业务团队如何高效实施自动化测试

摘要

美团酒旅的终端研发团队，在2017年实施了大规模的自动化测试。酒旅的终端研发团队负责美团在住宿、境内度假、大交通方向的业务研发工作，是一支面向业务的平台型团队。

由于其业务属性，客户端一直面临着业务需求压力大、业务需求迭代演进迅速的挑战。这样团队是否适合实施自动化测试？是否需要付出高额的成本？是否能够获得足够的收益？

下面由我来分享酒旅这次自动化实践的主要历程，相信一定能够给各位朋友带来帮助。

作者简介

梁士兴，2009年毕业于北京航空航天大学。毕业后在IBM中国研发中心工作了5年。2014年7月份加入到美团大众点评，现职位为技术研究员。

作为美团酒旅基础服务平台负责人，经历了美团客户端业务架构演变的全部过程，因而对客户端的业务架构有了进一步的理解和思考，愿意与大家进行分享。

业务团队是否应该实施自动化测试

按照固有的思维模式，业务团队通常面临业务压力大、需求迭代频繁等诸多痛点。业务压力大，意味开发自动化测试程序的人力成本是个巨大的挑战；而需求迭代频繁，则意味着自动化测试的稳定性很难得到保障。这是否意味着业务团队无法，或者很难实施自动化测试呢？

美团点评的酒旅终端团队，是典型的业务型团队。负责美团酒旅在住宿、境内度假、大交通方面的业务建设。酒旅的终端团队在2017年实施了大规模的自动化测试建设，因而在这一领域积累了不少经验和教训。我希望从团队实际遇到的困难与挑战入手，结合自动化测试的意义，帮助大家决策是否要实施自动化测试。

业务团队面临的困难和挑战

酒旅终端团队的痛点：

历史需求时间跨度长，质量保障困难：时间跨度最长的超过三年，一些细节包括产品、开发、QA都不能很好的解释清楚

尽管开发团队和QA同学都尽力去提升产品研发质量，但是仍然故障不断。

苹果封杀热补技术：

由于苹果与年初禁用了类似JSPatch的热补技术，“出现问题线上修复”的质量保证手段也不复存在。

业务对开发效率提出更高的要求

产品经理对研发团队的诉求概括起来通常只有两点：

- 能不能多做几个需求？
- 能不能早一点上线，最好明天就上？

如果认真的对待这两个问题，本质上是业务团队在研发效率和迭代频率方面，对研发团队提出了更高的要求。

综上所述，酒旅终端研发团队实际上面临的挑战就是：

- 需要更好的质量保障手段，包括保障的效果（少出bug）和时机（上线前暴露bug）；
- 需要更高的研发效率和产品迭代频率。一个字，快！

自动化测试的意义

软件测试的定义是：在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程[1]。

自动化测试是使用软件工具和既定程序，对软件所进行的测试活动。

对于前面提到，团队面临的第一个困难–即历史需求时间跨度长，如果有自动化测试约束，则其演变一定是一个受控的过程。这恰恰说明了实施自动化测试有非常重要的意义。

团队面临的第二个困难，相对于热补技术这种事后补救技术，自动化测试才是更根本性的质量保障手段。年初有篇文章曾经发起过讨论，硅谷的互联网公司并不热衷于热补这类“黑科技”，而是更多关注利用自动化测试技术，让应用在上线之前就有更好的质量保障。

第三个困难，需要提升研发效率和迭代频率。表面上看，似乎和自动化测试并没有太明显的关系。但随着我们的深入思考，发现了一个奇妙的结论：自动化测试是提升迭代效率的关键，也是提升研发效率的有效手段。见图1

