

测试金字塔实战

2018年10月9日 by [Ham Vocke](https://insights.thoughtworks.cn/author/ham-vocke/) — [Leave a Comment](https://insights.thoughtworks.cn/practical-test-pyramid/#respond)

“测试金字塔”是一个比喻，它告诉我们要把软件测试按照不同粒度来分组。它也告诉我们每个组应该有多少测试。虽然测试金字塔的概念已经存在了一段时间，但一些团队仍然很难正确将它投入实践。本文重新审视“测试金字塔”最初的概念，并展示如何将其付诸实践。本文将告诉你应该在金字塔的不同层次上寻找何种类型的测试，如何实现这些测试。

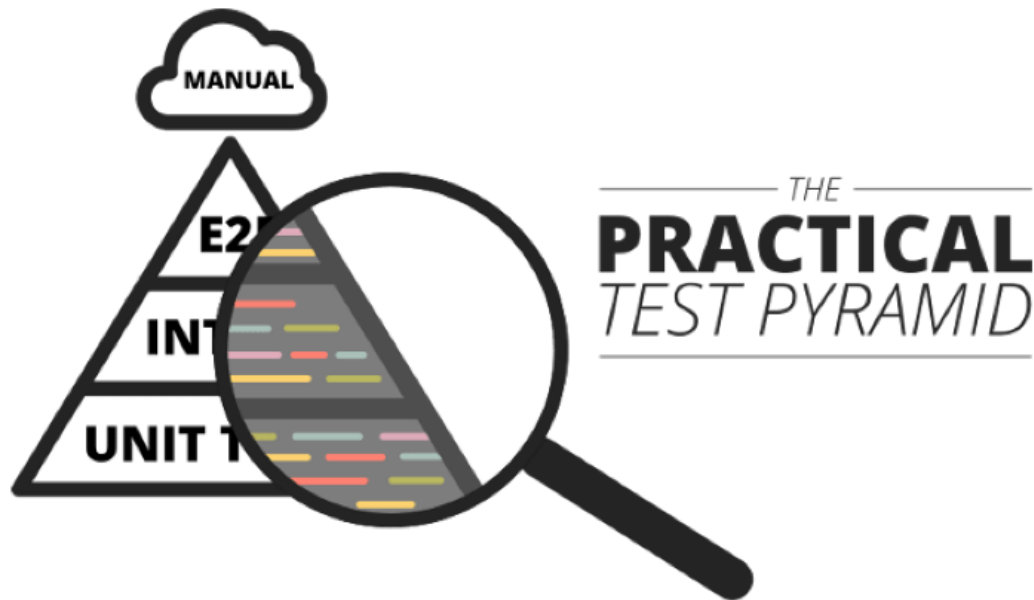
2018 年 2 月 26 日 作者：Ham Vocke

Ham 是德国 ThoughtWorks 的一名软件开发和咨询师。由于厌倦了在凌晨 3 点手动部署软件，他开始持续交付实践，加紧自动化步伐，并着手帮助团队高效可靠地交付高质量软件。这样他就可以把省出来的时间用在别的有趣的事情上了。

目录

- 测试自动化的重要性
- 测试金字塔
- 我们用到的工具和库
- 应用例子
 - 功能
 - 整体架构
 - 内部架构
- 单元测试
 - 什么是单元？
 - 社交和独处
 - 模拟和打桩
 - 测试什么？
 - 测试架构
 - 实现一个单元测试
- 集成测试
 - 数据库集成
 - REST API 集成
 - 几个独立服务的集成
 - JSON 的解析和撰写
- 契约测试
 - 消费者测试(我们团队)
 - 提供者测试(其他团队)
 - 提供者测试(我们团队)
- UI 测试
- 端到端测试
 - 用户界面端到端测试
 - REST API 端到端测试
- 验收测试 – 你的功能工作正常吗？
- 探索测试
- 测试术语误解
- 把测试放到部署流水线

- 避免测试重复
- 整洁测试代码
- 结论



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/1.png>)

准备上生产环境的软件在上生产之前需要进行测试。随着软件开发行业的成熟，软件测试方法也日趋成熟。开发团队正在逐渐自动化大部分的测试，以此取代大量测试人员手工测试。通过自动化测试，开发团队可以分分钟就知道他们的软件是否被破坏，而不是后知后觉几天后才知道。

自动化测试极大地缩短了反馈周期，这与敏捷开发实践、持续集成、DevOps 文化等是一脉相承的。拥有高效的软件测试方法，可以让你的团队快速而自信地前行。

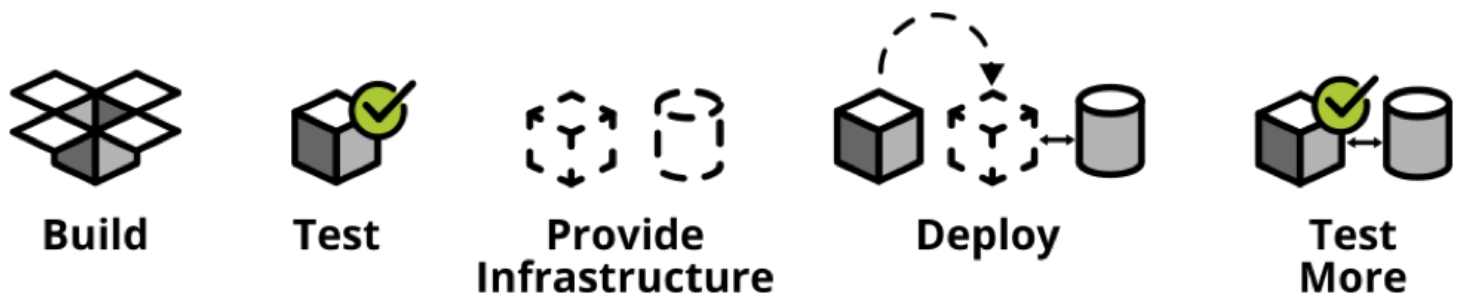
本文将探讨一个具备高响应力的、可靠并且可维护的测试组合应该如何构建，这与你具体构建的是一个微服务架构、移动应用程序或者物联网生态系统都无关。此外，我们还将详细介绍如何写出高效且可读的自动化测试。

(测试) 自动化的重要性

软件已经成为我们日常生活中的一个重要组成部分。早期它仅仅用于提高企业的效率，但如今它的作用远不止如此。如今许多公司都想方设法成为一流的数字化公司。作为用户，我们每天使用的软件越来越多。创新的车轮正加速向前滚动。

如果你想跟上时代的步伐，你必须研究如何在不牺牲质量的情况下更快地交付你的软件。持续交付——一种高度自动化的、确保你可以随时将软件发布到生产环境中的实践——正能帮你达到这个目的。它通过构建流水线自动测试你的软件，自动将其部署到测试和生产环境中。

软件的数量正以前所未有的速度增长，手动进行构建、测试和部署，很快就会变得不可能——除非你想把所有的时间都用来进行手动重复的工作，而不是用来开发可工作的软件。将一切自动化，从构建到测试，从部署到基础架构，这是你唯一的出路。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/2.png>)

(使用构建流水线来自动并可靠地将你的软件部署到生产环境)

传统的软件测试过于依赖手工操作：首先将应用程序部署到测试环境，然后执行一些黑盒测试，例如，通过点击用户界面来查看一切是否工作如常。通常这些测试将由文档指定，以确保测试人员每次测试的内容是一致的。

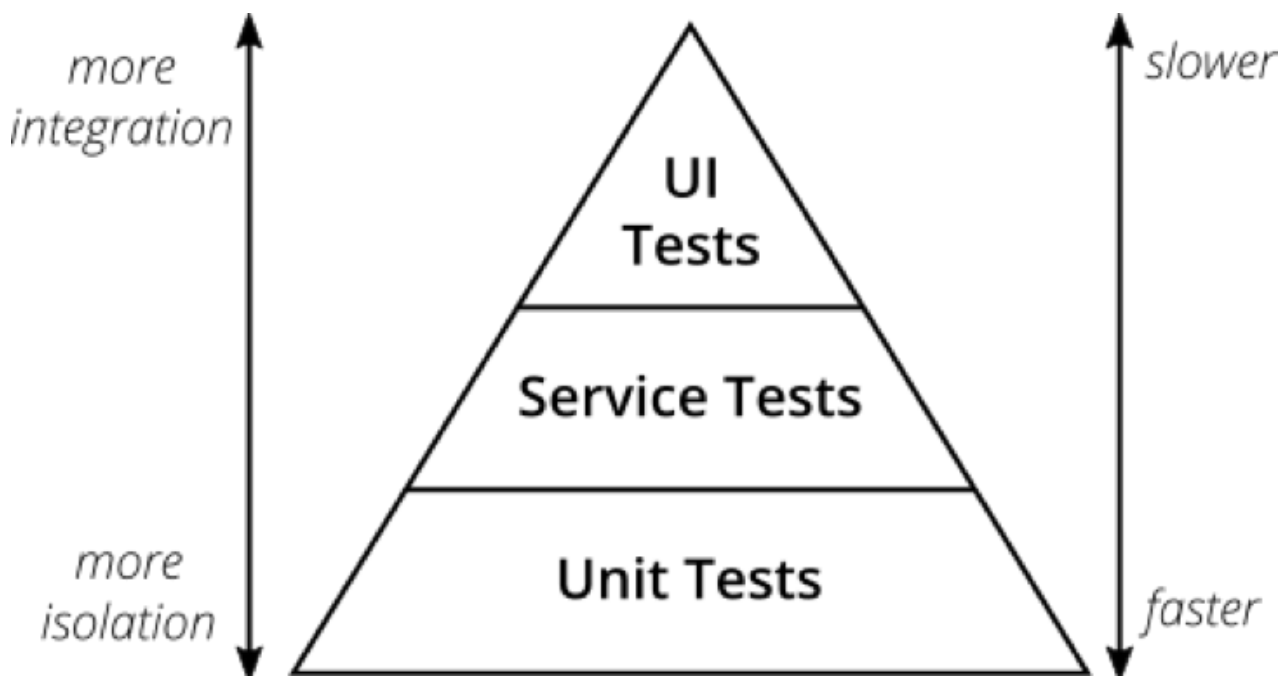
很明显，手动测试所有更改非常耗时、重复而且繁琐。重复很无趣，无趣就容易犯错，这样子还没测到这周工作结束你就会想找下一份工作。

幸运的是，重复性劳动还是有药可治的：自动化。

自动化繁琐重复的测试将给软件开发人员的生活带来重大改变。自动化这些测试后，你就不需要再一味遵循测试文档点点点以确保软件是否仍正常工作。自动化这些测试，你可以充满自信地修改你的代码。如果你曾试过在没有适当自动化测试的情况下进行大规模重构，那你应该知道这种体验多么恐怖。你怎么知道你是否意外地破坏了某些功能呢？显然，你需要将所有的测试用例手动点一遍。不过老实说，你真的享受这个过程吗？你想象一下，如果你对代码做了大规模改动后惬意地喝了一口咖啡，喝完咖啡后就能马上得知你的改动有没有破坏原有功能。这样的开发体验是不是听起来就让人舒服多了？

测试金字塔

如果你真的想为你的软件构建自动化测试，你必须知道一个关键的概念：测试金字塔。Mike Cohn 在他的著作《Succeeding with Agile》一书中提出了这个概念。这个比喻非常形象，它让你一眼就知道测试是需要分层的。它还告诉你每一层需要写多少测试。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/3.png>)

(测试金字塔)

