04 | 自动化测试：为什么程序员做测试其实是有优势的？

2021-08-11 郑晔

《程序员的测试课》 课程介绍 >

讲述：郑晔
时长 12:18 大小 11.27M

你好，我是郑晔！

在上一讲里，我们讨论了程序员做测试和测试人员做测试之间有什么不同，你现在应该不会担心因为程序员做测试就抢了测试人员的饭碗了。这一讲，我们来谈谈程序员做测试的优势所在。估计你已经想到了，没错，就是自动化测试。

其实，程序员的主要工作就是打造各种自动化工具，无论是为了业务的支撑，或者是对于开发过程本身的支持。自动化，一方面是为了提高效率，另一方面，也是将低效繁琐的工作交由机器去完成。（关于自动化的种种思考，我在《10x 程序员工作法》中有了一模块进行讲解，如果你有兴趣不妨去回顾一下。）

测试这种工作其实非常适合自动化，因为在整个软件的生命周期之内，新的需求总会不断出现，代码总会不断地调整。鉴于大部分软件常常都是牵一发动全身，所以，即便是只改动了一点代码，理论上来说也应该对软件的全部特性进行完整验证。如果只靠人工来做这个事情，这无疑是非常困难的。

很多团队只依赖于测试人员进行测试，而且测试以手工为主，结果就是大部分时间都是在进行低效地验证工作，而这些工作恰恰是最适合用自动化测试完成的。

从自测到自动化测试框架

你平时是怎么验证自己代码正确性的呢？最不负责任的做法是压根不验证，我曾见过最极端的做法是连编译都不通过的代码就直接提交了。不过，这是我职业生涯早期发生的事情。随着行业整体水平的提高，这种事情现在几乎看不到了。

现在很多人的做法是把整个系统启动起来，然后手工进行验证。当然，大多数人不会验证系统里面所有的内容，只会针对自己正在开发的部分进行验证。这种做法通常只能够保证自己刚刚编写的代码是正确的。结果常常是按下葫芦浮起瓢——这个功能是对了，但之前原本验证好的功能又不对了。

即便是一个再小的系统，其中的细节也多到没有人愿意每次去手工验证其中所有的细节。因为这样做既琐碎又重复，这显然是适合自动化发挥战斗力的地方。

最开始的自动化都是很简单的。通常来说，就是直接写一个 main 函数，直接调用代码中的模块。但每次要测试不同的代码时，程序员就要注释掉原来的测试代码，然后，再编写新的测试代码。

这种做法虽然可以去验证代码的正确性，但显然不适合反复验证。稍微优化点的做法就是把一个个测试用例放到不同的函数里。总的来说，这个阶段的自动化测试还处于草莽阶段。

真正让自动化测试这件事登堂入室的，就是自动化测试框架了。最早的测试框架起源是 Smalltalk 社区。Smalltalk 是一门早期的面向对象程序设计语言，它有很多拥趸，很多今天流行的编程概念都来自于 Smalltalk 社区，自动化测试框架便是其中之一。

不过，真正让测试框架广泛流行起来，要归功于则另外的自动化测试框架 JUnit，它的作者是 Kent Beck 和 Erich Gamma。Kent Beck 是极限编程的创始人，在软件工程领域大名鼎鼎，而 Erich Gamma 则是著名的《设计模式》一书的作者，很多人熟悉的 Visual Studio Code 也有他的重大贡献。

有一次，Kent Beck 和 Erich Gamma 一起从苏黎世飞往亚特兰大参加 OOPLSA（Object-Oriented Programming, Systems, Languages & Applications）大会，在航班上两个人结对编程写出了 JUnit。从这个名字你不难看出，它的目标是打造一个单元测试框架。二人之所以能够在一路上就完成 JUnit 最初版本的开发，是因为他俩本身就在 Smalltalk 社区摸爬滚打了一段时间，对 Smalltalk 的单元测试框架有着很深刻的认识。

今天流行的自动化测试框架统称为 xUnit，因为它们都有一个共同的根基，也就是 JUnit。所以，只要了解了 JUnit 中的基本概念，你再去看其它测试框架，几乎都是差不多的。