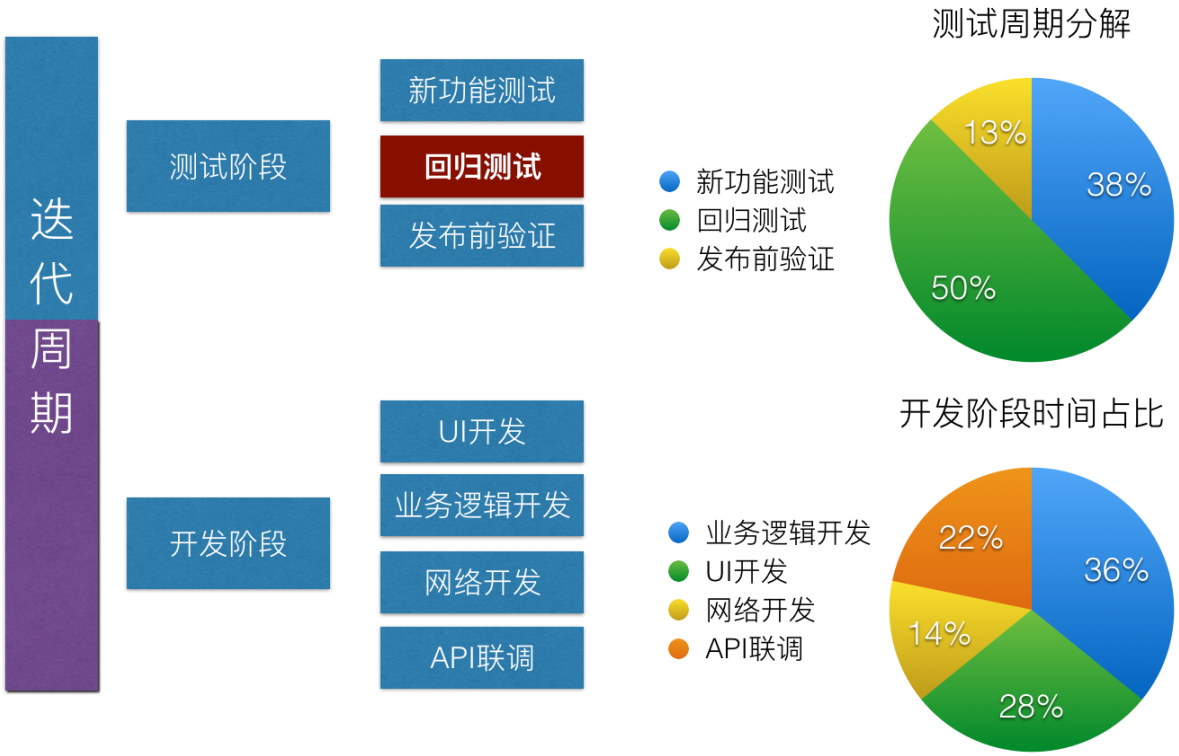


图1 酒旅迭代周期时间分配分析

从上图可以看到，每个迭代周期中，开发时间与测试时间的比例达到5：4。而回归测试时间占据了测试时间的50%。自动化测试手段可以大幅缩减回归测试的时间成本，因此可以大幅提高迭代的频率！

如何实施自动化测试

明确了自动化测试的意义，下一步应讨论如何实施自动化测试。作为团队的重要技术项目，我们需要制定明确的目标。

如何评估自动化测试的收益

评估自动化测试的收益，需要明确短期和长期目标的设定。从实际收益来看，则需要明确“能否测出bug”、“测试用例维护成本”、“减少了多少手工测试成本”。覆盖率指标则被当做过程管理的抓手，包括测试用例的覆盖率和代码执行的覆盖率。