根据 Mike Cohn 的测试金字塔，你的测试组合应该由以下三层组成（自下往上分别是）：

- 单元测试
- 服务测试
- 用户界面测试

不幸的是，如果你仔细思考就会发现，测试金字塔的概念有点太短了。有人认为，Mike Cohn 的测试金字塔里的命名或某些概念不是最理想的。我也同意这一点。从当今的角度来看，测试金字塔似乎过于简单了，因此可能会产生误导。

然而，由于其简洁性，在建立你自己的测试组合时，测试金字塔本身是一条很好的经验法则。你最好记住 Cohn 测试金字塔中提到的两件事：

- 编写不同粒度的测试
- 层次越高，你写的测试应该越少

为了维持金字塔形状，一个健康、快速、可维护的测试组合应该是这样的：写许多小而快的单元测试。适当写一些更粗粒度的测试，写很少高层次的端到端测试。注意不要让你的测试变成冰淇淋 (<https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern>) 那样子，这对维护来说将是一个噩梦，并且跑一遍也需要太多时间。

不要太拘泥于 Cohn 测试金字塔中各层次的名字。事实上，它们可能相当具有误导性：服务测试是一个难以掌握的术语（Cohn 本人说他观察到很多开发人员完全忽略了这一层 (<https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid>)）。在单页应用框架（如 react，angular，ember.js 等）的时代，UI 测试显然不必位于金字塔的最高层，你完全能够用这些框架对 UI 进行单元测试。

考虑到原始名称的缺点，只要在你的代码库和团队讨论中达成一致，你完全可以为测试层次提供其他名称。

我们将使用的工具和库

- JUnit (<https://junit.org/junit5/>): 测试执行库
- Mockito (<https://site.mockito.org/>): 模拟依赖
- Wiremock (<http://wiremock.org/>): 为外部服务打桩
- Pact (<https://docs.pact.io/>): 用于编写消费者驱动的契约测试
- Selenium (<https://docs.seleniumhq.org/>): 用于编写用户界面驱动的端到端测试
- REST-assured (<https://github.com/rest-assured/rest-assured>): 用于编写 REST API 驱动的端到端测试

示例应用

我已经写好了一个简单的微服务应用 (<https://github.com/hamvocke/spring-testing>)，其中涵盖了测试金字塔各种层次的测试。

示例应用体现了一个典型的微服务的特点。它提供了一个 REST 接口，与数据库进行通信并从第三方 REST 服务中获取信息。它是使用 Spring Boot 实现的，即使你之前从未使用过 Spring Boot，它也简单到应该让你很容易理解。

请下载 Github 上的代码。Readme 里写了你在计算机上运行应用程序及其自动化测试所需的说明。

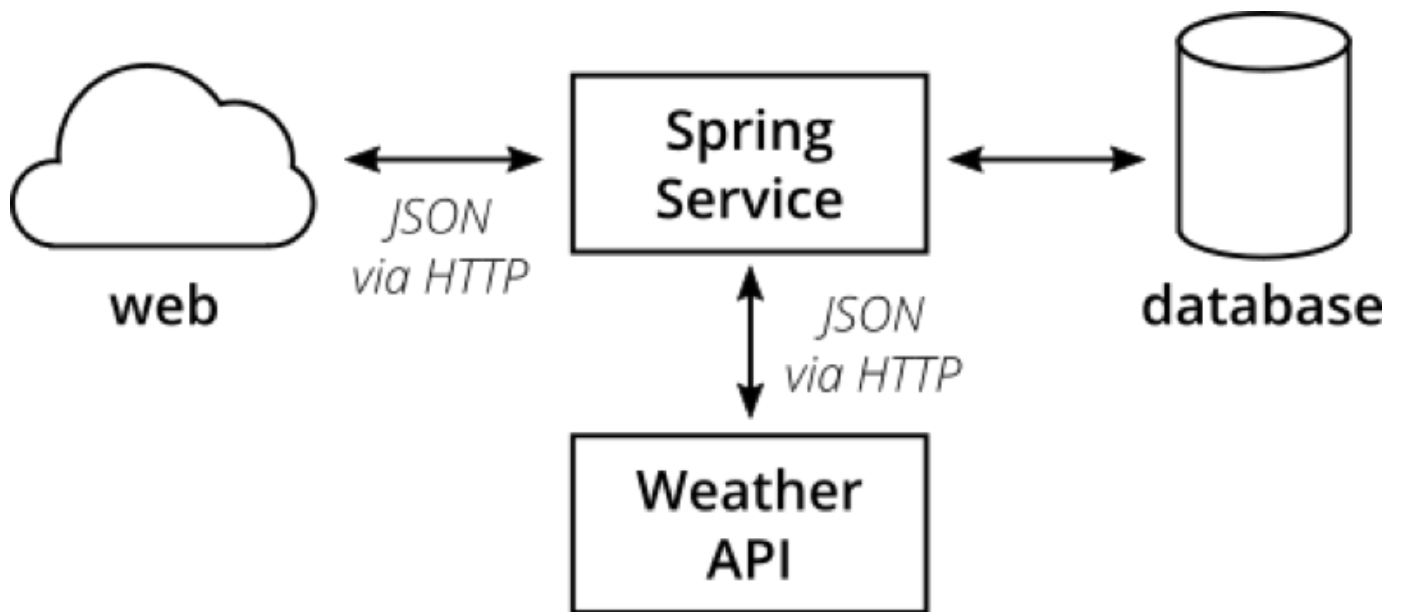
功能

应用的功能十分简单。它提供了三个 REST 接口：

- GET /hello 总是返回"Hello World"
- GET /hello/{lastname} 根据 lastname 来查询人，如果查到了结果将返回"Hello {Firstname} {Lastname}"
- GET /weather 返回现在德国汉堡的天气情况

高层架构

从高层次来看，这个系统的结构是这样：



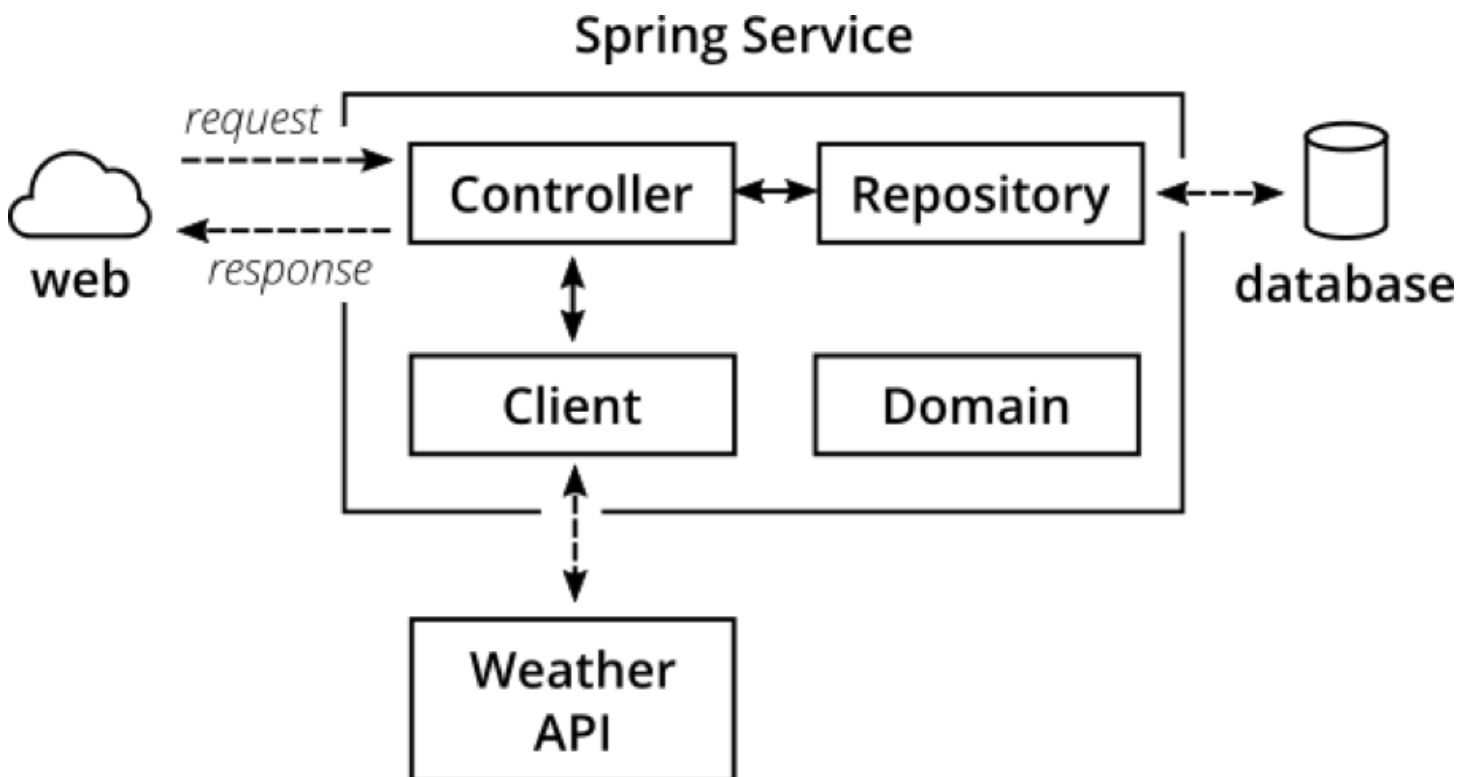
(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/4.png>).

(我们微服务系统的高层架构)

我们的微服务提供了一个可以通过 HTTP 调用的 REST 接口。对于某些接口，服务将从数据库获取信息。在其他情况下，服务将通过 HTTP 调用外部天气 API (<https://darksky.net/forecast/40.7127,-74.0059/us12/en>) 来获取并显示当前天气状况。

内部架构

在内部，Spring Service 有一个典型的 Spring 架构：



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/5.png>).

(我们微服务的内部架构)

- Controller 提供 REST 接口，处理 HTTP 请求和响应

- Repository 和数据库打交道，关注数据在持久化存储里的读写操作
- Client 和别的 API 交互，在我们的应用里它会通过 HTTPS 从 darksky.net 获取天气情况
- Domain 这是我们的领域模型 (https://en.wikipedia.org/wiki/Domain_model)，它包含了领域逻辑（相对来说，在我们的示例中不甚重要）

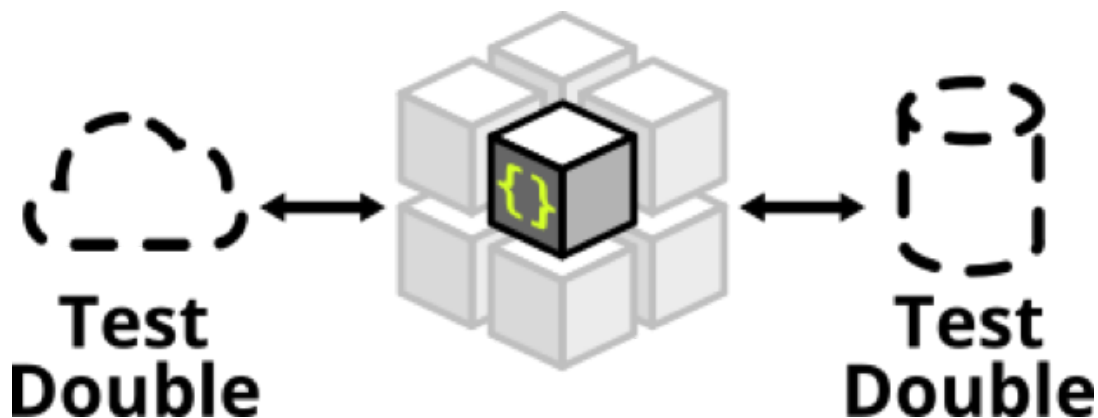
有经验的 Spring 开发人员可能会注意到这里缺失了一个常用的层次：受 [Domain-Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design) (https://en.wikipedia.org/wiki/Domain-driven_design) 的启发，很多开发人员通常会构建一个由服务类组成的服务层。我决定不在这个应用中包含服务层。原因之一是我们的应用程序很简单，服务层只会成为一个不必要的中间层。另一个是我认为人们过度使用服务层。我经常遇到在服务类中写了全部业务逻辑的代码库。领域模型仅仅成为数据层，而不是行为（贫血域模型 (https://en.wikipedia.org/wiki/Anemic_domain_model)）。对于每一个稍有复杂度的应用来说，这浪费了很多让代码保持结构良好且易于测试的优秀方案，并且没能充分利用面向对象的威力。

