



下载APP



11 | 集成测试：单元测试可以解决所有问题吗？

2021-08-27 郑晔

《程序员的测试课》

课程介绍 >



讲述：郑晔

时长 12:51 大小 11.77M



你好，我是郑晔！

前面我们花了大量的篇幅在讲单元测试。单元测试是所有测试类型中，运行速度最快，整体覆盖面可以达到最大的测试。因为单元测试的存在，我们甚至可以把测试覆盖率拉高到 100%。测试覆盖率都已经 100% 了，是不是我们用单元测试就可以解决所有的问题了？

正如我们在上一讲强调的那样，100% 的测试覆盖率并不代表代码没有问题。同样，即便是 100% 的单元测试也不能解决所有的问题。有一个重要的原因在于，我们在编写每个单元时都会假设这些单元彼此之间能够很好地协同，但这个假设是不是一定成立呢？答案不一定。



让一个个单元正常运行，我们靠的不是美好的预期，而是单元测试。同样，各个单元能够很好地协同，我们也不能靠预期，而是要靠集成测试。这一讲，我们就来讨论一下集成测试。

代码的集成

在具体讨论集成测试之前，我们澄清一下概念。集成测试到底测的是什麼？答案很显然是集成。问题是，集成要集成什麼呢？一种是代码之间的集成，一种是代码与外部组件的集成。说白了，集成测试就是把不同的组件组合到一起，看看它们是不是能够很好地配合到一起。

我们先来看代码的集成。代码之间的集成，主要看我们编写的各个单元能否很好地彼此协作。既然集成测试是为了测试单元之间的配合，那是不是只要有单元之间的协作，我们就要为它们编写一个集成测试呢？比如按照常规的架构分层，一个 REST 服务会有一个 Resource（或者叫 Controller），一个 Service，一个 Repository，那是不是要 Service 和 Repository 的集成要写一个集成测试，Resource 和 Service 的集成测一次，Resource、Service 和 Repository 的集成再测一次呢？

如果我们按照前面讨论的方式来编写了单元测试，其实，这就意味着我们每个组件都已经经过了测试。所以，集成测试的重点就不再是组件之间两两协同进行测试了。一般来说，在实践中，我们可以选择的测试方式是，**选择一条任务执行的路径，把路径上用到的组件集成到一起进行测试**。比如在前面提到的那种情况中，我们只要把 Resource、Service 和 Repository 都组装到一起就可以了。

如果所有的代码都是我们自己编写，那么我们就编写一个个的单元，然后组装到一起进行测试，这个很好理解。但是，现在很多人都在使用框架，比如我们在实战中处理命令行时使用了 Picocli 这个框架，所有的命令解析的过程都是由这个框架完成；再比如，很多人在开发后端服务时，使用了 Spring Boot，一些路由匹配，甚至参数检查都是由框架完成的。那么我们在集成测试中，要不要把这个部分集成进来呢？

我对此的答案是，**取决于你是否能够把这个框架集成进来，如果能，最好是做一个完整的集成测试**。在实战中，我们已经展示过如何去集成 Picocli，因为这个框架本身比较简单，很容易找到这个框架的外部入口，我们就把它集成起来，做了一个完整的测试。

有的框架可能就没有那么简单了，就像当年 Java EE 盛行时，我们编写的代码需要部署到一个 Java EE 的容器里面才能运行。在这种情况下，如果强行把 Java EE 容器也加到集成测试里，对于大多数人来说，这是非常有难度的一件事情。换言之，像这种有单独运行时的框架，做整体的集成难度很大，我们只能退而求其次，做大部分的代码集成。

现在的很多框架替我们做了很多的事情，有些甚至是业务验收标准上的事情，比如，Spring Boot 会替我们做参数检查，利用好 Spring Boot 给我们提供的机制，我们甚至不用写什么代码，只要给字段加上一些 Annotation 就够了。这些 Annotation 加的是否正确，我们其实是需要验证的，因为它是业务验收标准的一部分。