- 目标：
  - 短期目标：消灭SA级别的线上故障
  - 长期目标：减少75%的线上故障；发版频率从月提高到周

- 收益衡量方法
  - Bug的测出率（能不能测出来bug）
  - 测试用例的失效率（测试用例的维护成本高不高）
  - 减少了多少手工测试时间

- 过程指标：

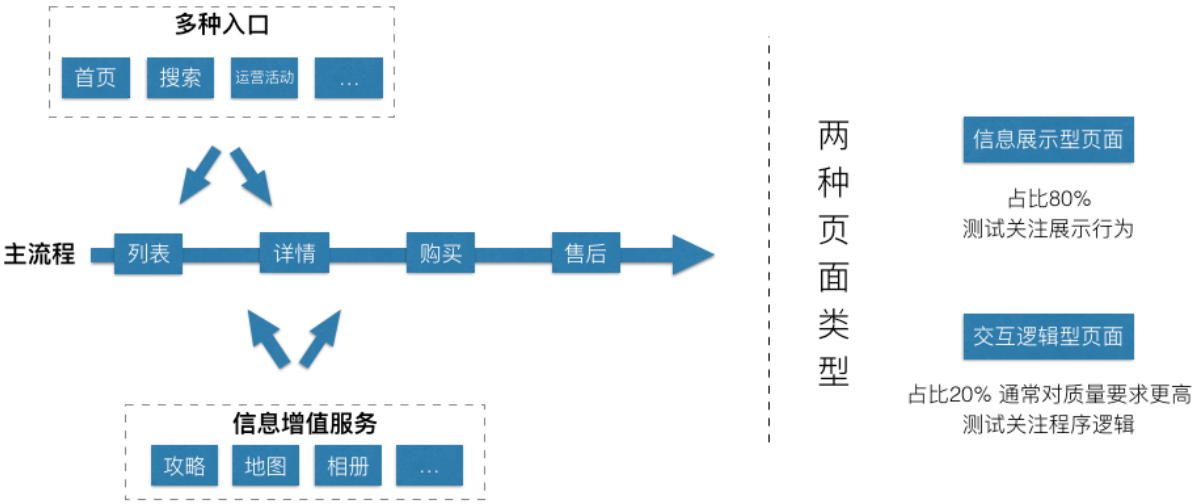
- 测试用例覆盖率（开发了多少自动化测试用例）
- 测试的执行覆盖率（代码行覆盖率、代码分支覆盖率）

明确了上面这些指标，我们在实施自动化测试的时候，就需要有意识的收集这部分的数据，用来评估自动化测试带来的实际价值。

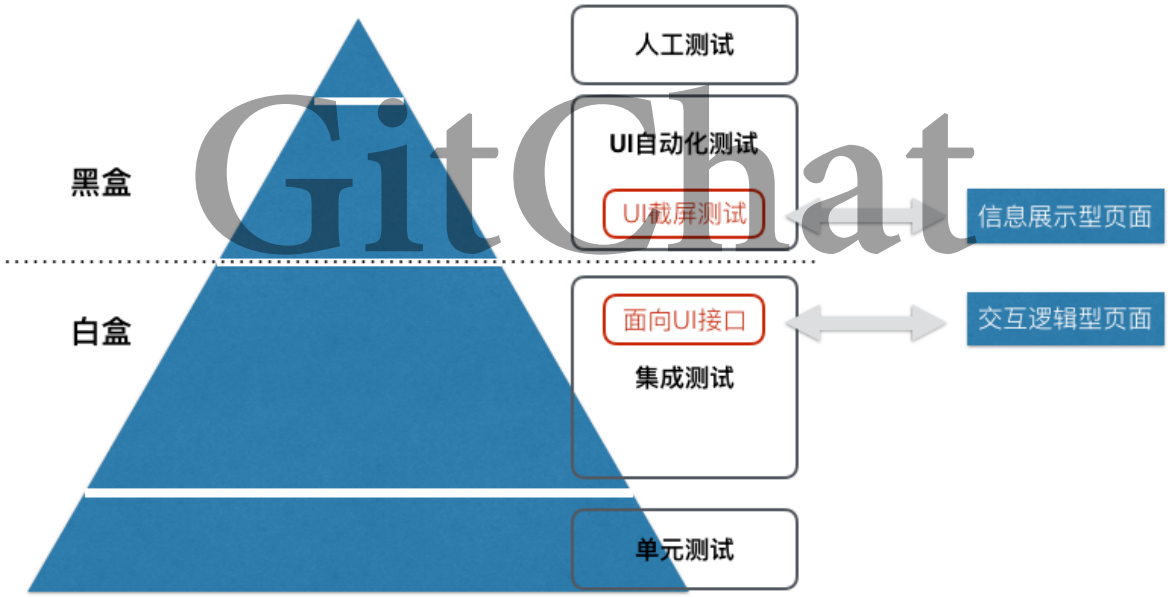
自动化测试的方法选型依据

明确目标和衡量方法之后，我们需要进行方法选型。Google曾经在《*Google* 测试之道》中介绍过，合理测试包括单元测试、集成测试、UI测试（见下图），其占比大概为7：2：1。