我们的 repositories 非常简单，它提供简单的 CRUD 功能。为了保持代码简单，我使用了 Spring Data。Spring Data 为我们提供了一个简单而通用的 CRUD 实现，我们可以直接使用而不需再造轮子。它还负责为我们的测试启动一个内存数据库，而不是像生产中一样使用真正的 PostgreSQL 数据库。

看看代码库，熟悉一下内部结构。这将有助于我们的下一步：测试我们的应用！

单元测试

单元测试将成为你测试组合的基石。你的单元测试保证了代码库里的某个单元（被测试的主体）能按照预期那样工作。单元测试在你的测试组合里测试的范围是最窄的。它的数量在测试组合中应该远远多于其他类型的测试。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/6.png>)

（一个用测试替身隔绝了外部依赖的典型单元测试）

一个单元指的是什么？

如果你去问三个人同样的问题：“单元”在单元测试的上下文中意味着什么，你很可能会获得四种非常相似的答案。某种程度上讲，对“单元”的定义取决于你自己，因此这个问题没有标准答案。

如果你正在使用函数式语言，一个单元最有可能指的是一个函数。你的单元测试将使用不同的参数调用这个函数，并断言它返回了期待的结果。在面向对象语言里，下至一个方法，上至一个类都可以是一个单元（从一个单一的方法到一整个的类都可以是一个单元）。

群居和独居

一些人主张，应该将被测试主体下的所有合作者（比如在测试里被你的类调用的其他类）都使用模拟或者桩替换掉，这样可以建立完美的隔离，避免副作用和复杂的测试准备。而有些人主张，只有那些执行起来很慢或者有较大副作用的合作者（比如读写数据库或者发送网络请求的类）才应该被模拟或者打桩替代。

偶尔 (<https://martinfowler.com/bliki/UnitTest.html>) 有人会把用桩隔离所有依赖的测试称为独居单元测试，把和依赖有交互的测试成为群居单元测试（Jay Fields 的《[Working Effectively with Unit Tests](https://leanpub.com/wewut)》(<https://leanpub.com/wewut>) 这本书里创造了这些概念）。如果有空你可以继续深究下去，读一读不同思想流派各自的利弊 (<https://martinfowler.com/articles/mocksArentStubs.html>) 在哪。

说到底，决定采用群居方式还是独居方式的单元测试其实并不重要。写自动化测试才是重要的。就我自己而言，我发现我自己经常两种方式都用。如果使用真正的合作者很麻烦，我就会用模拟对象或者桩。如果我觉得引用真正的合作者能让我对测试更有信心，我会仅仅打桩替代掉 service 最外层的依赖。

模拟和打桩(这里以及下文的桩都指 stub)

模拟对象和桩是两种不一样的测试替身（测试替身还不止这两种）。很多人会混用模拟对象和桩这两个概念。我认为，准确的用词会好一点，并且最好能将它们各自的特性谙熟于心。你可以使用测试替身来替换掉真实的对象，给它一个可以更方便测试的实现。

换句话说，这意味着你是用一个假的实现来代替真的那个（例如，一个类，一个模块或者一个函数）。这个假的实现外表和行为和真的很像（都能响应同样的方法调用），只不过真实的响应内容是你单元测试开始前就定义好的。

并不是在单元测试时我们才使用测试替身。还有很多精妙的测试替身能以非常可控的方式来模拟整个系统的功能。然而，在单元测试里使用模拟对象和桩的概率会更高（取决于你是喜欢群居风格还是独居风格的开发者），这主要是因为现代语言和库使得构建模拟对象和桩变得更加简单了。

不管你的技术选型是怎麼样的，一般来说，编程语言的标准库或一些比较有名的三方库都会提供一些优雅的方式来帮你构建 mocks。即使需要自己编写 mock 对象，也只是写一个假类/模块/函数的事，只需要让它与真实的合作者有相同的签名，并设置到你的测试中去即可。

你的单元测试跑起来应该非常快。在一般的机器上跑完数千个单元测试应该只需要几分钟。为了得到快速的单元测试，你应该独立地测试代码库的每一小块，并避免进行真实的数据库操作、文件系统操作，或者发送真实的 HTTP 请求（使用模拟对象和桩来隔离这一部分）。

一旦你掌握了写单元测试的诀窍，你写起来就能越来越顺畅。打桩隔离掉外部依赖，准备一些输入数据，调用被测试的主体，然后检查返回值是不是你所期待的。看看测试驱动开发，让单元测试指引你的开发；如果使用得当，[测试驱动开发](https://en.wikipedia.org/wiki/Test-driven_development) (https://en.wikipedia.org/wiki/Test-driven_development) 将帮你进入一个非常顺畅的工作流，它能帮你创造出一个良好且可维护的设计，顺便还能送你一套全面且自动化的测试。当然，测试驱动开发并不是银弹。但是建议你尝试一下，看看它是否适合你。

应该测试什么？

单元测试有个好处，就是你可以为所有的产品代码类写单元测试，而不需要管它们的功能如何，或者它们在内部结构中属于哪个层次。你可以对 controller 进行单元测试，也可以用同样的方式对 repository、领域类或者文件读写类进行单元测试。良好的开端，从坚持一个实现类就有一个测试类的法则开始。

一个单元测试类至少应该测试这个类的公共接口。私有方法不能直接测试的原因是你不能从测试类直接调用它们。受保护的和包私有的方法可以被测试类直接调用（如果测试类和生产代码类的包结构是一样的），但是测试这些方法可能就太过了。

编写单元测试有一条细则：它们应该保证你代码所有的路径都被测试到（包括正常路径和边缘路径）。同时它们不应该和代码的实现有太紧密的耦合。

为什么这样说呢？

测试如果与产品代码耦合太紧，很快就会令人讨厌。当你重构代码时（快速回顾一下：重构意味着改变代码的内部结构而不改变其对外的行为）你的单元测试就会挂掉。

这样的话你就损失了单元测试的一大好处：充当代码变更的保护网。你很快就会厌烦这些愚蠢的测试，而不会感到它能带来好处，因为你每次重构测试就会挂掉，带来更多的工作量。不过说起来这些愚蠢的测试又是谁把它写成这样的呢？

那么正确的做法是什么？是不要在你的单元测试里耦合实现代码的内部结构。要测试可观测的行为。你应该这样思考：

如果我的输入是 x 和 y，输出会是 z 吗？

而不是这样：

如果我的输入是 x 和 y，那么这个方法会先调用 A 类，然后调用 B 类，接着输出 A 类和 B 类返回值相加的结果吗？

私有方法应该被视为实现细节。这就是为什么你不应该有去测试他们的冲动。我经常听单元测试（或者 TDD）的反对者说，编写单元测试是无意义的工作，因为为了获得一个高的测试覆盖率，你必须测试所有的方法。他们经常引用这样的场景：一个过于激昂的团队领导强硬地让他们为 getter、setter 及其他所有琐碎的代码施加测试，以达到 100% 的测试覆盖率。

这就大错特错啦。

确实你应该测试公共接口。但是更重要的是，不要去测试微不足道的代码。别担心，[Kent Beck 说这样是 OK 的](https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/) (<https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/>)。你不会因为测试 getter，setter 抑或是其他简单的实现（比如没有任何条件逻辑的实现）而得到任何价值。把时间省出来，你就能多参加一个会了，万岁！

测试结构

一个好的测试结构（不局限于单元测试）是这样的：

1. 准备测试数据
2. 调用被测方法
3. 断言返回的是你期待的结果

这里有个口诀可以帮你记住这种结构：“Arrange，Act，Assert (<https://xp123.com/articles/3a-arrange-act-assert/>)”。另一个口诀则是从 BDD 获取的灵感。就是“given”，“when”，“then” (<https://martinfowler.com/bliki/GivenWhenThen.html>)三件套，given 说的是准备数据，when 指的是调用方法，then 则是断言。

这种模式也可以应用于其他更高层次的测试。在任何情况下，它们都能让你的测试保持一致，易于阅读。除此之外，使用这种结构写出来的测试，往往更简短，更具表达力。

实现一个单元测试

知道了测什么、如何组织单元测试后，我们终于可以看一个真正的例子了。让我们来看一个简化版的 ExampleController 类：

```
@RestController
public class ExampleController {

    private final PersonRepository personRepo;

    @Autowired
    public ExampleController(final PersonRepository personRepo) {
        this.personRepo = personRepo;
    }

    @GetMapping("/hello/{lastName}")
    public String hello(@PathVariable final String lastName) {
        Optional<Person> foundPerson = personRepo.findByLastName(lastName);

        return foundPerson
            .map(person -> String.format("Hello %s %s!",
                person.getFirstName(),
                person.getLastName()))
            .orElse(String.format("Who is this '%s' you're talking about?",
                lastName));
    }
}
```

一个针对hello(lastname)方法的单元测试可能是这样的：


```
public class ExampleControllerTest {

    private ExampleController subject;

    @Mock
    private PersonRepository personRepo;

    @Before
    public void setUp() throws Exception {
        initMocks(this);
        subject = new ExampleController(personRepo);
    }

    @Test
    public void shouldReturnFullNameOfAPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
        given(personRepo.findByLastName("Pan"))
            .willReturn(Optional.of(peter));

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Hello Peter Pan!"));
    }

    @Test
    public void shouldTellIfPersonIsUnknown() throws Exception {
        given(personRepo.findByLastName(anyString()))
            .willReturn(Optional.empty());

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Who is this 'Pan' you're talking about?"));
    }
}
```

我们写单元测试用的是JUnit (<https://junit.org/junit5/>), Java 实际意义上的标准测试框架。我们使用Mockito (<https://site.mockito.org/>)来打桩隔离掉真正的PersonRepository类。这个桩允许我们在测试里重新定义 PersonRepository 被调用后产生的响应。桩能让我们的测试更简单, 更可预测, 更容易组织测试数据。

依照 Arrange, Act, Assert 的结构, 我们写了两个单元测试: 一个是正常的场景, 另一个是找不到搜索人的场景。首先, 正常场景创建了一个新的 person 对象, 然后告诉 mock 类, 当你接受到以“Pan”作为参数的调用时, 返回这个 person 对象。这个测试接着调用了被测试方法。最后它断言返回值是等于期待结果的。

第二个测试和第一个类似, 但它测试的是被测方法找不到对应人名时的场景。

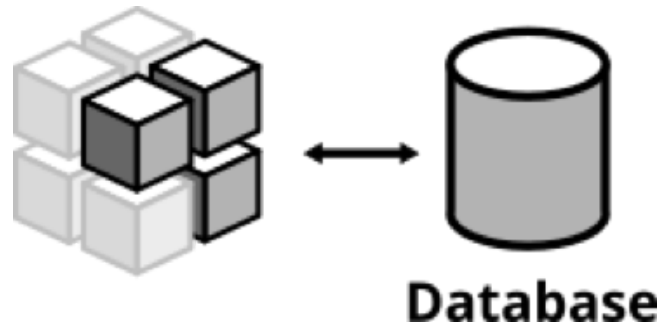
集成测试

所有常见的应用都会和一些外部环境做集成(数据库, 文件系统, 向其他应用发起网络请求)。为了使测试有更好的隔离、运行更快, 我们通常不会在编写单元测试时涉及这些外部依赖。不过, 这些交互始终是存在的, 它们也需要被测试覆盖到。这正是集成测试的用处所在。它们测试的是应用与所有外部依赖的集成。

