



下载APP



## 12 | 集成测试（一）：一条Happy Path扫天下

2022-04-15 柳胜

《自动化测试高手课》

课程介绍 &gt;



讲述：柳胜

时长 11:09 大小 10.22M



你好，我是柳胜。

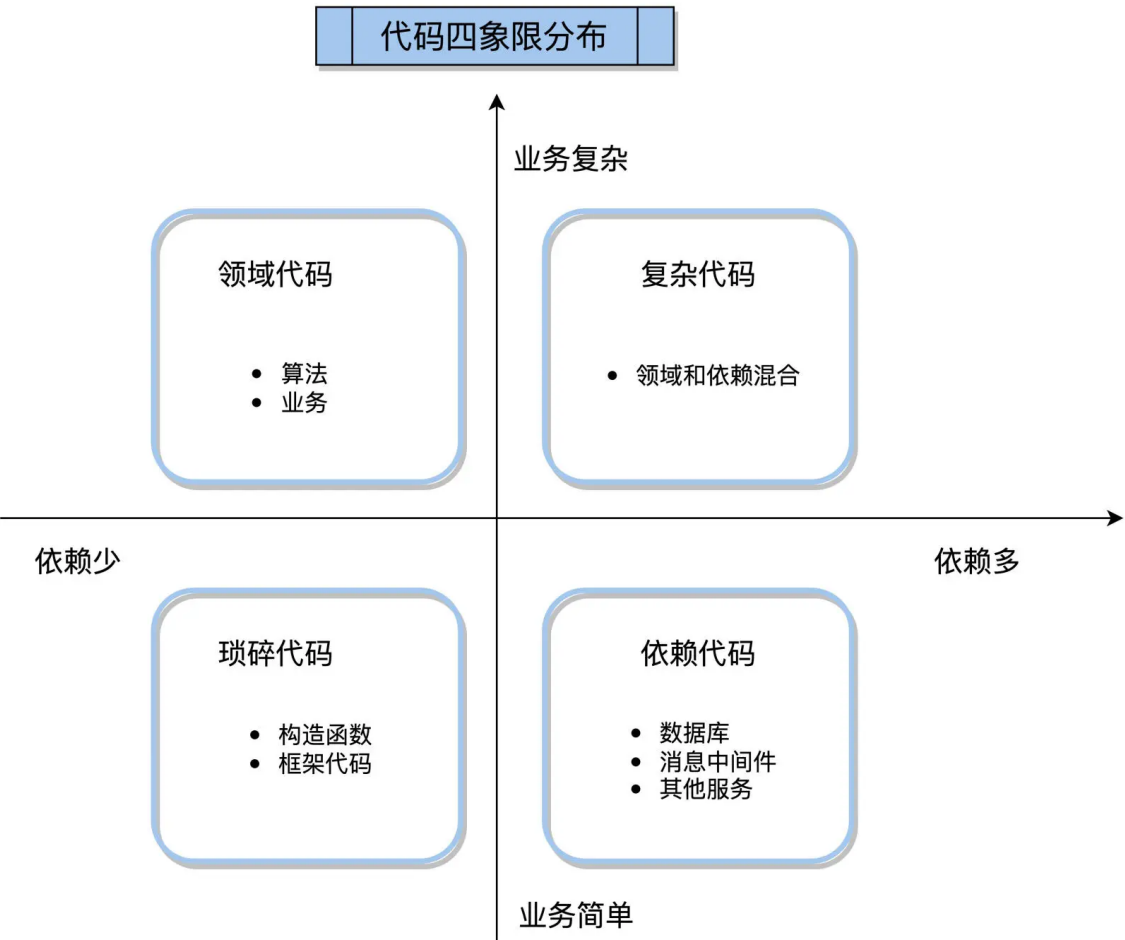
上一讲，我们学习了单元测试，在验证业务逻辑方面，它的优势在于速度又快，阶段又早。既然单元测试看起来是一个完美的自动化测试方案，那为什么还需要集成测试呢？

我在 [第二讲](#) 的 3KU 原则说过，测试需求首先要找 ROI 最高的截面来验证。在金字塔模型里，ROI 最高的就是单元测试，如果无法实现，才回退到 ROI 第二高的截面，一直到 ROI 最低的端到端测试。



那集成测试存在的价值，一定是做得了单元测试层面做不到的事，否则，集成测试这个概念就没必要存在。那这些事具体有哪些呢？你要是能找到这些事，就找到了集成测试省力又见效的窍门。今天咱们就一起寻找这个答案。

