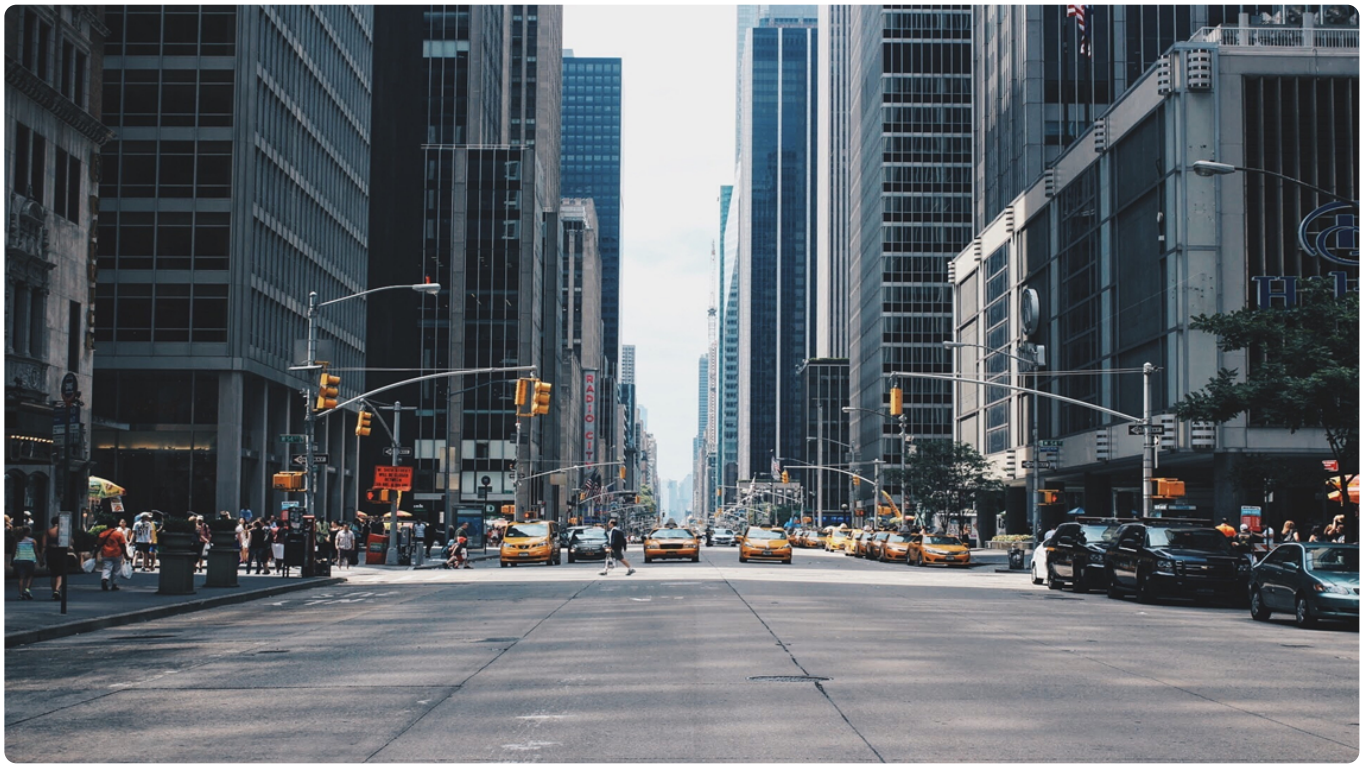
06 | 测试不好做，为什么会和设计有关系？

2021-08-16 郑晔

《程序员的测试课》

课程介绍 >



讲述：郑晔

时长 13:36　大小 12.46M

你好，我是郑晔！

在前面几讲里，我们讲了测试的一些基础，要有测试思维、要学会使用自动化测试框架、要按照好测试的样子去写测试……但是，懂了这些就能把测试写好吗？

答案显然是否定的。因为这些东西很多人都知道，但真正到了实际的项目中面对自己的代码，还是不会写测试。主要的问题就是不好测，这也是测试劝退了很多程序员的一个重要原因。

不好测实际上是一个结果。那造成这个结果的原因是什么呢？答案就是我们今天要讨论的话题：软件设计。

可测试性

为什么说不好测是由于软件设计不好造成的呢？其实，更准确的说法是绝大多数人写软件就没怎么考虑过设计。

软件设计是什么？软件设计就是在构建模型和规范。

然而，大多数人写软件的关注点是实现。我们学习写程序的过程，一定是从实现一个功能开始的。这一点在最开始是没有问题的，因为需求的复杂度不高。不过需求一旦累积到一定规模，复杂度就会开始大幅度升高，不懂软件设计的人就开始陷入泥潭。