对于自动化测试来说, 不仅需要运行自己的应用, 也需要运行与之集成的组件。如果要测试和数据库的集成, 那就需要在跑测试的时候运行数据库。如果要测试能否从硬盘里读取文件, 就需要先保存一个文件到硬盘上, 然后在集成测试中去读取它。

前面我提到过「单元测试」是一个模糊的术语, 对于集成测试而言, 更是如此。对于一些人来讲, 集成测试意味着去测试和多方应用产生交互的整个应用。我理解的集成测试更加狭义: 每次只测试一个集成点。测试时应使用测试替身来替代其他的外部服务、数据库等。同时, 使用契约测试对测试替身和真实实现进行覆盖。这样出来的集成测试更快, 更独立, 更易理解和调试。

狭义的集成测试测的是服务的边界。从概念上来说，这样的测试总是在触发导致应用和外部依赖（文件系统，数据库，其他服务等）集成的行为。比如说，一个数据库集成测试可能会这么写：

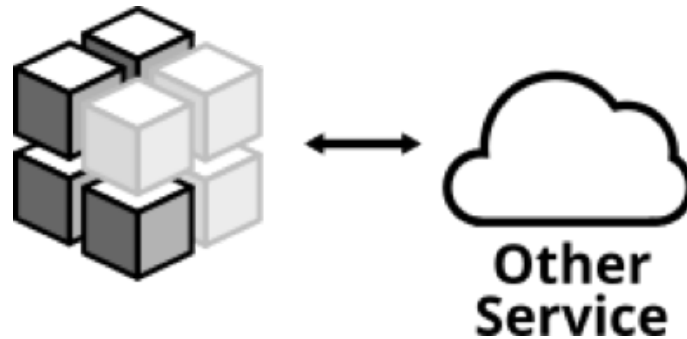


(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/7.png>).

(一个集成了你的代码和数据库的集成测试)

1. 启动数据库
2. 连接应用到数据库
3. 调用被测函数，该函数会往数据库写数据
4. 读取数据库，查看期望的数据是不是被写到了数据库里

另一个例子，一个通过 REST API 和外部服务集成的测试可能是会这么写：



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/8.png>).

(这种集成测试检查了应用是否能正确和外部服务通信)

1. 启动应用
2. 启动一个被测外部服务的实例（或者一个具有相同接口的测试替身）
3. 调用被测函数，该函数会从外部服务的 API 读取数据
4. 检查应用是否能正确解析返回结果

与单元测试一样，集成测试也可以写得很白盒。一些框架在应用启动后，仍然支持对应用的一些部分进行 mock。这使得你可以去检查正确的交互是否发生。

代码中所有涉及数据序列化和反序列化的地方都要写集成测试。这些场景可能比你想象得更多，比如说：

- 调用自身服务的 REST API
- 读写数据库
- 调用外部服务的 API
- 读写队列
- 写入文件系统

为这些边界编写集成测试，保证了对外部系统的数据读写操作是正常工作的。

编写狭义的集成测试时，你应该尽可能在本地运行外部依赖，如启动一个本地的 MySQL 数据库、针对本地的 ext4 文件系统进行测试等。如果是与外部服务集成，可以在本地运行该服务的实例，或构建一个模拟真实服务的假服务，并在本地运行。

如果有些三方服务，你没法在本地运行一个实例，那么可以考虑运行一个专用实例，并在集成测试中指向它。避免在自动化测试里集成真实的生产环境的服务。在生产环境上爆出上千个测试请求是个惹人生气的好办法，因为你会扰乱日志（这是最好的情况），最坏的情况是你会对该服务产生 DoS 攻击。透过网络和一个服务集成是广义集成测试的一大特征，这会让你的测试更慢，通常也更难编写。

在测试金字塔中，集成测试的层级比单元测试更高。集成缓慢的外部依赖（如文件系统或数据库等）通常比隔离了这些依赖的单元测试需要更长时间。他们可能比小型并且独立的单元测试难写，毕竟你需要让外部依赖在你的测试中运行起来。然而，它的优势在于建立了你对应用能正确访问外部依赖的自信，这是单纯的单元测试做不到的。

数据库集成

PersonRepository 是代码里唯一的数据库类。它依赖于Spring Data，我们并没有实际去实现它。只需要继承CrudRepository接口并声明一个方法名。剩下的就是 Spring 魔法了，Spring 会帮我们实现其他所有的东西。

```
public interface PersonRepository extends CrudRepository<Person, String> {  
    Optional<Person> findByLastName(String lastName);  
}
```

对于CrudRepository接口，Spring Boot 提供了完整的 CRUD 方法例如findOne, findAll, save, update和delete。我们自定义的方法（findByLastName()）继承了这些基础功能并实现了根据 last name 获取 Persons 对象的功能。Spring Data 会解析方法的返回类型，并按照命名规范解析方法名，从而决定如何实现方法。

虽然 Spring Data 已经实现了和数据库交互的功能，我还是写了一个数据库集成测试。你可能会反对，认为这是在测试框架，而我们应该避免测试不属于我们开发的代码。然则，我坚信在这里写一个集成测试是至关重要的。首先它测试了我们自定义的 findByLastName 方法实际的行为如我们所愿。次之，它证明了我们的数据库类正确地使用了 Spring 的装配特性，它是能正确连接到数据库的。

为了让你能更容易在本地把测试运行起来（而不必真的装一个 PostgreSQL 数据库），我们的测试会连接到一个内存H2数据库。

我已经在build.gradle里定义 H2 作为测试依赖。而测试目录下的application.properties没有定义任何spring.datasource属性。这会告诉 Spring Data 使用内存数据库，它会在 classpath 里找到 H2 来跑我们的测试。

当你用 int profile 真正启动应用时（例如把SPRING_PROFILES_ACTIVE=int设置到环境变量里），它会连接到application-int.properties里定义的 PostgreSQL 数据库。

我知道这涉及到了很多 Spring 的知识。你必须仔细阅读许多文档(<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-embedded-database-support>)才能理解这个例子。实现代码只有几行，非常直观，但是如果你不知道 Spring 的一些知识点是很难加以理解的。

除此以外，测试使用内存数据库其实是有风险的。毕竟我们集成测试针对的数据库和我们生产用的数据库不一样。你可以自己选择，是利用 Spring 的强大能力获得简洁的代码，亦或者是写显式但较为冗长的实现。

解释已经足够多了，这里有一个集成测试的例子。它先保存了一个 Person 对象到数据库里，然后根据 last name 去查找它。

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryIntegrationTest {
    @Autowired
    private PersonRepository subject;

    @After
    public void tearDown() throws Exception {
        subject.deleteAll();
    }

    @Test
    public void shouldSaveAndFetchPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
        subject.save(peter);

        Optional<Person> maybePeter = subject.findByLastName("Pan");

        assertThat(maybePeter, is(Optional.of(peter)));
    }
}
```

你可以看到我们的集成测试像单元测试那样遵循了arrange, act, assert的结构。我说过这是一个普适的概念吧。

和外部服务集成

我们的微服务会调用`darksky.net` (<https://darksky.net/>)——一个关于天气的 REST API。当然啦，我们希望保证服务调用时能发送正确的请求，并且能正确地解析响应。

跑自动化测试时，我们希望避免真实地调用darksky的服务。当然，我们使用的免费版有调用次数限制，这是个原因。但真正的原因是要去耦合。我们的测试应该能独立运行，而不需要管 darksky.net 可爱的开发者们在干些啥。即使我们的机器访问不到darksky服务器，或darksky服务器在进行宕机维护，都不应该使我们的测试挂掉。

我们可以在集成测试中用自己的假darksky服务器来代替真正的服务器。这听起来像是个巨大的任务。幸亏有像Wiremock (<http://wiremock.org/>)这样的工具，事情变得很简单。看这里：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class WeatherClientIntegrationTest {

    @Autowired
    private WeatherClient subject;

    @Rule
    public WireMockRule wireMockRule = new WireMockRule(8089);

    @Test
    public void shouldCallWeatherService() throws Exception {
        wireMockRule.stubFor(get(urlPathEqualTo("/some-test-api-key/53.5511,9.9937"))
            .willReturn(aResponse()
                .withBody(FileLoader.read("classpath:weatherApiResponse.json"))
                .withHeader(CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
                .withStatus(200)));

        Optional<WeatherResponse> weatherResponse = subject.fetchWeather();

        Optional<WeatherResponse> expectedResponse = Optional.of(new WeatherResponse("Rain"));
        assertThat(weatherResponse, is(expectedResponse));
    }
}
```

为了使用 Wiremock，我们在固定的端口(8089)实例化了一个WireMockRule。使用领域特定语言，我们可以配置一个 Wiremock 服务器，定义它需要监听的路径并设置相应的应答。

接着调用我们要测试的方法——它会调用第三方服务，然后检查结果是否能被正确解析。

理解测试怎样调用 Wiremock 服务器而不是真正的darksky很重要。秘密就在src/test/resources下的application.properties文件。这是 Spring 在运行测试时会加载的属性文件。出于测试目的——比如，调用一个假的 Wiremock 服务器而不是真实服务器——我们在这个文件里覆写了一些配置，如 API keys 和 URL 等：weather.url = http://localhost:8089

值得注意的一点是，这里声明的端口必须和我们在测试里实例化WireMockRule时的端口保持一致。我们之所以能为了测试注入一个假的 API url，是因为我们通过注入的方式将 url 传给了WeatherClient类的构造函数：

```
@Autowired
public WeatherClient(final RestTemplate restTemplate,
                    @Value("${weather.url}") final String weatherServiceUrl,
                    @Value("${weather.api_key}") final String weatherServiceApiKey) {
    this.restTemplate = restTemplate;
    this.weatherServiceUrl = weatherServiceUrl;
    this.weatherServiceApiKey = weatherServiceApiKey;
}
```

这样我们告诉WeatherClient要把我们定义在 application properties 的weather.url值赋给weatherUrl。

借助类似 Wiremock 这样的工具，为外部服务编写狭义的集成测试就变得很简单。不幸的是这种方式有个缺点：如何保证我们启动的假服务器与真的服务行为一致？按我们目前的实现，当外部服务改变了它的 API 时，我们的测试依然能跑过。现在我们仅仅测试了 WeatherClient可以解析假服务器返回的应答信息。这是个好的开始，但是它非常脆弱。如果使用端到端测试，针对真实服务的实例运行测试，而不使用假的服务，固然能解决这个问题，但这又会让我们的测试对被测服务的可用性产生依赖。幸运的是，针对这个难题还是有更好的方案：针对真实和假的服务运行契约测试。这能保证我们集成测试里用的假服务是个忠实的测试替身。下面来看看这种方案是怎么工作的。