上一讲我们介绍了代码四象限法则，不同的代码按照业务复杂性、依赖性和耦合性，可以划分到下面四个象限里。

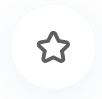


极客时间

那集成测试和单元测试分别应该归到第几象限呢？

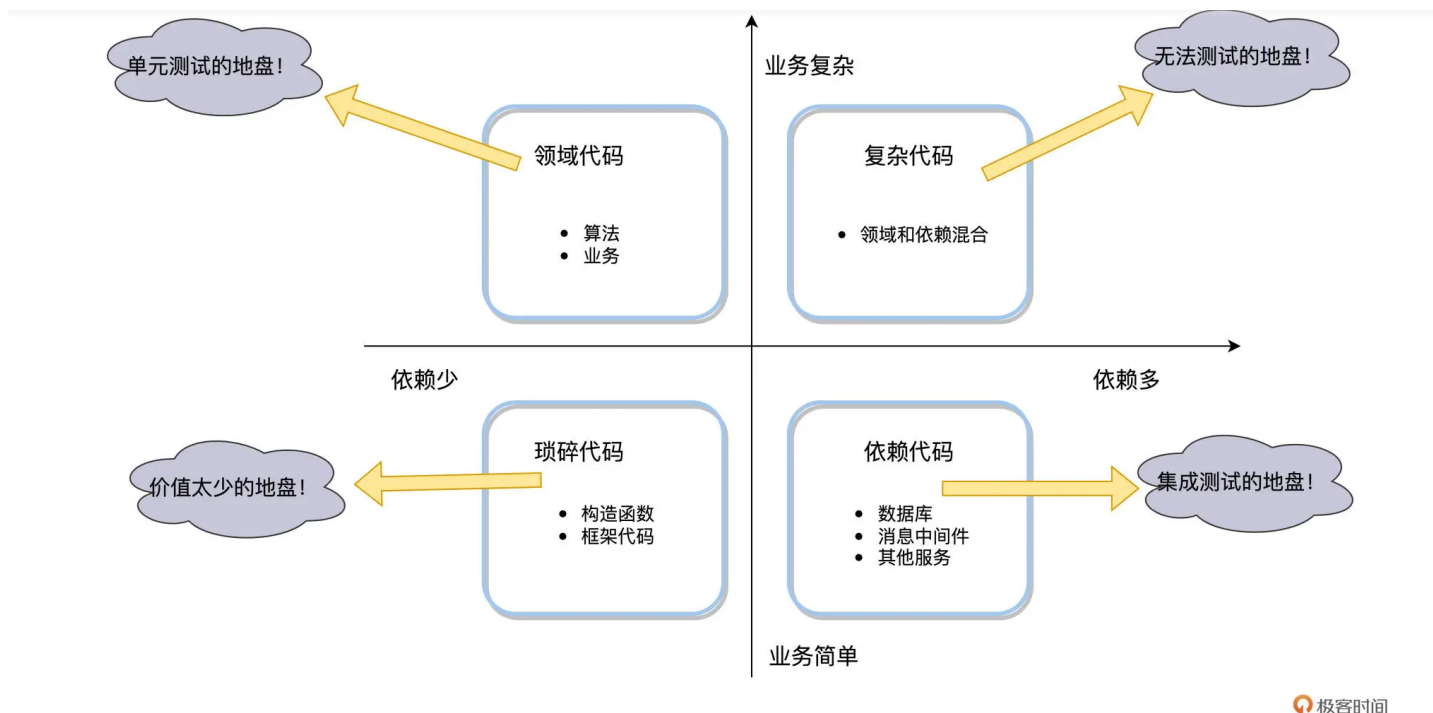
集成测试，顾名思义，是验证本服务代码和其他进程的服务能不能一起配合工作。在上面的四象限里，集成测试的活动领域就在“依赖代码”象限，而单元测试的活动领域是在“领域代码”象限。