所以我希望尽可能地去集成，如果我们能够把整个框架集成起来，这些东西也就可以验证了。从代码上来看，这种测试只是针对一个单元在测试，在某种程度上说，这种集成测试其实是一种单元测试，只不过，它必须把系统集成起来才行，所以，它兼具单元测试和集成测试的特点。

小小预告一下，Spring Boot 在测试上的支持是真的很不错，让我们可以很容易地在测试里对框架处理过程进行集成，在后面的课程里你会看到如何使用 Spring Boot 提供的测试基础设施进行测试。

你也看到了，我们希望尽可能地把框架集成进来，但市面上的各种框架层出不穷，不是所有的框架都提供了对测试很好地支持。所以，**一个框架设计得好坏与否，对测试的支持程度也是一个很重要的衡量标准，这能很好地体现出框架设计者的品味。**

能够方便测试的框架，通常来说都是很轻量级的，这样的框架对开发非常友好，我们能够在普通的 IDE 里很方便地进行调试，对于定位问题也是极其友好的。而各种有运行时需要部署的框架，相对来说，就是重量级的框架，对于开发非常不友好。如果你用过一些 IDE 支持的远程调试功能，你会发现这些功能跟本地调试相比，便捷程度完全不在一个档次上。

好消息是，我们还是能看到一些框架的进步，即便重如 Java EE 这样的框架，现在也有了嵌入式容器的概念。今天，我们之所以能够很方便地使用 Spring Boot 这样的框架，嵌入式容器给我们提供了非常好的基础。

集成外部组件

说完了代码的集成，我们再来看看与外部组件的集成。

在真实世界的开发中，我们经常会遇到与外部组件打交道的情形，最简单是数据要写到数据库里，还有发消息可能会用到消息队列，甚至还可能会涉及与第三方系统的集成。

理想情况下，我们当然希望把所有相关的组件都集成到一起，但是，一旦牵扯到外部组件，测试的难度立刻就增大了。比如在测试中添加了 Todo 项，如果我的断言写的是先判断数据库里 Todo 项表里有唯一的一条记录，执行之前，你因为其它操作在数据库里插入了数据，这个断言就失败了。即便没有人操作，这个测试执行第一次成功了，再执行一次，可能就失败了，因为第二次执行测试又插入了一条数据。

所以，**与外部组件集成，难点就在于外部组件的状态如何控制。**

如果能够控制外部组件的状态，在系统里集成它是没有问题的。比如拿数据库集成来说，通常的做法是一方面，我们会建立单独的数据库，保证不与其他部分冲突。比如在 MySQL 里面，我们会建立一个测试用的数据库。

 复制代码

```
1 CREATE DATABASE todo_test;
```

另一方面，我们要保证它在每个测试之后，都能够恢复到之前的状态。一种做法就是使用数据库的回滚技术，每个测试完成之后就回滚掉，保证数据的干净。后面讲到 Spring Boot 测试的时候，我们会看到具体的做法。

相对来说，数据库在测试方面的实践已经算是比较成熟了。这也让我们可以去验证 Repository，也就是数据访问层的代码实现。不管使用什么样的框架，写了 SQL 之后，我们都需要验证其正确性。只不过，很多人的选择是把整个系统跑起来，人工去验证 SQL 的正确性，这种做法一方面有些小题大做了，另一方面还是不够自动化。

有了数据库在测试上的实践，我们就可以用自动化测试的方式进行测试了。其实，从某种意义上说，这也是一种单元测试，因为它的代码只涉及到了一个单元，只不过它需要集成数据库，所以，它还是集成测试

还有一些外部组件在这方面的支持相对来说，就不那么令人满意了。比如第三方系统。即便是服务做得很完善的第三方系统，也很少有专门为测试提供支持的。

遇到这种情况，我们就要分析一下，看看有没有什么替代方案。很多第三方系统对外提供服务的方式都是 REST API，对于这种情况，我们就可以用通用的模拟服务器来代替。模拟服务器的价值就在于能够替代这样的第三方服务。