契约测试

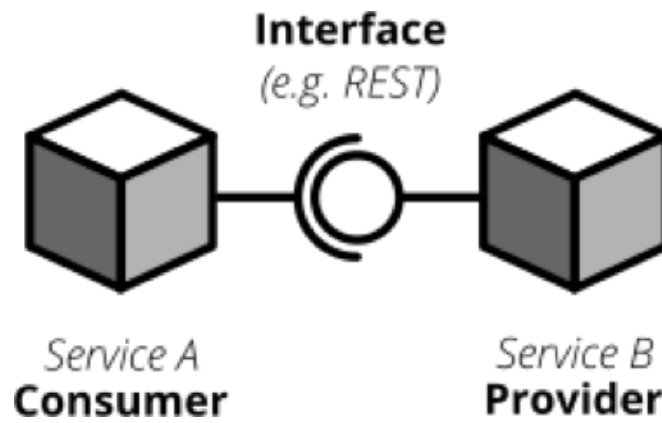
越来越多现代软件组织发现，对于增长的开发需求，可以让不同的团队来开发同一系统的不同部分。每个团队负责构建独立、松耦合的服务，团队间开发不互相影响。最终再将这些服务集成为一个大而全的系统。最近关于微服务的讨论日益热烈，关注的正是这一点。

将系统拆分成多个更小的服务，常常意味着这些服务之间需要通过确定的（最好是定义明确的，但有时候会有变动演进）接口通信。

不同应用间的接口可能形态各异，或基于不同的技术栈。常见的有：

- 基于 HTTPS 使用 JSON 交互的 REST 接口
- 基于类似gRPC (<https://grpc.io/>)的 RPC（Remote Procedure Call，远程进程调用）接口
- 使用队列构建的事件驱动架构

对于任意一个接口，一定会涉及两个实体：提供方和消费方。提供方为消费方提供数据。消费方处理来自提供方的数据。在 REST 世界里，提供方为所有要暴露的 API 创建一个 REST API；消费方则调用这些 API 来获取数据，或进一步触发其他的服务。而在一个由异步、事件驱动的世界，提供方（通常被称为发布者）发布数据到一个队列中；消费方（通常被称为订阅者）订阅这些队列，读取并处理相关数据。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/9.png>).

(每一个接口都有提供方（或者发布者）和消费方（或者订阅者）实体。接口之间的规范可以视为是一个契约。)

当你把服务消费方和服务提供方分散到不同的团队去时，你就需要清楚地了解这些服务之间的接口（也就是我们所讲的契约）。传统的公司一般是通过以下方式解决这个问题：

- 写一个钜细靡遗的接口文档（就是契约）
- 根据定义好的契约实现提供方服务
- 把接口文档扔给隔壁的消费团队
- 等。等到消费方团队实现接口消费部分的工作
- 运行一些大型的、手动的系统测试，保证软件能正常工作
- 祈祷双方团队永远都维持接口定义不变，不要把事情搞砸

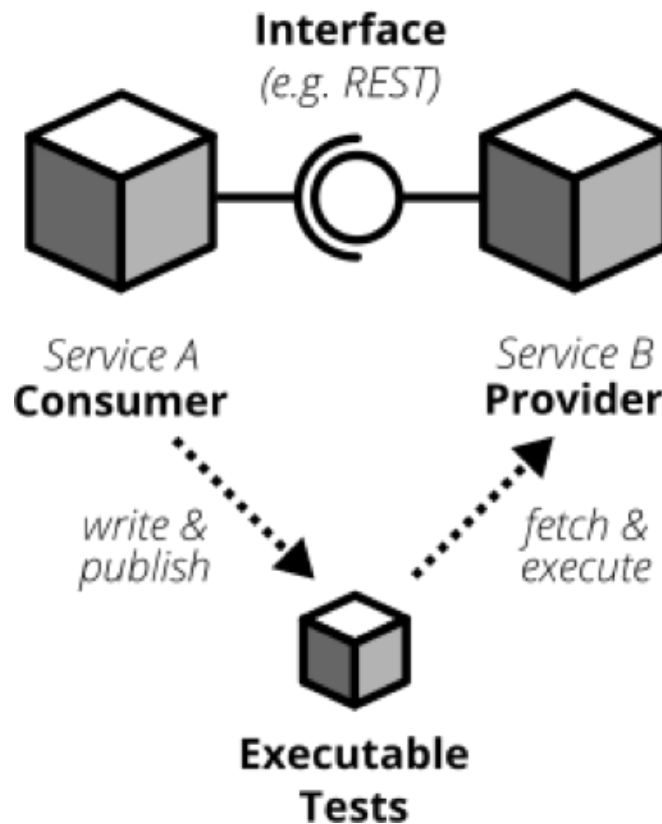
越来越多现代软件开发团队已经把第五步和第六步用更加自动化的方式来替代：自动化契约测试

(<https://martinfowler.com/bliki/ContractTest.html>)保证了消费方和提供方实现的时候依然遵循契约。这种测试提供了一个良好的回归测试组合，保证契约的变更能被及早发现。

在现代敏捷组织，你应该选择效率高浪费少的路子。你们是在同一个公司里构建应用。比起扔出去一个面面俱到的文档，与其他服务的开发者们直接交流本应容易得多。毕竟他们是你的同事，而不是一个只能通过客户支持或合同进行沟通的第三方供应商。

消费方驱动的契约测试（Consumer-Driven Contract tests，CDC 测试）是让消费方驱动契约实现

(<https://martinfowler.com/articles/consumerDrivenContracts.html>)。使用 CDC，接口消费方会写测试，检查接口是不是返回了他们想要的所有数据。消费方团队会发布这些测试从而让提供方可以轻松获取到这些测试并执行。提供方团队现在可以一边运行 CDC 测试一边开发他们的 API 了。一旦所有测试通过，他们就知道已经实现了所有消费方想要的东西。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/10.png>)

(契约测试保证了提供方和所有的消费方基于同一个定义好的接口契约。用 CDC 测试，消费者就可以通过自动化测试发布他们的需求，提供方则可以持续不断获取这些测试并执行)

这种方式允许提供方团队只实现必要的东西（让设计保持简约，[YAGNI](https://martinfowler.com/bliki/Yagni.html) (<https://martinfowler.com/bliki/Yagni.html>)等）。提供方团队需要持续地获取并运行这些 CDC 测试（从他们的构建 Pipeline 里），从而能立即发现任何打破契约的代码变更。如果有代码变更破坏了接口，CDC 测试应该会执行失败，这样可以防止破坏性改动上线。当这些测试保持通过，团队就可以做任何他们想做的改动而不需要担心其他团队。使用消费方驱动测试的话，一般过程会是这样的：

- 消费方团队根据他们期待的结果编写自动化测试
- 发布自动化测试给提供方团队
- 提供方持续不断地运行这些测试，并保持他们都能通过
- 如果 CDC 测试挂掉了，则需要双方进行沟通

如果你的组织正在践行微服务，那么拥有 CDC 测试将是迈向自治团队的一大步。CDC 测试是一种促进团队交流的自动化途径。它们保证了团队间的接口能一直如期工作。如果有 CDC 测试挂掉，则可能是个好的信号，意味着你应该走过去到那个被测试影响的团队，了解他们最近是否有 API 变更，弄清楚你们希望如何处理这些变更。

一个稚嫩的 CDC 测试实现非常简单，比如说你可以对一个 API 发送请求，并断言响应中包含了你需要的所有东西。然后你把这些测试打包成可执行文件（.gem, .jar, .sh），并将它们上传到一个其他团队可以获取到的地方（例如一些诸如 [Artifactory](https://www.jfrog.com/artifactory/) (<https://www.jfrog.com/artifactory/>)这样的仓库）。

在过去几年里，CDC 正变得越来越受欢迎。同时也涌现了一些工具，使得编写及上传 CDC 更加简单。

在这些工具中，[Pact](https://github.com/realestate-com-au/pact) (<https://github.com/realestate-com-au/pact>)可能是最显眼的的一个了。它为编写提供方或消费方的测试提供了详尽的支持，为外部服务隔离提供了开箱即用的（打）桩工具，它还支持你与其他团队交换 CDC 测试。Pact 已经被移植到很多平台上，并且可以和 JVM 语言一起使用，例如 Ruby，.NET，JavaScript 等等。

如果你想开始编写 CDC 测试但不知道怎么开始，不妨试试 Pact。文档(<https://docs.pact.io/>)一开始可能会让你应接不暇。保持耐心克服一下。它能帮助你深刻理解 CDC 测试，也会让你更容易在团队合作时推行 CDC。

消费方驱动契约测试真的可以说是建立自治团队的基石，它让这样的团队充满自信，快速前行。不要掉队，好好读读相关的文档，尝试一下。一套稳固的 CDC 测试集非常宝贵，它让你能快速开发，同时又不会挂掉已有的服务，引起其他团队的不满。

消费方测试（我方团队）

上面的例子中，我们的微服务消费了天气 API。所以我们有责任写一个消费方测试来定义我们期望从 API 契约中获得的结果。

首先，我们要在 build.gradle 里引入一个库来写基于协议的消费方测试：

```
testCompile('au.com.dius:pact-jvm-consumer-junit_2.11:3.5.5')
```

得益于这个库，我们可以用协议的仿造服务来实现一个消费方测试：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class WeatherClientConsumerTest {

    @Autowired
    private WeatherClient weatherClient;

    @Rule
    public PactProviderRuleMk2 weatherProvider =
        new PactProviderRuleMk2("weather_provider", "localhost", 8089, this);

    @Pact(consumer="test_consumer")
    public RequestResponsePact createPact(PactDslWithProvider builder) throws IOException {
        return builder
            .given("weather forecast data")
            .uponReceiving("a request for a weather request for Hamburg")
                .path("/some-test-api-key/53.5511,9.9937")
                .method("GET")
            .willRespondWith()
                .status(200)
                .body(FileLoader.read("classpath:weatherApiResponse.json"),
                    ContentType.APPLICATION_JSON)
            .toPact();
    }

    @Test
    @PactVerification("weather_provider")
    public void shouldFetchWeatherInformation() throws Exception {
        Optional<WeatherResponse> weatherResponse = weatherClient.fetchWeather();
        assertThat(weatherResponse.isPresent(), is(true));
        assertThat(weatherResponse.get().getSummary(), is("Rain"));
    }
}
```

如果观察得仔细，你会发现 WeatherClientConsumerTest 和 WeatherClientIntegrationTest 很相似。这次我们用 Pact 取代了 Wiremock 来对服务器打桩。事实上消费方测试工作方式与集成测试完全一致：我们用打桩的方式隔离第三方服务，定义我们期望的响应，然后检查我们的客户端可以正确处理响应。从这个意义上讲，WeatherClientConsumerTest 本身就是一个狭义的集成测试。这种方式相比使用 Wiremock 好在，它每次运行都会创建一个协议文件（会生成到 target/pacts/< pact-name >.json）。这个协议文件使用特殊的 JSON 格式描述了这个契约的期望结果，它可以被用来验证我们打桩的服务与真实服务行为确实是一致的。我们可以把这个协议文件交给提供 API 的团队，他们可以根据这个文件的期望输出来编写提供方测试。这样的话他们就能测试，他们的 API 是不是满足我们期望的所有结果。

消费方通过描述他们的期望结果来驱动接口实现，这就是 CDC 里消费方驱动所想要表达的意思。提供方必须保证他们满足了所有期望结果。没有过度设计，保持简洁。把 Pact 文件交给提供方团队可以有几种方式。一种简单方式就是把它们加入到版本控制系统里，告诉提供方永远拉取最新的文件即可。更先进一点的方式则是用一个文件仓库，类似 Amazon S3 这样的服务或者 pact broker。起步迅速，按需拓展。

在真实的软件中，你并不需要为一个客户端类既写集成测试又写消费方测试。上面的示例代码同时包含了这两种测试，只是想告诉你这两种测试的写法。如果你想用协议来写 CDC 测试，我推荐你只写消费方测试。两种测试的编写成本是一样的。用协议的方式就有协议文件这个好处，这样把协议文件递交给其他团队，他们就能很容易实现他们的提供方测试。当然这取决于你能说服其他团队也使用协议。如果不行，那么用 Wiremock 来实现集成测试可以作为替代方案。

提供方测试（其他团队）

提供方测试必须由提供天气 API 的团队来实现。我们消费的是 darksky.net 提供的一个公共 API。理论上 darksky 团队会实现提供方测试，以保证不打破他们应用和我们的服务之间的契约。

很明显他们不会关注我们这个简单的示例代码库，也不会为我们实现 CDC 测试。这是公共 API 和组织内微服务的一大不同点。公共 API 不可能考虑到每一个消费方，否则他们就得整天忙于写测试了。而在我们自己组织内，你能够、也应该考虑每个消费方。你的应用一般只会服务于少量的，最多几十个消费方。为这些接口编写提供方测试应该不是太大的问题，这可以保证系统稳定。

提供方团队拿到协议文件后，会在他们的服务上运行一下。这需要一个提供方测试，在测试中读取协议文件，打桩隔离掉一些测试数据，运行他们的服务，并检查是否返回了协议文件中期望的结果。

Pact 团队写了一些库来实现提供方测试。他们在主 GitHub 仓库 (<https://github.com/DiUS/pact-jvm>) 写了一个很好的概览，告诉你有哪些消费方/提供方测试的库是可用的，你只需要从中选择适用于你技术栈的即可。

为了简单起见，我们假设 darksky 的 API 也是用 Spring Boot 来实现的。这样的话他们就可以用 [Spring pact provider](https://github.com/DiUS/pact-jvm/tree/master/pact-jvm-provider-spring) (<https://github.com/DiUS/pact-jvm/tree/master/pact-jvm-provider-spring>) 来写，这个库和 Spring 的 MockMvc 机制做了很好的适配。我们假想 darksky.net 团队写了提供方测试，那么它大概长这样：