我再用图解的方式划分一下地盘，你会看得更清楚。





下载APP



极客时间

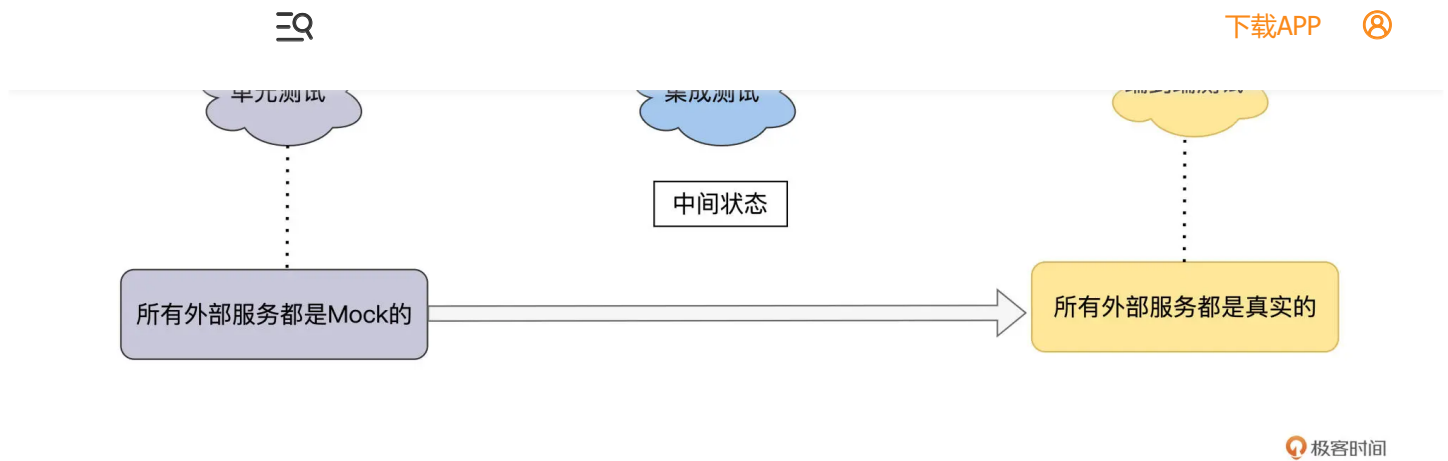
这张图里的信息量很大，展示了单元测试和集成测试的各自战场，我来跟你细说一下。

单元测试掌管领域代码的测试，这些领域代码只是负责数据计算，并不会触及外部依赖。像上一讲的 `changeEmail` 方法，只是计算出一个新的餐馆数目，单元测试只需要验证这个计算逻辑是否正确就好了。

那什么是单元测试测不了的呢？那就是依赖代码。在 FoodCome 的代码设计里，这些外部的依赖管理交给一个独立的 `Controller Class` 去做，它负责读写数据库、发送消息等等。这块就是集成测试的领域。

看到这里，你脑袋里可能会冒出这样一个问题：不对呀！单元测试也可以测试外部依赖，我们在前面讲过可以 Mock 外部依赖，如果我把 `Database`、`MessageBus` 都 Mock 了，那不就也可以做单元测试了么？

你能想到这一层，说明你已经关注概念背后真正的事情了。是的，如果所有的外部服务都 Mock 了，集成测试就变成了单元测试，往另外一个方向，如果所有的外部服务都是真的，集成测试又变成了端到端的测试。**集成测试就是处在单元测试和端到端测试中间的一个状态。**



在这里，我们要关注 **Mock 和 Real 的优劣势**，**集成测试怎么能做得更聪明一些，用最少的工作量，获得最大的测试效果**。下面我们就展开来说一说。

## 集成测试测什么？

相比单元测试，集成测试有 2 个特点。

第一，集成测试运行速度慢，这个时间主要花在 2 个地方，第一个是准备集成测试环境的时间，你要先把依赖的外部服务启动起来，让环境处在一个健康状态；第二个是运行集成测试的时间，因为集成测试不像单元测试是进程内工作，它是跨进程通讯，除了计算时间，还要加上网络通讯时间等等。

第二，执行集成测试，要运行的代码量比单元测试要多。因为它走过的路径更长，从网络请求，到处理请求，再到网络返回结果，中间需要经历过  $n$  个代码单元，还有框架代码，库代码等等。

这两个特征告诉我们，集成测试是有比较大的成本的，并且它测试的代码逻辑和单元测试是有重叠的。

本着追求整体最大 ROI 效益的目标，集成测试和单元测试需要协同作战，保持一个平衡，这个平衡的原则是：

1. 在单元测试阶段验证尽可能多的业务逻辑，这样能让集成测试关注在外部依赖上。

2. 集成测试至少覆盖一条长路径案例，叫“Happy Path”。



下载APP



Happy Path 是指一个业务逻辑的测试案例，是尽可能少的外部依赖服务。比如，一个案例，同时走了 Database 和 MessageBus。

针对 [上一讲](#) 提到的用户修改邮箱功能，我们有几个案例：

1. 修改邮箱名从 a@foodcome.com 到 b@foodcome.com
2. 修改邮箱名从 a@example.com 到 a@foodcome.com
3. 修改邮箱名从 a@example.com 到 b@example.com

哪个案例是 Happy Path 呢？再回头看一下代码：

复制代码

```
1 public class UserController
2 {
3     .....
4     public void ChangeEmail(int userId, string newEmail)
5     {
6         .....
7         user.ChangeEmail(newEmail, restaurant);
8         _database.SaveUser(user);
9         //如果restaurant数量有变化，就写数据库，发送通知信息
10        if(restaurant.numberChanged()){
11            _database.SaveRestaurant(restaurant);
12            _messageBus.SendEmailChangedMessage(userId, newEmail);
13        }
14    }
15 }
```

我们不难发现案例 2 符合 Happy Path，因为它触发了多次与 2 个外部依赖的交互，更新了 Database 的用户信息和餐馆信息，还触发了消息总线发送一条通知出去。