即便一个人认识到软件设计的重要性，学习了软件设计，但在做设计的时候还是常常会对**可测试性**考虑不足。可测试性是一个软件 / 模块对测试的支持程度，也就是当我执行了一个动作之后，我得花多大力气知道我做得到底对不对。

我们所说的代码不好测，其实就是可测试性不好。当我们添加了一个新功能时，如果必须把整个系统启动起来，然后给系统发消息，再到数据库里写 SQL 把查数据去做对比，这是非常麻烦的一件事。为了一个简单的功能兜这么大一圈，这无论如何都是可测试性很糟糕的表现。然而，这却是很多团队测试的真实状况。因为系统每个模块的可测试性不好，所以，最终只能把整个系统都集成起来进行系统测试。

如果建楼用的每块材料都不敢保证质量，你敢要求最终建出来的大楼质量很高吗？这就是很多团队面临的尴尬场景：每个模块都没有验证过，只知道系统集成起来能够工作。所以，一旦一个系统可以工作了，最好的办法就是不去动它。然而，还有一大堆新需求排在后面。

相应地，**对一个可测试性好的系统而言，应该每个模块都可以进行独立的测试。**在我们把每一个组件都测试稳定之后，再把这些组件组装起来进行验证，这样逐步构建起来的系统，我对它的质量是放心的。即便是要改动某些部分，有了相应的测试做保证，我才敢于放手去改。

可测试性很重要，但我要怎么让自己的代码有可测试性呢？

编写可测试的代码

编写可测试的代码，最简单的回答就是**让自己的代码符合软件设计原则。**在《软件设计之美》的专栏里，我专门讲了 SOLID 原则，这是目前软件设计中最成体系的一套设计原则。如果代码真的能做到符合 SOLID 原则，那它基本上就是可测的。

比如，符合单一职责原则的代码一般都不会特别长，也就没有那么多的分支路径，相对来说就比较容易测试。再比如，符合依赖倒置原则的代码，高层的逻辑就不会依赖于底层的实现，测试高层逻辑的部分也就可以用 Mock 框架去模拟底层的实现。

编写可测试的代码，如果只记住一个通用规则，那就是**编写可组合的代码。**什么叫可组合的代码？就是要能够像积木一样组装起来的代码。

既然要求代码是组装出来的，由此得出的**第一个推论是不要在组件内部去创建对象。**比如，我们在前面的实战中有一个 TodoItemService，它有一个 repository 字段。这个字段从哪来呢？直接创建一个实例理论上是可以的，但它会产生耦合。根据我们的推论，不要在组件内部创建对象，所以，我们考虑从构造函数把它作为参数传进来。

复制代码

```
public class TodoItemService {
    private final TodoItemRepository repository;

    public TodoItemService(final TodoItemRepository repository) {
        this.repository = repository;
    }
    ...
}
```

你或许会问了，如果不在内部创建对象，那谁来负责这个对象的创建呢？答案是组件的组装过程。组件组装在 Java 世界里已经有了一个标准答案，就是依赖注入。

不在内部创建，那就意味着把组件的组装过程外置了。既然是外置了，组装的活可以由产品代码完成，同样也可以由测试过程完成。

站在测试的角度看，如果我们需要测试 TodoItemService 就不需要依赖于 repository 的具体实现，完全可以使用模拟对象进行替代。

我们可以完全控制模拟对象的行为，这样，对 TodoItemService 的测试重点就全在 TodoItemService 本身，无需考虑 repository 的实现细节。在实战的过程中你也看到了，我们在实现了 TodoItemService 时，甚至还没有一个 repository 的具体实现。

现在你知道了，编写可组合的代码意味着，我们把组件之间的关联过程交了出去，组件本身并不会去主动获取其相关联组件的实现。由此，我们要得出**第二个推论：不要编写 static 方法。**

我知道很多人喜欢 static 方法，因为它用起来很方便，但对测试来说却不是这样。使用 static 方法是一种主动获取的做法。一旦组件主动获取，测试就没有机会参与到其中，相应地，我们也就控制不了相应的行为，测试难度自然就增大了。所以，如果团队需要有一个统一约定，那就是不使用 static 方法。

如果非要说有什么特例，那就是编写一些基础库（比如字符串处理等），这种情况可以使用 static 方法。但基本上大部分程序员很少有机会去写基础库，所以，我们还是把不编写 static 方法作为统一的原则。