```
@RunWith(RestPactRunner.class)
@Provider("weather_provider") // same as the "provider_name" in our clientConsumerTest
@PactFolder("target/pacts") // tells pact where to load the pact files from
public class WeatherProviderTest {
    @InjectMocks
    private ForecastController forecastController = new ForecastController();

    @Mock
    private ForecastService forecastService;

    @TestTarget
    public final MockMvcTarget target = new MockMvcTarget();

    @Before
    public void before() {
        initMocks(this);
        target.setControllers(forecastController);
    }

    @State("weather forecast data") // same as the "given()" in our clientConsumerTest
    public void weatherForecastData() {
        when(forecastService.fetchForecastFor(any(String.class), any(String.class)))
            .thenReturn(weatherForecast("Rain"));
    }
}
```

你可以看到提供方测试必须做的就是两点：加载一个协议文件（例如，用 @PactFolder 注解来自动加载已下载好的协议文件）、提供需要准备的数据（例如使用 Mockito 来仿造）。此外，不需要再实现额外的测试，它会从协议文件自动派生出来。对于消费方测试里声明的提供方名称 (provider name) 和状态，提供方测试应该一一匹配。

提供方测试（我方团队）

我们已经看了如何测试我们服务和天气提供方之间的契约。对于这个接口，我们的服务扮演的是消费方，天气服务则扮演了提供方。考虑得更远一些，会发现我们的服务同时也是其他系统的提供方：我们为数个路径提供了 REST API 以供其他系统消费。

我们刚认识到了契约测试什么场景都能用，当然我们也会想给我们的契约写一写契约测试。幸运的是，我们使用了消费方驱动契约，所以我们手里有所有的消费方发过来的协议，可以用它们来实现我们的 REST API 提供方测试。

先把 pact-jvm-provider 库装上：

```
testCompile('au.com.dius:pact-jvm-provider-spring_2.12:3.5.5')
```

提供方测试的实现与前面所述的范式相同。为简单起见，我直接从[simple consumer \(https://github.com/hamvocke/spring-testing-consumer\)](https://github.com/hamvocke/spring-testing-consumer)拿来一份协议文件放到了我们的仓库中，这会让我们操作起来简单一些。在真实的项目，你可能需要更完善的机制来分发协议文件。

```
@RunWith(RestPactRunner.class)
@Provider("person_provider") // same as in the "provider_name" part in our pact file
@PactFolder("target/pacts") // tells pact where to load the pact files from
public class ExampleProviderTest {

    @Mock
    private PersonRepository personRepository;

    @Mock
    private WeatherClient weatherClient;

    private ExampleController exampleController;

    @TestTarget
    public final MockMvcTarget target = new MockMvcTarget();

    @Before
    public void before() {
        initMocks(this);
        exampleController = new ExampleController(personRepository, weatherClient);
        target.setControllers(exampleController);
    }

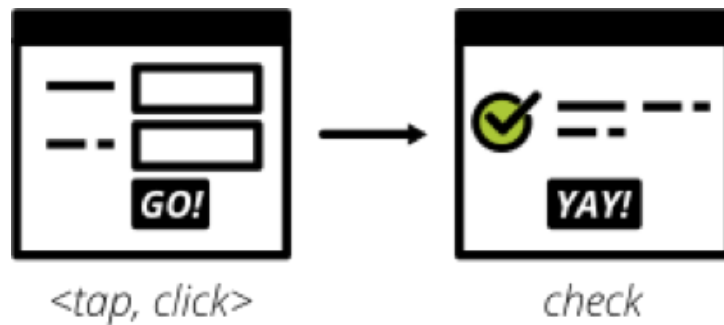
    @State("person data") // same as the "given()" part in our consumer test
    public void personData() {
        Person peterPan = new Person("Peter", "Pan");
        when(personRepository.findByLastName("Pan")).thenReturn(Optional.of(
            peterPan));
    }
}
```

ExampleProviderTest需要做的事只有一件，那就是根据协议文件里的内容提供State信息。当我们运行提供方测试时，Pact 就会用到指定的协议文件，并发送 HTTP 请求到我们的服务，然后根据我们配置的 State 来决定响应。

UI 测试

大部分的应用都会有些用户界面。在 web 应用的上下文中，我们所谈的界面就是指网页界面。但人们经常会忘记，除了多彩的网页页面，还有许多的 REST API 界面或命令行界面等。

UI 测试测的是应用中的用户界面是否如预期工作。比如，用户的输入需要触发正确的动作，数据需要能展示给用户看，UI 的状态需要发生正确变化等。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/11.png>)

有时候，人们提到 UI 测试和端到端测试时（比如 Mike Cohn）说的是一个东西。对我而言，这种观点混淆了这两个有交集的不同概念。

诚然，端到端的测试通常意味着会测到许多用户界面。但是反过来讲却并不能成立。

测试用户界面不必非得通过端到端的方式完成。根据技术栈不同，有时测试用户界面也可以很简单，只需要为前端的 JavaScript 代码写一些单元测试，同时用桩将后端隔离开即可。

对于传统的网页应用，UI 测试可以用 [Selenium](https://docs.seleniumhq.org/) (<https://docs.seleniumhq.org/>) 这一类工具完成。如果你把 REST API 也当成一个用户界面，对你的 API 写一些恰当的集成测试可以达到完全相同的目的。

对于网页界面而言，你的 UI 大概可以围绕这几个部分进行测试：行为，布局，可用性，以及少数人认为需要测试的设计一致性。

幸运的是，测试用户界面的行为非常简单。点击一下，输入数据，然后看到用户界面状态如实变更。现代的单页应用框架（以 [react](https://reactjs.org/) (<https://reactjs.org/>)，[vue.js](https://vuejs.org/) (<https://vuejs.org/>)，[Angular](https://angular.io/) (<https://angular.io/>) 等为代表）通常都会提供一些工具或组件，帮你从很低的测试层级（单元测试）对界面交互进行测试。即便你没有使用任何框架，只使用纯 JavaScript，也有常规的测试工具（如 [Jasmine](https://jasmine.github.io/) (<https://jasmine.github.io/>) 或 [Mocha](https://mochajs.org/) (<https://mochajs.org/>) 等）可供选择。对于更传统一些的服务端渲染应用，使用 Selenium 会是最佳的选择。

测试应用的布局是否前后一致则有些困难。根据应用类型和用户需求的差异，也许你可能需要确保代码的更改不会意外破坏页面的布局。

问题是众所周知...计算机在检查某物「看起来是否不错」方面一直表现不佳（也许未来一些好的机器学习算法可以改善这一现状）。

如果你依然希望在构建流水线中集成自动化的测试来检查应用的设计，还是有些工具可以试一下。大部分的这些工具都是使用 Selenium 帮你在不同浏览器中打开应用、截图、跟之前的截图做对比。如果新旧截图的差异超过了预设阈值，工具就会告诉你。

[Galen](http://galenframework.com/) (<http://galenframework.com/>) 就是其中一种工具。即便你有特殊的需求，自己实现一套工具也不是很难。我之前工作过的一些团队就构建了 [lineup](https://github.com/otto-de/lineup) (<https://github.com/otto-de/lineup>)，以及基于 Java 的 [jlineup](https://github.com/otto-de/jlineup) (<https://github.com/otto-de/jlineup>)，用以实现类似的测试工具。如我前面所说，这两种工具都使用了 Selenium。

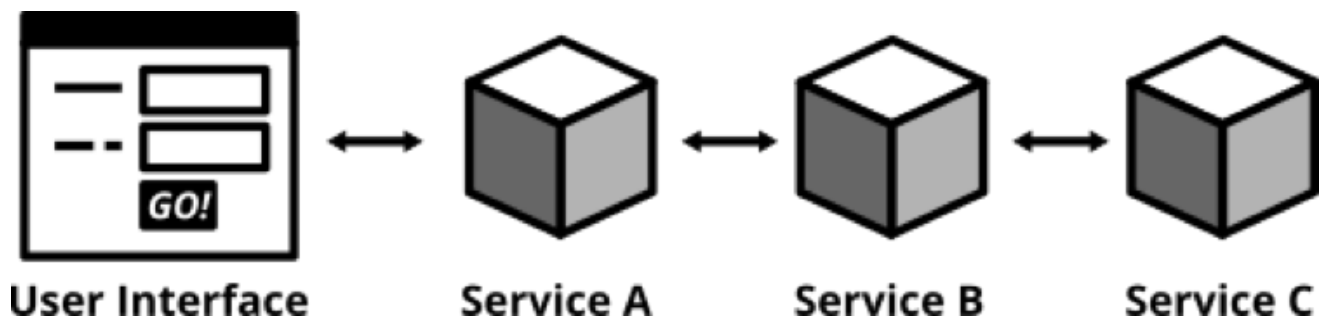
当你想测试可用性或一些「看起来对不对」的东西时，你已经超越了自动化测试的范畴。这是 [探索性测试](https://en.wikipedia.org/wiki/Exploratory_testing)

(https://en.wikipedia.org/wiki/Exploratory_testing)，可用性测试（这甚至可以像 [走廊测试](https://en.wikipedia.org/wiki/Usability_testing#Hallway_testing)

(https://en.wikipedia.org/wiki/Usability_testing#Hallway_testing) 那般简单) 的领域。你需要给你的用户展示产品，看看他们是否喜欢使用它，有没有什么功能会让他们在使用时感到困惑或恼火。

端到端测试

通过用户界面测试一个已部署好的应用，可以说是最端到端的方式了。前面说的以 webdriver 驱动的 UI 测试就是一个很好的端到端测试案例。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/12.png>)

(端到端测试测试了整个的、完全集成的系统)

端到端测试（也被称为**广域栈测试** (<https://martinfowler.com/bliki/BroadStackTest.html>)) 会赋予你极大的信心，让你了解软件是否正常工作。**Selenium** (<https://docs.seleniumhq.org/>)和**WebDriver** 协议 (<https://www.w3.org/TR/webdriver/>)使你能够针对部署好的服务进行自动化测试，它能启动一个无头 (headless) 浏览器来对用户界面执行点击、输入、检查状态的操作。当然你也可以直接使用 Selenium，或者用类似**Nightwatch** (<http://nightwatchjs.org/>)这种基于 Selenium 的工具。

端到端测试也有它特有的一些问题。众所周知，它们通常比较脆弱，经常因为一些意料之外的问题挂掉。并且这些错误信息通常不是真正的原因所在。用户界面越复杂，测试常常越是脆弱。浏览器差异、时间 (时序) 问题、元素渲染、意外的弹出框...这还仅是列表一角，已经让我经常花大量时间进行调试，这实在令人沮丧。

在微服务的世界中，谁负责写这些测试也是一个大问题。因为端到端测试覆盖到数个服务 (整个系统)，导致编写端到端测试不是任何一个团队的责任。

如果你们有一个集中的质量保障团队，由他们来编写端到端测试看起来就不错。但是呢，拥有一个集中式的 QA 团队同时也是一种明显的反模式，这根本不应该出现在 DevOps 的世界中。你的团队应该是真正的跨职能团队。回答谁该对端到端测试负责这个问题并不容易。也许你的组织里会有一些社区实践，或有个质量协会之类的机构能为此负责。一个合适的答案，与你的组织本身高度相关。

此外，端到端测试还需要大量的维护成本，运行起来也相当慢。试想一下这样的场景，除非只有几个微服务，否则你根本没办法在本地运行端到端测试，因为你需要启动所有的服务。祝你的机器能同时跑起几百个应用，并且内存没被撑爆。

因其高昂的维护成本，你应该尽量将端到端测试的数量减少到最低限度。

考虑一下应用中对用户而言具有高价值的交互。定义好产品产生核心价值的用户旅程，然后将这些旅程中最重要的步骤变成自动化的端到端测试。

举例来说，假设你正在构建一个电子商务网站，最具价值的顾客旅程可能是这样的：用户搜索一件商品、将其加入购物车，然后付款。就这么简单。只要这个旅程正常工作，你应该无需过多担心。也许你可以找出一两个重要的用户旅程，并将其用端到端测试来覆盖。但到此为止，再多的测试就会开始带来痛苦了。

谨记：在测试金字塔里，有很多更低层级的测试，这些测试已经全面测试了各种边缘情况及与其他系统的集成。不需要再在高层级测试里重复测一遍。否则，高维护成本和一堆谎报错误将会降低开发速度，迟早会让你对测试失去信心。

用户界面端到端测试

对于端到端测试来说，**Selenium** (<https://docs.seleniumhq.org/>)和**WebDriver** (<https://www.w3.org/TR/webdriver/>)协议是大部分开发者的选择。用 Selenium 你可以选一个喜欢的浏览器，它会自动帮你访问网页、触发点击事件、输入一些数据，并检查用户界面的变更。

Selenium 需要一个浏览器用来运行测试。有几种所谓的驱动 (drivers) 可以用来启动不同的浏览器。选一种 (<https://www.mvnrepository.com/search?q=selenium+driver>) (或几种) 加到 build.gradle 里。不管选择什么浏览器，都需要保证团队里所有开发者以及 CI 服务器上安装对应版本的浏览器。保持同步有时相当困难。如果你用的是 Java，有一个轻量级的库叫 **webdrivermanager** (<https://github.com/bonigarcia/webdrivermanager>)，它会自动帮你下载并设置好正确版本的浏览器。把这些依赖加到 build.gradle 文件里即可：