在这种情况下，我们该怎么做呢？我们需要按照我们的使用场景去访问第三方服务，把整个访问的报文记录下来，作为设置模拟服务器的参考依据。我介绍过的 Moco 甚至提供了代理（proxy）功能，你可以让你的服务去连接 Moco，然后，用 Moco 连接第三方的服务，只要查看 Console 输出，所有的报文就清清楚楚地展现在你面前了。

如果外部组件没有现成的替代方案怎么办？有两个角度看待这个问题。一个角度是，这也许是一个做新项目的机会，我在《软件设计之美》中讲过 [Moco](#) 的开发过程，其起始点就是一个没有很好解决的问题。

另一角度，估计是大多数人的选择，那就是既然这里测不了，我可以选择在集成测试里使用模拟对象，而不是真实的对象。在这样的情况下，我们的系统在测试方面其实有一个漏洞没有被测试很好地覆盖。也就是说，我们要把这个漏洞留到更上一层的测试。如果这个漏洞是一个简单的逻辑（比如一个消息队列发消息的接口），这样还好。如果里面有逻辑，我们必须把它作为一个重点的风险提示加以重视。不过，好在这种情况下并不是很多，毕竟像 SQL 这种有复杂逻辑的东西，我们已经有了解决方案。

总结时刻

今天我们讲了集成测试，相对于单元测试只关注单元行为，集成测试关注的多个组件协同工作的表现。今天我们讨论了两类典型的集成问题，一种是代码之间的集成，一种是代码与外部组件的集成。

对代码之间的集成来说，一方面要考虑我们自己编写的各个单元如何协作；另一方面，在使用各种框架的情况下，要考虑与框架的集成。如果我们有了单元测试，这种集成主要是关心链路的通畅，所以一般来说我们只要沿着一条执行路径，把相关的代码组装到一起进行测试就可以了。

如果涉及框架，最好是能够把框架集成一起做了，设计得比较好的框架是对于测试的支持比较好的（比如像 Spring Boot），可以让我们很方便地进行测试。

对于外部组件的集成而言，难点在于如何控制外部组件的状态。数据库在这方面相对已经有比较成熟的解决方案：使用单独的数据库，以及在测试结束之后进行回滚。

但大部分系统没有这么好的解决方案，尤其是第三方的服务。这时候，我们就要看有没有合适的替代方案。对于大多数 REST API，我们可以采用模拟服务器对服务进行模拟。

通过今天的讨论你会发现，严格地说，有些代码由于基础设施的问题是不容易在自动化场景覆盖的，这也是我们为什么要强调与框架结合的代码一定要薄，让这种代码的影响尽可能少。这也是在减少用上层测试覆盖的工作量。

到这里，大部分的场景我们都已经可以用自动化测试进行覆盖了，我们对自己的系统已经有了更完整的理解。其实，测试的种类还有更多，比如系统测试，把整个系统集成起来测试；验收测试，交由业务人员或测试人员进行测试。但这些测试对于很多团队来说，已经到了测试人员的工作范畴了。作为程序员，我们能够把单元测试和集成测试做好，整个软件的质量已经是初步合格了。

如果今天的内容你只能记住一件事，那请记住：**想办法将不同组件集成起来进行测试。**

思考题

今天我们讲了集成测试，你也看到了集成测试难点就在于如何集成。在实际工作中，你遇到过哪些难以在测试中集成的情况吗？欢迎在留言区分享你的经验。

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 赞 1  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 10 | 为什么 100% 的测试覆盖率是可以做到的？

精选留言 (1)

写留言



grandgraph 
2021-08-27

如果一个用例需要集成非常多的外部依赖才能做集成测试, 是不是也说明了这个用例的实现本身可能有问题, 比如非核心流程可以考虑用异步消息进行解耦, 以减少依赖?

作者回复: 它好像是系统测试