然而，从我们设定的收益衡量方法来看，这并非收益最大的做法。占比最高（70%）的单元测试，并不能测出多少Bug。根据我们的经验，以用户视角进行测试，更容易发现Bug。因此，我们需要基于实际的业务形态，选择对投入产出比最高的方案。首先我们看一下酒旅的业务形态，见下图：



其主要包含两种页面类型：信息展示型页面和交互逻辑型页面。两种页面的关注点差异很大：信息展示型通常只是将后端API返回的数据进行“简单”的处理，并最终绘制在屏幕上。其关注重点在于展示是否正确（UI控件的布局，文字截断、折行，等等）。交互逻辑型页面重点关注的是用户操作带来的逻辑处理，需要进行端到端的测试，即用户操作产生了对服务端API的请求。显然，这两种类型页面的测试关注点并不相同，因而技术选型上也会有所差异。我们的技术选型方案如图所示：



对于交互逻辑型页面，我们选择了“面向UI接口”的集成测试方案。这种方案主要解决细粒度的程序逻辑校验，这类页面占比较低（约20%），但是意义重大，一旦出现Bug，很有可能是较严重的故障。因此对这一类页面的测试也显得尤为重要。

对于信息展示型页面，我们选取了UI截屏测试的方案，可以进行像素级别的展示结果校对。截屏测试方案可以覆盖大约80%的客户端页面，同时覆盖成本也较低。

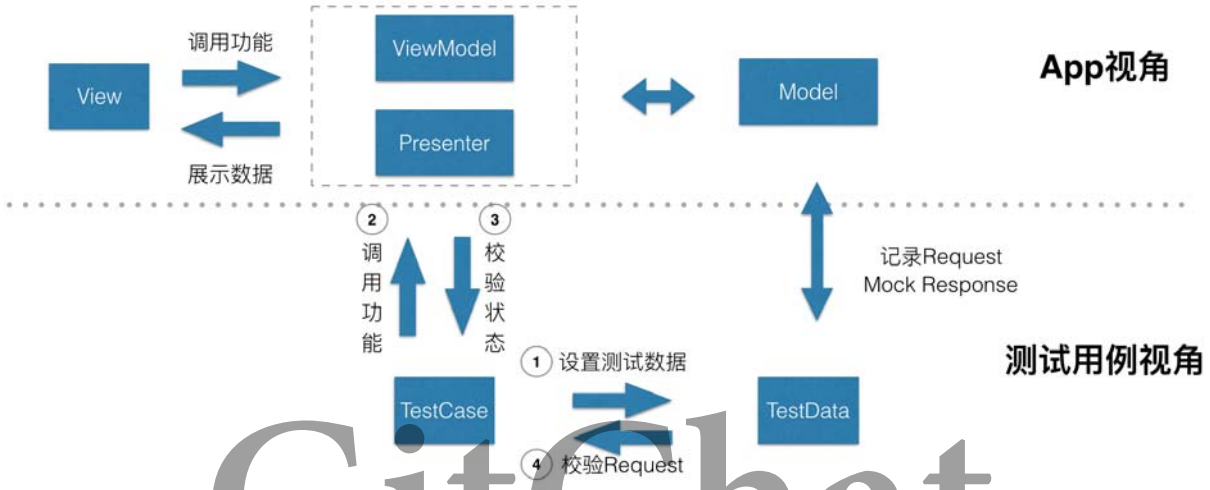
任何一个自动化测试的选型都会遵循以下的步骤：

1. 设置上下文状态（数据打桩、全局初始化）
2. 模拟用户行为
3. 校验执行结果

下面我们就对信息展示型和交互逻辑型页面，使用自动化测试的三部曲的套路，实施测试。

交互逻辑型页面的测试方法

对于交互逻辑型的页面，测试的重点是逻辑细节，我们选取对ViewModel进行测试。这里有一个背景，我们在两年前就开始积极实践MVVM(Model-View-View-Model)和FRP(Fuctional Reactive Programming)。MVVM带给我们最大的好处就在于自动化测试。当然，MVVM只是使得测试变得更容易，但并不是实施自动化测试的必要条件。传统的MVC模式，同样可以进行细粒度的代码逻辑测试。