测试框架简介

接下来，我们就来一次快速的自动化测试框架简介，如果你已经对自动化测试框架非常熟悉的话，可以当做一次轻松的复习。

我们理解测试框架有两个关键点，一是要去理解测试组织的结构，一是要去理解断言。掌握了这两点，就足够应付日常的大多数情况了。

测试结构

我们先来看看组织测试的结构。首先，最核心的概念就是怎么表示一个测试用例。JUnit 怎么表示测试用例，我们在前面讲实战的时候已经见识过了，代码如下所示。

复制代码

```
@Test
public should_work() {
  ...
}
```

我们前面说过，草莽阶段稍微优化一点的做法就是把测试用例放到一个个不同的函数里面，而测试框架就是把这种做法做了一个延伸，同样是用一个一个的函数表示一个一个的

测试用例。不同的是，在草莽阶段，你每写一个函数就要在执行的部分注册一下这个函数。

使用测试框架的话，需要对表示测试用例的函数进行统一的标识，以便框架能够在运行时识别出来。在我们上面这个例子里面，用来识别测试用例的就是 @Test。如果你用过 4.0 之前版本的 JUnit，它是约定以 test 开头的函数就是测试用例，所以，你会看到下面这样的写法。

复制代码

```
public test_should_work() {
  ...
}
```

两种不同的写法本质上是程序设计语言层面的差别，因为 Java 5 引入了 Annotation 这个语法，才有了基于 @Test 进行标注的做法。**很多的语法层面的改进都是为了提升语言的表达能力，而这一点在程序库的设计上体现得最为明显**。如果你去看不同程序语言的测试框架时就会发现，做得比较差就是直接照搬 test 开头的做法，而做得比较好的则是会结合自己的语言特点。

了解了最基本的测试用例结构，其实写测试就够了。但是，测试也是代码，好的测试代码要兼具好代码的属性，最基本的要求就是消除重复。

比如，同样的初始化代码反复在写，由于测试的特殊性，这些初始化的代码需要在每个测试之前都去执行。为了解决这个问题，JUnit 引入了 setUp 去做初始化的工作。在 JUnit 4 之后，这个由函数名称进行定义的做法，改成了使用 @BeforeEach 进行定义的方式。我们在前面的实战中也提到过。

复制代码

```
@BeforeEach
void setUp() {
  ...
}
```

由于 @BeforeEach 的存在，setUp 这个名字在这里已经没有意义，只不过因为这是一个函数，需要有一个名字。从习惯上，我们还是称呼它为 setUp 函数。如果 JUnit 进一步将

语法升级到 Java 8 的语法，这里完全可以使用 lambda，去掉对名字的依赖。

与 @BeforeEach 和 setUp 对应的是 @AfterEach 和 tearDown，它们处理的是要在每个测试之后执行的清理工作。相对来说，这一对用的就比较少了，除非是你用到了一些需要释放的资源。

知道了测试用例的写法，知道了 setUp/tearDown，你就基本上掌握了测试结构的核心了。如果你具体学习一个测试框架，还会有人告诉你 TestSuite、TestRunner 等等的概念，但它们现在基本上可以归入到实现层面了（也就是执行测试所需要了解的概念），而在编写代码的层面上，有前面说到的这几个概念就够了。

断言

我们接下来看理解测试框架的第二个关键点，断言。测试结构保证了测试用例能够按照预期的方式执行，而断言则保证了我们的测试需要有一个目标，也就是我们到底要测什么。

断言，说白了就是拿执行的结果和预期的结果进行比较。如果执行一个测试连预期都没有，那它到底要测什么？所以，我们可以说，**没有断言的测试不是好测试。**

几乎每个测试框架都有自己内建的断言机制，比如下面这个。

```
1 assertEquals(2, calculator.add(1, 1));
```

复制代码

这个 assertEquals 是最典型的一个断言，也几乎是使用最多的断言，很多其它语言的测试框架也把它原封不动地搬了过去。但这个断言有一个严重的问题，你如果不看 API，根本记不住哪个应该是预期值，哪个应该是你函数返回的实际值。这就是典型的 API 设计问题，让人很难用好。

