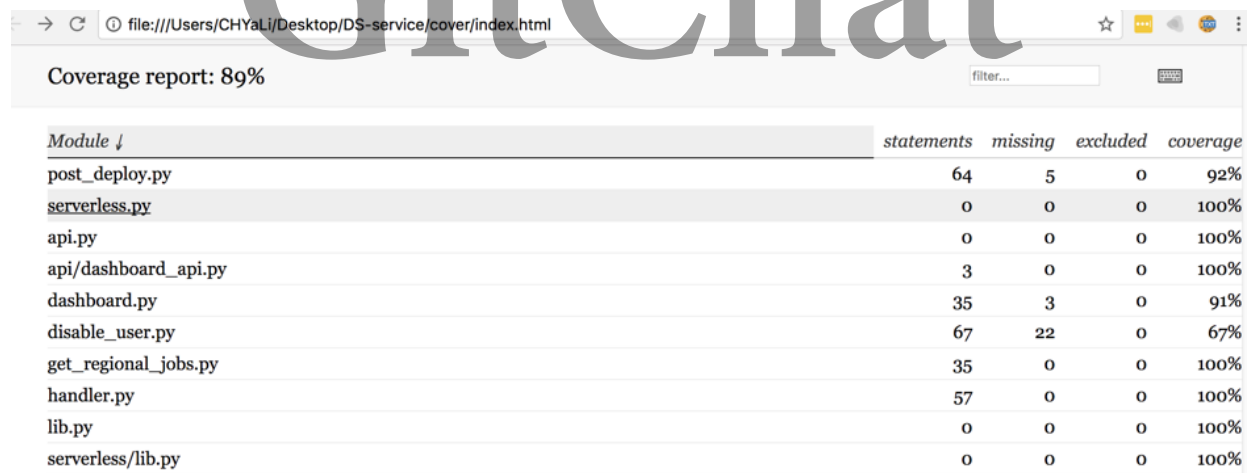
- 为开发人员提供单元测试的用例。

我们需要提前把需要验证的用例列在开发的任务卡片里面，这样能帮助开发更有效率的去完成我们期望测试的用例。

- 定期回顾开发人员写的单元测试。

这里并不是要检查代码和具体实现，而是和开发一起去回顾看看单元测试的写法和角度是不是在同一认知上面。这样有助于整个团队建立一种质量保证的意识。

在具体实现上，我们选择nose test这个工具去做单元测试，通过nose test的插件，我们可以拿到单元测试覆盖率的报表，在第二个图中，我们可以看到，没有被测试覆盖的代码会有红色的标记，这样就有利于我们找到测试的遗漏点。



The screenshot shows a web browser window with the address bar displaying 'file:///Users/CHYali/Desktop/DS-service/cover/index.html'. The page title is 'Coverage report: 89%'. Below the title is a table with the following columns: 'Module', 'statements', 'missing', 'excluded', and 'coverage'. The table lists several modules and their corresponding coverage statistics.

Module	statements	missing	excluded	coverage
post_deploy.py	64	5	0	92%
serverless.py	0	0	0	100%
api.py	0	0	0	100%
api/dashboard_api.py	3	0	0	100%
dashboard.py	35	3	0	91%
disable_user.py	67	22	0	67%
get_regional_jobs.py	35	0	0	100%
handler.py	57	0	0	100%
lib.py	0	0	0	100%
serverless/lib.py	0	0	0	100%