```
testCompile('org.seleniumhq.selenium:selenium-chrome-driver:2.53.1')
testCompile('io.github.bonigarcia:webdrivermanager:1.7.2')
```

为测试运行一个完整的浏览器有时候会是一种麻烦事。尤其对于持续交付跑 Pipeline 的服务器，也许没有时间打开一个包含用户界面的浏览器(例如因为当前没有可用的 X-Server)。变通办法就是使用类似xvfb (<https://en.wikipedia.org/wiki/Xvfb>)那样的虚拟 X-Server。(看来得好好了解一下 X-Server 是啥，否则不翻译可能会有问题)

更先进一点的方案是使用无头浏览器（也就是没有用户界面的浏览器）来运行 webdriver 测试。截至目前PhantomJS (<http://phantomjs.org/>)依然是做浏览器自动化最好的无头浏览器（译者注：其实现在 PhantomJS 作者已经宣布停止维护了，建议转向使用Puppeteer (<https://github.com/GoogleChrome/puppeteer>)）。但这之后Chromium (<https://developers.google.com/web/updates/2017/04/headless-chrome>)和Firefox (https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode)双双宣布他们在各自的浏览器里实现了无头模式，这使得 PhantomJS 突然显得有些过时了。比起仅为了你自己作为开发者的方便而使用人工浏览器，使用用户真正使用的浏览器（如 Firefox 和 Chrome）来做测试应该是更好的选择。

不管是无头 Firefox 还是 Chrome，都是新出的东西，还没有广泛应用到实现 webdriver 测试的场景。这里我们希望简化一些，不去折腾走在时代前沿的浏览器无头模式，而是直接用传统的 Selenium 和一个普通的浏览器。以下是一个简单的端到端测试的例子，它将启动 Chrome、访问我们的服务，并检查页面的内容是否正确：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloE2ESeleniumTest {

    private WebDriver driver;

    @LocalServerPort
    private int port;

    @BeforeClass
    public static void setUpClass() throws Exception {
        ChromeDriverManager.getInstance().setup();
    }

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
    }

    @After
    public void tearDown() {
        driver.close();
    }

    @Test
    public void helloPageHasTextHelloWorld() {
        driver.get(String.format("http://127.0.0.1:%s/hello", port));

        assertThat(driver.findElement(By.tagName("body")).getText(), containsString("Hello World!"));
    }
}
```

这个测试只能在你装好 Chrome 的系统里运行起来（本地机器，CI 服务器）。

这个测试很直观。它用@SpringBootTest在一个随机端口启动了整个 Spring 应用。然后我们实例化了一个 Chrome 的 webdriver，告诉它去访问我们微服务的/hello 路径，然后检查浏览器里是不是打印出了"Hello World!"的字样。看起来很酷哟！

REST API 端到端测试

在测试应用时，如果能避免涉及图形化的用户界面，将有望写出比完整的端到端测试更健壮的测试，同时依然能覆盖到大部分的应用。这在测试 web 界面异常困难时大有用处。也许你的应用根本没有界面，仅仅是提供了 REST API（比方说你有个单页应用会调用到这个 API，或单纯因为你鄙视一切好用而漂亮的界面）。不管怎么说，有一类皮下测试

(<https://martinfowler.com/bliki/SubcutaneousTest.html>)仅仅测试图形化用户界面背后的东西，但依然能给你带来足够的信心。如果你也像咱的示例代码一样，只是暴露出一个 REST API，那么这种测试方法就非常合适。

```
@RestController
public class ExampleController {
    private final PersonRepository personRepository;

    // shortened for clarity

    @GetMapping("/hello/{lastName}")
    public String hello(@PathVariable final String lastName) {
        Optional<Person> foundPerson = personRepository.findByLastName(lastName);

        return foundPerson
            .map(person -> String.format("Hello %s %s!",
                person.getFirstName(),
                person.getLastName()))
            .orElse(String.format("Who is this '%s' you're talking about?",
                lastName));
    }
}
```

有一个库，在测试提供 REST API 的服务时很好用：[REST-assured](https://github.com/rest-assured/rest-assured) (<https://github.com/rest-assured/rest-assured>)。它提供了优雅的 DSL，让你可以优雅地向待测 API 发出真实的 HTTP 请求，并检验收到的响应。第一件事：把这个库加到 build.gradle 里。

```
testCompile('io.rest-assured:rest-assured:3.0.3')
```

用这个库，我们可以这样实现我们针对 REST API 的端到端测试：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloE2ERestTest {

    @Autowired
    private PersonRepository personRepository;

    @LocalServerPort
    private int port;

    @After
    public void tearDown() throws Exception {
        personRepository.deleteAll();
    }

    @Test
    public void shouldReturnGreeting() throws Exception {
        Person peter = new Person("Peter", "Pan");
        personRepository.save(peter);

        when()
            .get(String.format("http://localhost:%s/hello/Pan", port))
            .then()
            .statusCode(is(200))
            .body(containsString("Hello Peter Pan!"));
    }
}
```

这里我们还是用 `@SpringBootTest` 启动了一个完整的 Spring 应用。我们 `@Autowire` 了一个 `PersonRepository`，以便能很容易将测试数据写到数据库里。现在当我们发送了 API 请求，对 Pan 先生打招呼说 hello 时，我们将会收到一个友好的打招呼响应（Hello Peter Pan!）。神奇吧！如果你的应用没有用户界面，那么这个端到端测试就已绰绰有余了。

验收测试——你的功能工作正常吗？

测试金字塔越往上，越有可能需要从用户的角度来测试你所构建的功能是否能正常工作。你可以将应用视为黑盒，然后把测试关注点从这样的方式：

当我输入的值是 x 和 y 时，返回值应该是 z

转变为：

Given：在用户已经登陆

并且有一篇名为“bicycle”的文章的情况下

When：当用户进入了“bicycle”这篇文章的详情页面

并点击“添加到购物篮”按钮

Then：那么“bicycle”这篇文章应该出现在用户的购物车里

对于这样的测试，有时候你会听到像功能性测试 (https://en.wikipedia.org/wiki/Functional_testing) 或者验收测试 (https://en.wikipedia.org/wiki/Acceptance_testing#Acceptance_testing_in_extreme_programming) 这样的说法。偶尔会有人跟你说功能性和验收测试不是一回事。有时这些说法却又是一回事。人们可能对这些说法和定义陷入无尽的争论。这样的讨论经常会导致更多的混乱。

我认为是这样的：无论如何，你总会需要从用户的角度而非仅仅从技术角度来测试软件是否正常工作。你怎么称呼这种测试并不是太重要，写测试本身才重要。称呼随便挑一个，保持后续术语一致，然后就开始编写这些测试吧。

人们时常提及 BDD 及一些相关的工具，它们可以用 BDD 风格来编写这类测试。BDD 或者 BDD 风格写出来的测试较易将你的思维从实现细节转向关注用户需求。你完全可以试一试。

你甚至不必要采用已经十分成熟的 BDD 工具，例如 Cucumber (<https://cucumber.io/>) (虽然你也可以用)。有些断言库 (如 chai.js (<https://www.chaijs.com/guide/styles/#should>)) 也支持你使用 should-风格的断言，这可以使你的测试读起来更 BDD 一些。即便你不用这样的库，精心组织一下代码也可以使测试专注在用户行为上。一些小巧的 helper 方法/函数就能让你做到这一点：

一个用Python写的验收测试示例

```
def test_add_to_basket():
    # given
    user = a_user_with_empty_basket()
    user.login()
    bicycle = article(name="bicycle", price=100)

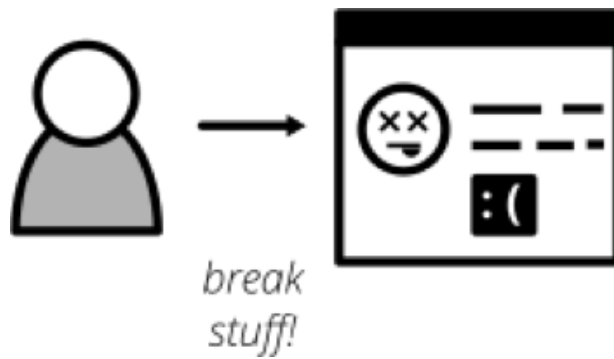
    # when
    article_page.add_to_.basket(bicycle)