如图所示，基于MVVM或MVP的设计模式，测试会变得更容易。测试用例模拟视图，访问ViewModel或Presenter，对业务逻辑进行验证，既可以校验应用的状态，也可以校验提交的后台服务器的数据（参数）。

```
//1. 设置测试数据（打桩）
[OHHTTPStubs stubRequestsPassingTest:^BOOL(NSURLRequest *request {
    return [request.URL.path containsString: @"替换URL的路径"];
} withStubResponse:^OHHTTPStubsResponse *(NSURLRequest *request) {
    NSData *data = 要返回的数据
    OHHTTPStubsResponse *fakeResponse = [OHHTTPStubsResponse
responseWithData:data statusCode:200 headers:@{@"Content-Type" :
@"application/json"}];
    return fakeResponse;
}]

//2. 模拟用户行为
LoginViewModel* vm = [[LoginViewModel alloc] init];
//模拟输入用户名、密码
vm.userID = XXXX
vm.password = XXXX

//执行登录动作
[vm.loginCommand execute:nil]
```

```
//3. 校验登录结果

[vm.loginCommand.executionSignals.flatten subscribeNext:^(id x) {
    expect(x).equal(@"LoginSuccess");
    done();
}];
```

信息展示型页面的测试方法

我们选择截屏测试方案对信息展示型页面进行测试。想要使用截屏测试方案有一个前提：就是除掉程序代码的因素，每次截屏都需要有唯一的结果。这就对应了自动化测试三部曲的第一步，“设置上下文状态（数据打桩、全局初始化）”。通常情况，信息展示型页面总是需要展示后台API返回的数据，并且可能展示一些和用户状态相关的内容，比如，用户名、用户设备定位信息（坐标、地图）、系统日期与时间，等等。

因此，在实施截屏测试方案时，第一步需要设置许多状态，比如对后台API的打桩和Mock。这里建议将Mock代码进行一些封装沉淀，因为它们很大概率会被其他Case复用。

第二步，模拟用户操作。信息展示型页面的典型操作通常是加载->展示，和滚动一屏->展示。这里我们选择了KIF框架模拟用户行为，滚屏操作十分容易。

第三步，进行截屏或图片比对。这里我们选用了Facebook开源的框架FBSnapShoot。代码如下：


```
//测试用例继承于FBSnapshotTestCase

//1. 打桩 （与交互逻辑型页面相同，不在赘述）

//2. 获取需要比对的视图

UIView* view = 获取需要比对的视图

//3. 截屏比较

FBSnapshotVerifyView(view, nil);
```

是不是简单的有些不可思议？

未来展望

针对不同的业务类型，我们同时选择UI截屏和UI接口集成测试的方案。使用这两种方案仍都有较大的效率提升空间：

UI截屏方案的**Case**失效率较高。我们需要用工具化、平台化的方法提升Case维护效率。

- 自测场景：集中展示新需求的UI截屏，并且覆盖主流机型的全部分辨率。通过工具批量确认截图的内容是否这确，可以显著提升自测的效率和覆盖率。
- **Case**测试失败场景：当Case测试失败时，集中展示全部执行失败的Case。如果确实有Bug，就点击“报Bug”按钮；否则点击“更新截图”按钮。
- 新设备支持：当出现新的设备，比如iPhone X，我们可以集中运行全部的截屏测试用例，一次性得到所有的截图。这比手工跑一遍人工测试要高效的多

UI接口集成测试开发成本较高。我们选取的业务场景，对质量有更高的要求，因此投入更多的人力实施自动化测试是符合投入产出比的。针对这种场景，我们建议通过改进框架提高复用，提升校验环节的效率。

---

1. 摘自维基百科 ↩