你可能还想到一个疑问，如果我们找不到一个能触发全部外部依赖交互点的 Happy Path，那怎么办？很简单，那就再加一条 Happy Path。



## 集成测试用 Mock 还是 Real 测试？



优劣，都有适用的场景（可以回看 [🔗第十讲](#)）。

今天我们详细说说，选择 Mock 还是 Real 的方法。

首先要看外部依赖的特征，我把它划分成 2 种类型。

1. 完全可控依赖

2. 不可控依赖

什么是完全可控依赖呢？这个外部的服务被你的应用独享，你也能够控制它的开发和运维，那这个服务就是完全可控依赖的。一个典型的例子，就是数据库，在微服务模式下，每一个服务独享一个自己的数据库 Schema。

那什么又是不可控依赖？与可控依赖相反，这个外部的服务不止你的应用调用，大家都得遵守一个协议或规范，和这个公共的外部服务交互。典型的例子，就是外部的支付系统，SMTP 邮件通知服务等等。

与这两种类型相对应的 Mock 策略如下：

Mock策略\依赖类型	完全可控依赖	不可控依赖
Mock	不推荐	推荐
Real	推荐	不推荐



为什么是这样的？完全可控依赖的服务，虽然是在你的应用之外的一个进程，但你可以跟它的交互当作是你开发的内部实现。你可以升级数据库版本、修改表格结构、增加数据库函数，只要跟着应用的代码一起修改即可。

而不可控依赖服务就不一样了，它是公共的，你控制不了它，而且你跟它的交互还要遵守一个规范的契约。在这种情况下，做 Mock 就划算了，原因有二：**第一，基于契约的 Mock 的维护成本比较低；第二，使用 Mock 可以保证你的应用持续重构，向后兼容。**

分析到这，我们就能梳理出 FoodCome 的 Mock 策略了。


外部依赖服务	类型	Mock策略
数据库	完全可控	真实实例
支付服务	不可控	Mock
通知服务	不可控	Mock
物流服务	不可控	Mock
消息总线	不可控	Mock



## 集成测试的实现

找出了 Happy Path，也定了 Mock 策略后，就可以动手写代码了。

根据 2 号案例，我们来创建一个测试方法，方法名为 `change_email_from_example_to_foodcome`：

 复制代码

```
1 [Fact]
2 public void Changing_email_from_example_to_foodcome()
3 {
4     // Arrange
5     var db = new Database(connectionString);
6     User user = CreateUser(
7         "a@example.com", UserType.customer, db);
8     var messageBusMock = new Mock<IMessageBus>();
9     var sut = new UserController(db, messageBusMock.Object);
```







```
13     Assert.Equal("OK", result);
14     // 校验数据库里字段
15     object[] userData = db.GetUserById(user.UserId);
16     User userFromDb = UserFactory.Create(userData);
17     Assert.Equal("b@foodcome.com", userFromDb.Email);
18     Assert.Equal(UserType.Restaurant, userFromDb.Type);
19     messageBusMock.Verify(
20         x => x.SendEmailChangedMessage(
21             user.UserId, "b@foodcome.com"),
22         Times.Once);
23 }
```

上面的代码完成了以下步骤，我特意分点列出来，方便你看清楚每一步。

1. 创建真实的数据库连接对象；
2. 创建 MessageBus 的 Mock 对象；
3. 把 2 个依赖注入到被测 UserController class 里，调用 changeEmail 方法；
4. 检验数据库里的 User 状态；
5. 检验 Mock 的 MessageBus 里的消息。

## 小结

今天我们学习了和外部服务的集成测试的方法，在动手之前，我们要想明白测什么，用什么测，Mock 还是 Real。

测什么，怎么测，这就是集成测试方案要回答的问题，而且，这个方案的制定遵循 3KU 原则，也就是尽量不做重复的事，把精力和时间花在有价值的地方。

单元测试需要做好业务逻辑的验证，集成测试主要是测试与外部依赖的集成，集成又有两种策略，采用 Mock 还是 Real 真实的依赖，应该遵循**能 Real 就 Real 的原则，不能 Real 的再采用 Mock**，如果一股脑 Mock 所有依赖，你会发现集成测试没测到什么有用的逻辑，都在 Mock 上，而真正集成时还是会遇到问题。





下载APP



原则，尽量把开发和维护 Mock 的工作量花在最有价值的外部依赖上。

## 思考题

在实际工作中，你有多个测试案例，怎么找出那条 Happy Path？除了看代码，还有别的方法么？

欢迎你在留言区跟我交流互动，也推荐你把这讲内容分享给更多同事、朋友。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 0

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 单元测试（二）：四象限法让你的单测火力全开

下一篇 13 | 集成测试（二）：携手开发，集测省力又省心

## 精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