    # then
    assert user.basket.contains(bicycle)
```

验收测试可以有不同粒度的层次。大部分时候它们所处的测试级别都比较高，一般是直接从用户界面上测试服务。不过从技术上讲，验收测试不必总是写在测试金字塔的最高层。如果你的应用设计和场景允许你在低层级写验收测试，那就这样做。把它写成低层级测试要比写成高层级测试好。验收测试的概念——证明在用户视角看来，应用是正常工作的——是和测试金字塔完全契合的。

探索测试

即便是最详尽的自动化测试，它也不是十全十美的。有时你总会在自动化测试里漏掉一些边缘情况。偶尔会出现一些难以单靠单元测试检测出来的 bug。某些质量问题甚至很难从自动化测试的视角被发现 (诸如设计和可用性等)。即使对自动化测试抱着崇高的期望，一定程度的手工测试也是不可避免的。



(<https://insights.thoughtworks.cn/wp-content/uploads/2018/10/13.png>)

(使用探索性测试揪出构建流水线上没能发现的问题)

把探索性测试 (https://en.wikipedia.org/wiki/Exploratory_testing) 纳入测试组合里。它是这样一种手工测试法：给予测试者自由，依赖他们的创造性来发现系统中的质量问题。定期花点时间，撸起袖子，试着对你的应用做些破坏性的操作，使其不能正常工作。开动你的破坏性思维，想方设法制造一些问题和错误。然后记录下你发现的所有东西，以供分析。其中要特别留意那些 bug、设计问题、很长的响应时间、缺失或有误导性的错误信息等一切会让作为用户的你感到恼怒的东西。

好消息是，一般你发现的大部分问题都能写自动化测试来覆盖。写自动化测试来覆盖发现的这些 bug，有助于日后的回归测试中不会重现同样的错误。并且，它能帮你在修复 bug 时，缩小 bug 产生根因的排查范围。

做探索性测试时，你可能会发现一些问题，但它们被构建流水线放过了。不用沮丧，这是关于流水线成熟度的绝佳反馈。收到反馈后，请采取必要的行动：思考一下，做点什么才能避免此类问题以后再次发生。也许是缺失了某一类自动化测试。也许是这个迭代的自动化测试做得马马虎虎，需要测试得更透彻。也许有好用的新工具或方案可以让你的流水线日后避开这类问题。总之，采取行动是必要的，这样你的流水线，你整个的软件交付将变得越来越成熟。

测试术语的困惑

讨论测试的不同分类总是十分困难。当我在讲单元测试的时候，可能与你理解的那个单元测试有些许差异；对于集成测试而言，可能差异会更大。一些人觉得，集成测试的覆盖面非常广，能测试到整个系统中的很多方面。对我而言它的范围则小得多，每个集成测试应仅仅测试一个与之集成的外部系统。有些人称这为集成测试，有些人则更喜欢称它们为组件测试，还有些人喜欢称为服务测试。很多人会争辩说，这三个术语是完全不同的东西。这里并无绝对的对与错。软件开发社区至今也没法给出关于测试术语的明确定义。

术语含义本身有模糊性，不必孜孜不倦于其中。叫端到端测试也好，广域栈测试也罢，功能性测试也行，都没问题。你认为是集成测试，可能和其他公司的人的认知也不同，这也没问题。当然，如果业界能有一些定义明确的术语并统一语言，那是再好不过了。可惜的是这件事尚未发生。而且在编写测试时会有很多细微差别，它们的范围更像是互相重叠而不是互相离散的，这使得保持术语的一致性更为艰难。

找到适合你和你团队的术语，这就足够了。清晰理解不同类别测试的区别。团队要在测试命名上保持统一，要为每一类测试划分清晰的范围。只要能在团队内部达成一致（或甚至能上升到组织内部一致），你真的不需要关心别的事情了。[Simon Stewart](https://testing.googleblog.com/2010/12/test-sizes.html) (<https://testing.googleblog.com/2010/12/test-sizes.html>) 在谈到他们在 Google 里的做法时有很好的总结。这篇文章完美展示了，为什么过于纠结名称和命名习惯本身不太值得。

把测试放到你的部署流水线上

如果你正在践行持续集成或者持续交付的实践，那么你会有一条部署流水线 (<https://martinfowler.com/bliki/DeploymentPipeline.html>) 来在每一次提交改动时运行自动化测试。通常这个流水线会被分成几个阶段，它们会逐步建立起让你把软件部署到生产环境的自信。听了这么多不同类型的测试，你可能想进一步了解它们在部署流水线中应如何放置。要回答这个答案，你需要思考一下持续交付（实际上是极限编程 (<http://www.extremeprogramming.org/values.html>) 和敏捷软件开发的核心价值之一）的其中一项核心价值观：快速反馈。

避免测试重复

现在你已经理解了为何你需要为软件编写不同类型的测试，但是这还有一个陷阱你需要避开：金字塔不同层级进行了重复测试。虽然你本会说测试太多没啥问题，但我向你保证，会有问题。测试组合中每一个测试都有一定的成本，它们不是免费的。编写和维护测试都要花费时间。阅读和理解其他人写的测试也要花时间。当然，运行这些测试也要花时间。

对于产品代码，你应该力争简洁，杜绝重复。在实现测试金字塔时，你也应该牢记这两条基本法则：

1. 如果一个更高层级的测试发现了一个错误，并且底层测试全都通过了，那么你应该写一个低层级测试去覆盖这个错误
2. 竭尽所能把测试往金字塔下层赶

第一条法则很重要，这是因为低层级测试让你更容易缩小错误的范围，并且隔离掉大部分上下文把错误重现。在调试手头上的问题时，低层级测试运行起来更快，没有太多冗余的东西。同时它们也是很好的回归测试。第二条法则很重要，它能保持测试组合快速运行。如果你已经在低层级测试里覆盖了所有情况，那么再维护一个高层级的测试就没有必要了。因为这并不能给你就软件的工作带来更多信心。如果有许多无效的测试，它也会让你的日常工作很恼火。测试组合会因此拖慢节奏，当你改变代码行为时就需要改更多的测试。

或者让我们这样总结一下：如果写一个更高层级测试能给你带来更多的信心，那就写高层的测试。给一个 Controller 类写单元测试可以测试它内部的逻辑。不过它还是没法告诉你这个 Controller 提供的 REST 路径是否能真正响应 HTTP 请求。那你可以上移一下测试层级，多写一个测试来测试这个点——就测这个点，不能更多了。你不需要再测试所有的条件分支和边缘场景，因为低层级测试已经覆盖到了。保证高层级测试仅仅关注低层级测试覆盖不到的地方。

我对于失去了价值的测试很严格，必须消灭掉它们。我会删掉已经被低层级测试覆盖完全的高层级测试（考虑到它们不会再提供额外的价值了）。我尽可能用低层级测试取代高层级测试。有时候，这会有些困难，尤其是你知道设计测试本身就很艰难。警惕沉没成本的思维陷阱，果断摁下删除键。没有理由在不再提供价值的测试上浪费宝贵时间。

译者注：沉没成本 (<https://zh.wikipedia.org/wiki/%E6%B2%89%E6%B2%A1%E6%88%90%E6%9C%AC>) 是一个经济学概念，沉没成本谬误在这里指的是不忍删除花了时间精力撰写的测试

整洁测试代码

和写代码一样，良好整洁的测试代码同样需要悉心照料。在你于自动化测试之路上继续前进之前，我有些写好可维护测试代码的窍门想告诉你：

1. 测试代码跟生产代码一样重要。要对它们赋予同等的关注和照顾。“这只是测试代码”不能成为你写出邋遢代码的借口
2. 一个测试只测试一个分支。这能帮你的测试保持短小，容易理解
3. “arrange, act, assert”或者“given, when, then”等口诀有助于你写结构良好的测试
4. 可读性很重要。不要过于追求 DRY。如果能提高可读性，重复有时候也是可以接受的。尝试在 DRY 和 DAMP (<https://stackoverflow.com/questions/6453235/what-does-damp-not-dry-mean-when-talking-about-unit-tests>) 之间寻找好平衡
5. 如果对于重复有疑惑，试试用 Rule of Three (<https://blog.codinghorror.com/rule-of-three/>) 法则来决定是不是要重构。重构之前先试用一下

结论

好了！我知道这是一篇非常漫长并且艰深的文章，它解释了为什么我们需要测试，以及如何对软件进行测试的问题。好消息是，这篇文章提供的信息经得起时间推敲，无论你在构建什么样的软件都能适用。不管你是工作在一个微服务项目上，还是 IoT 设备上，抑或是手机应用或者网页应用，这篇文章提供的观点应该都有章可寻。


我希望这篇文章能对你有些帮助。有兴趣你可以去 示例代码 (<https://github.com/hamvocke/spring-testing>) 看看，把这里介绍的一些概念纳入到你的测试组合中。想拥有一套稳固的测试组合确实需要付出努力。但长远来看，它们是会给你回报的，它们会给作为开发者的你带来更多清静。相信我。

致谢

感谢 Clare Sudbery, Chris Ford, Martha Rohte, Andrew Jones-Weiss David Swallow, Aiko Klostermann, Bastian Stein, Sebastian Roidl 及 Birgitta Böckeler 为本文的早期手稿提供反馈和建议。感谢 Martin Fowler 的建议，洞见和支持。

更多精彩洞见，请关注微信公众号：思特沃克

[_\(/#wechat\)](#) [_\(/#sina_weibo\)](#) [_\(/#evernote\)](#) [_\(/#pocket\)](#) [_\(/#instapaper\)](#) [_\(/#email\)](#) [_\(/#linkedin\)](#)
[_\(/#pinterest\)](#)

 [0](https://www.addtoany.com/share#url=https%3A%2F%2Finsights.thoughtworks.cn%2Fpractical-test-pyramid%2F&title=%E6%B5%8B%E8%AF%95%E9%87%91%E5%AD%97%E5%A1%94%E5%AE%9E%E6%88%98) [\(https://www.addtoany.com/share#url=https%3A%2F%2Finsights.thoughtworks.cn%2Fpractical-test-pyramid%2F&title=%E6%B5%8B%E8%AF%95%E9%87%91%E5%AD%97%E5%A1%94%E5%AE%9E%E6%88%98\)](https://www.addtoany.com/share#url=https%3A%2F%2Finsights.thoughtworks.cn%2Fpractical-test-pyramid%2F&title=%E6%B5%8B%E8%AF%95%E9%87%91%E5%AD%97%E5%A1%94%E5%AE%9E%E6%88%98)

Posted in: [软件测试](https://insights.thoughtworks.cn/category/testing/) (https://insights.thoughtworks.cn/category/testing/). Tagged: [测试](https://insights.thoughtworks.cn/tag/testing/) (https://insights.thoughtworks.cn/tag/testing/), [测试金字塔](https://insights.thoughtworks.cn/tag/%E6%B5%8B%E8%AF%95%E9%87%91%E5%AD%97%E5%A1%94/) (https://insights.thoughtworks.cn/tag/%E6%B5%8B%E8%AF%95%E9%87%91%E5%AD%97%E5%A1%94/).

发表评论

电子邮件地址不会被公开。 必填项已用*标注

评论

姓名 *

电子邮件 *

站点

发表评论

此站点使用Akismet来减少垃圾评论。 [了解我们如何处理您的评论数据](https://akismet.com/privacy/) (https://akismet.com/privacy/)。