所以，社区中涌现了大量的第三方断言程序库，比如，Hamcrest、AssertJ、Truth。其中，Hamcrest 是一个函数组合风格的断言库，一度被内建到 JUnit 4 里面，但出于对社区竞争的鼓励，JUnit 5 又把它挪了出来，下面是一段使用了 Harmcrest 的代码。

```
assertThat(calculator.subtract(4, 1), is(equalTo(3)));
```

复制代码

AssertJ 是一种流畅风格的程序库，扩展性也非常不错，它也是我们在前面实战部分选择的程序库，下面是一段使用了 AssertJ 的代码。

```
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");
```

复制代码

Truth 是 Google 开源的一个断言库，和 AssertJ 很类似，它对 Android 程序支持得比较好，我也放了一段代码，风格上和 AssertJ 如出一辙。

```
assertThat(projectsByTeam())
    .valuesForKey("corelibs")
    .containsExactly("guava", "dagger", "truth", "auto", "caliper");
```

复制代码

断言，不仅仅包括有返回值的处理，还包括其它的特殊情况，比如，抛出异常也可进行断言，这是 JUnit 5 内建的异常断言，你可以参考一下。

```
Assertions.assertThrows(IllegalArgumentException.class, () -> {
  Integer.parseInt("One");
});
```

复制代码

具体有哪些情况可以进行断言，你可以查阅所使用断言库的 API 文档。

最后，我还要讲一个不在这些断言库里的断言，那就是 Mock 框架提供的一种断言：verify。

关于 Mock 框架，后面我们还会讲到，这里只是简单地提一下，verify 的作用就是验证一个函数有没有得到调用。在某些测试里面，函数既没有返回值，也不会抛出异常。比如拿

保存一个对象来说，我们唯一能够判断保存动作是否正确执行的办法，就是利用 verify 去验证保存的函数是否得到调用，就像下面这样。

```
1 verify(repository).save(obj);
```

复制代码

虽然它不在断言库中，但它确确实实是一种断言，它判断的是一个动作是否得到正确的执行。所以，当我们说一个测试应该包含断言时，有 verify 的情况也算是有断言了。至于怎么用好 verify，我们后面讲到 Mock 框架时再说。

讲过测试结构和断言，我们已经把测试框架的核心内容说完了。但这些只是写测试的基础，要想写好测试，我们还需要对什么样的测试是好的测试有个基本的认识，这就是我们下一讲要讲的内容了。

总结时刻

这一讲，我们讲了程序员在测试上的优势所在，也就是自动化。软件开发本身就是一个不断迭代的过程，对每一次代码的改动来说，理论上就应该把整个系统从头到尾地测一遍。这种工作手工做是非常琐碎的，所以非常适合使用自动化。

验证程序的正确性是程序员的基本工作，不过，很多人的做法还是手工验证。为别人打造自动化工具的人，自己的开发过程还不够自动化，这是很多程序员面对的尴尬。实际上，还有一些人在探索自动化的做法，从最早的 main 函数，到后来的自动化测试框架，就是在这方面一点一点的进步。自动化测试框架的出现，让自动化测试从业余走向了专业。

理解自动化测试框架，主要包含两个部分：组织测试的结构以及断言。组织测试的结构最核心的就是测试用例如何写，以及 setUp 和 tearDown 函数。而断言则是保证了我们测试的目标。断言程序库有很多，你可以根据自己的喜好进行选择。除了断言程序库，Mock 框架的 verify 也是一种断言。

如果今天的内容你只能记住一件事，那请记住：**没有断言的测试不是好测试。**

思考题

今天我们讲了自动化测试框架最核心的部分，但现在的测试框架都已经有了更多丰富的功能，希望你找一个你喜欢的测试框架，深入地了解一下它们新特性，挑一个让你印象深刻的特性和我们分享。期待在留言区看到你的想法。

上一篇 03 | 程序员的测试与测试人员的测试有什么不同？

精选留言 (1)

bigben
2021-08-11

测试框架多了也让人头疼，不知道该选哪个？

展开