如果你能够摒弃掉 static 方法，**还有两样东西你也就可以抛弃了，一个是全局状态，一个是 Singleton 模式。**

如果你的系统中有全局状态，那就会造成代码之间彼此的依赖：一段代码改了状态，另一端代码因为要使用这个状态而崩溃。

但如果我们抛弃了 static 方法，多半你也就没有机会使用全局状态了，因为直接访问的入口点没有了。如果需要确实有状态，那就可以由一个组件来完成，然后，把这个组件注入进来。

如果你能够理解 static 方法的问题，你也就能够理解 Singleton 模式存在的问题了。它也是一样没有办法去干涉对象的创建，而且它本身限制了继承，也没有办法去模拟。

你或许已经意识到了，之所以说编写可组合的代码是可测试性的关键，是因为我们在测试的过程中要参与到组件的组装过程中，我们可能会用模拟对象代替真实对象。模拟对象对我们来说是完全可控的，而真实对象则不一定那么方便，比如真实对象可能会牵扯到外部资源，带来的问题可能比解决的问题更多。

要使用模拟对象，就要保证接口可继承，函数可改写，这也是我们对于编写可测试代码的一个要求。所以，这又回到了设计上，要想保证代码的可测试性，我们就要保证代码是符合面向对象设计原则的，比如要基于行为进行封装等等。

与第三方代码集成

如果说前面讨论的内容更多的是面向自己写的代码，那在实际工作中，我们还会面临一个真实的问题，就是与第三方的代码集成。无论是用到开源的程序库，还是用到别人封装好的代码，总之，我们要面对一些自己不可控的代码，而这些代码往往也会成为你编写测试的阻碍。

对于测试而言，第三方的代码难就难在不可控，要想让它们不再成为阻碍，就要让它们变得可控。

如何让第三方代码可控呢？答案就是隔离，也就是将第三方代码和我们自己编写的业务代码分开。如何隔离呢？我们分成两种情况来讨论。

调用程序库

第一种情况是我们的代码直接去调用一个程序库。在实际工作中，这应该是最广泛的使用场景，可能是对一个协议解析，也可能调用一个服务发送通知。

在实战的例子中，我们也曾经调用 Jackson 去实现 JSON 的处理。那个例子就表现了一个典型的第三方代码不可控，它抛出的异常我们不好去模拟，所以，很难用测试去覆盖。不过，因为那个例子比较特殊，算是基础库的范畴，我们就直接封装成 static 方法了。

在大部分的情况下，**我们做代码隔离，需要先定义接口，然后，用第三方代码去做一个相应的实现。**比如，我们在实战中定义过一个 TodoItemRepository，当时给的实现是一个基于文件的实现。

复制代码

```
interface TodoItemRepository {
  ...
}

class FileTodoItemRepository implements TodoItemRepository {
  ...
}
```

如果我们要把数据存到数据库里，那我们就可以给出一个数据的实现。

```
复制代码
1 class DbTodoItemRepository implements TodoItemRepository {
2   ...
3 }
```

而要存到云存储，就写一个云存储的实现。

```
复制代码
1 class S3TodoItemRepository implements TodoItemRepository {
2   ...
3 }
```

这里的关键点是定义一个接口，这个接口是高层的抽象，属于我们业务的一部分。但要使用的第三方代码则属于一个具体的实现，它是细节，而不是业务的一部分。如果熟悉软件设计原则，你已经发现了，这其实就是依赖倒置原则。

有了这层隔离之后，我们就可以竭尽全力地把所有的业务代码用测试覆盖好，毕竟它才是我们的核心。

由框架回调

我们再来看与第三方代码集成的另外一种情况，由框架回调。比如，我们在实战里面用到了一个处理命令行的程序库 Picocli，它会负责替我们解析命令行，然后，调用我们的代码，这就是一个典型的由框架回调的过程。

这种情况在使用一些框架时非常常见，比如，使用 Spring Boot 的时候，我们写的 Controller 就是由框架回调的。使用 Flink 这样的大数据框架时，我们写的代码最终也是由框架回调的。

不同的框架使用起来轻重是不同的，比如在实战中，我们就直接触发了 Picocli，因为它本身比较轻量级；而像 Flink 这样的大数据框架想要在本地运行就需要做一些配置。

总而言之，要想测试使用了这些框架的程序，多半就是一种集成测试，而集成测试相对于单元测试来说，是比较重的，启动配置比较麻烦，运行时间比较长。

如果应用能在一个进程中启动起来，这还是好的情况。我还依然记得当年 Java 的主流开发方式是部署到应用服务器上，每次打包部署都是一个让人痛苦不堪的过程。像今天本地能够启动一个 Spring Boot 进程，这完全是需要感谢嵌入式 Web 服务器的发展。