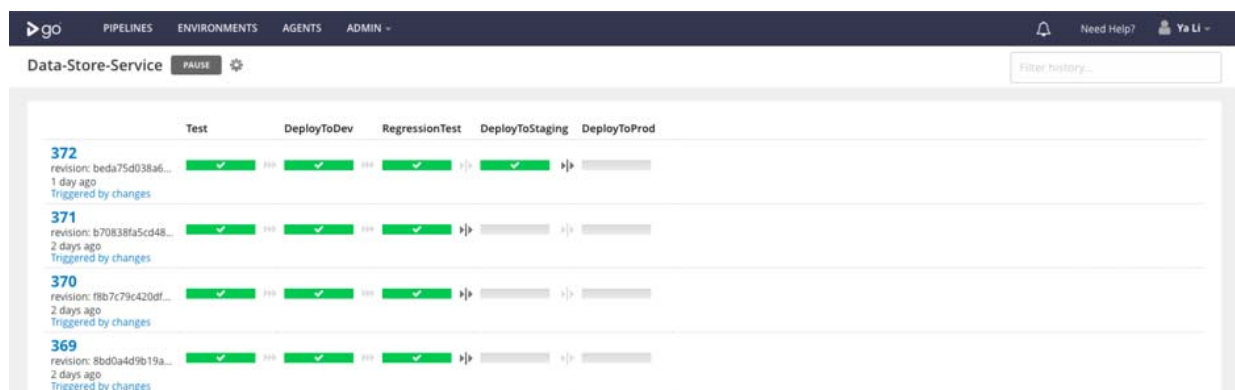
```
def get_application(application_json):
    if application_json.get("current_stage") is None:
        current_stage_name = None
    else:
        current_stage_name = application_json.get("current_stage").get("name")
    application = {
        "id": application_json.get("id"),
        "prospect": application_json.get("prospect"),
        "candidate_id": application_json.get("candidate_id"),
        "current_stage_name": current_stage_name,
        "last_activity_at": application_json.get("last_activity_at"),
        "status": application_json.get("status"),
        "job": None
    }
    jobs_json = application_json.get("jobs")
    if len(jobs_json) > 0:
        application["job"] = {
            "id": jobs_json[0].get("id"),
            "name": jobs_json[0].get("name")
        }
    return application
```

在写单元测试时，为了解决对数据库的依赖，我们可以建立一个内存数据库去模拟真实数据库，便于我们的测试用例能快速的运行。如在我们的真实项目中，我们的数据库选用的是亚马逊的RDS + Postgres，但是在做单元测试的时候我们使用的sqlite + python绑定来模拟真实的数据库

三、做完自动化测试之后还可以做什么

只让这些自动化测试运行在本地IDE上是不够的，在我们的项目中，我们建立了一套持续集成部署的体系：

1. 推送到代码到远端后会自动开始运行自动化测试和代码审查。
2. 当单元测试通过后会自动部署到测试环境。
3. 在部署完成后会自动生成测试报告。
4. 小组所有成员会收到部署成功或失败的邮件提醒。



在工具的选择上，我们的持续集成平台是ThoughtWorks的GoCD，其他类似的工具还有jenkins，可以灵活的选用这些工具。

四、总结

在实际的项目实施过程中，我们其实是按照下面的步骤依次逐步实施我们的持续集成自动化体系的：

1. 在项目开始之前首先搭建持续集成的框架，第一次的时候先写一个最简单的单元测试，如 $1+1=2$ ，确保可以在CI上运行测试，为后续的开发奠定基础。
2. 开发在项目实现过程中进行单元测试，每次开发推送代码时都可以自动运行单元测试和代码风格审查，当单元测试覆盖低于85%或代码风格检查不通过时，构建就会失败。
3. 测试和开发在项目实现过程中合作写接口层的功能测试。
4. 功能开发大体完成后，测试和开发合作写接口的性能测试。
5. 当项目发布之后，测试开始根据核心功能编写UI层面的自动化测试，也相当于是写项目的回归测试。

GitChat

最后谈一点心得体会吧：

1. 项目只有UI自动化测试是不够的，越低层的自动化测试反而越有意义。
2. 自动化测试的目的是减少重复的手动测试的成本，使测试人员可以做更多有意义的事情，在实现自动化的过程中，我们花费的精力甚至更多。
3. 测试并不是越多越好，除了用例数量还要考虑维护代价。我们希望测试代码能够尽量稳定，因为代码需要不断的被重构，如果发现重构一次代码就修改很多测试，那么这种测试可能会成为负担，也是一种坏味道。
4. 测试人员在自动化测试落地的实践中，更多的是一个推动者而不是实现者，我们需要帮助团队建立起一种质量保证的意识，然后共同实现自动化测试的落地。