面对这样的框架，我们有一个统一的原则：**回调代码只做薄薄的一层，负责从框架代码转发到业务代码。**

我们在实战的代码中已经见到了，比如，下面这段代码是添加一个 Todo 项的实现。

复制代码

```
@CommandLine.Command(name = "add")
public int add(@CommandLine.Parameters(index = "0") final String item) {
  if (Strings.isNullOrEmpty(item)) {
    throw new CommandLine.ParameterException(spec.commandLine(), "empty item i
  }
  final TodoItem todoItem = this.service.addTodoItem(TodoParameter.of(item));
  System.out.printf("%d. %s%n", todoItem.getIndex(), todoItem.getContent());
  System.out.printf("Item <%d> added%n", todoItem.getIndex());
  return 0;
}
```

这里面的核心代码就一句话，剩下的要么是做校验，要么是与框架的交互，几乎没有太多逻辑可言。

复制代码

```
this.service.addTodoItem(TodoParameter.of(item));
```

正如你在实战过程中见到的那样，我会先编写自己的业务核心代码，而把与框架接口的部分放到了后面去编写。因为最容易出问题的地方往往不是在这个转发的过程，而是业务的部分。只有当你的业务代码质量提升了，整个系统的质量才会得到真正的提升。所以，如果你看到这一层写了很多的代码，那这段代码一定是有坏味道了。

或许你也发现了，这其实也是一种常见的模式：防腐层。是的，不仅仅是我们与第三方系统交互有防腐层，与外界的交互同样需要防腐层。

你也看到了无论是调用程序库还是由框架回调，说来说去，都会回到软件设计上。所以，一个可测试的系统，关键要有一个好的设计。**想要写出高质量的代码，软件设计就是程序员必备的一项能力**。

通过软件设计，我们将业务代码同一些实现细节分离开来。但如果我们在测试中使用同样的实现，结果必然是把复杂性又带回来了。那不用同样的实现该怎么测试呢？下一讲，我们就来说说，怎么在测试中给出一个可控的实现。

总结时刻

这一讲，我们讲了软件的可测试性，这是影响到一个系统好不好测的一个重要因素。可测试性好的软件，各个模块都可以独立测试。而可测试性不好的软件，只能做整体的测试，其复杂度和过程中花费的时间都是不可同日而语的。

提升软件的可测试性，关键是改善软件的设计，编写可测试的代码。关于如何编写可测试的代码，我给了一个路标：编写可组合的代码。从这个路标出发，我们得出了两个推论：

不要在组件内部创建对象；

不要编写 static 方法。

由不编写 static 方法，我们可以推导出：

不要使用全局状态；

不要使用 Singleton 模式。

在实际工作中，除了要编写业务代码，还会遇到第三方集成的情况：

对于调用程序库的情况，我们可以定义接口，然后给出调用第三方程序库的实现，以此实现代码隔离；

如果我们的代码由框架调用，那么回调代码只做薄薄的一层，负责从框架代码转发到业务代码。

如果今天的内容你只能记住一件事，那请记住：**编写可测试的代码。**

思考题

今天我们讲了代码中不好测的情况主要是由于软件设计不好造成的。在实际的工作中，你还有遇到过哪些不好测的情况呢？欢迎在留言区分享你的经验。

上一篇 05 | 一个好的自动化测试长什么样？

下一篇 07 | Mock 框架：怎么让测试变得可控？

精选留言 (4)

写留言

亦无
2021-08-18

软件设计本身就是一个很重要的事情，但是大家都知道重要，落实的时候并不完全都按照设计原则来进行实现，加上所有项目都在赶工期，大家就真的完全关注实现了，先提测再说，成了首要目的。

这次通过老师说的可测试性要求，让软件设计的重要性再次提升，其实软件设计做好了...

展开

1

grandgraph
2021-08-18

对于不是以面向对象范式为核心的编程语言 (比如go), 需要做出一些针对性的调整吗? 在go语言中写出function主导的过程式代码还是比较普遍的.

展开

¦并向你招手¦祥子

2021-08-17

目前团队在使用sonar 作为代码质量的管理工具，其中有一条规则，没有属性依赖的方法应该是static 方法，但这种static方法实际上并没有为测试增加障碍，反而是更容易写测试的，不需要任何外部依赖，也不用做测试准备，连实例化都不用，直接调用对输入输出进行检查即可

展开

作者回复: 这种情况和我说的属于基础库是类似的，大部分人其实是很少有机会写这样的代码，这也是我建议从整体上规避写static的原因。

lanlyhs

2021-08-16

赞，老师为单元测试的痛点指出了明路。

我们的系统现在全是 static 方法.... 只能在外围做一些接口测试 ，非常痛苦。

作者回复: 听上去就很痛苦