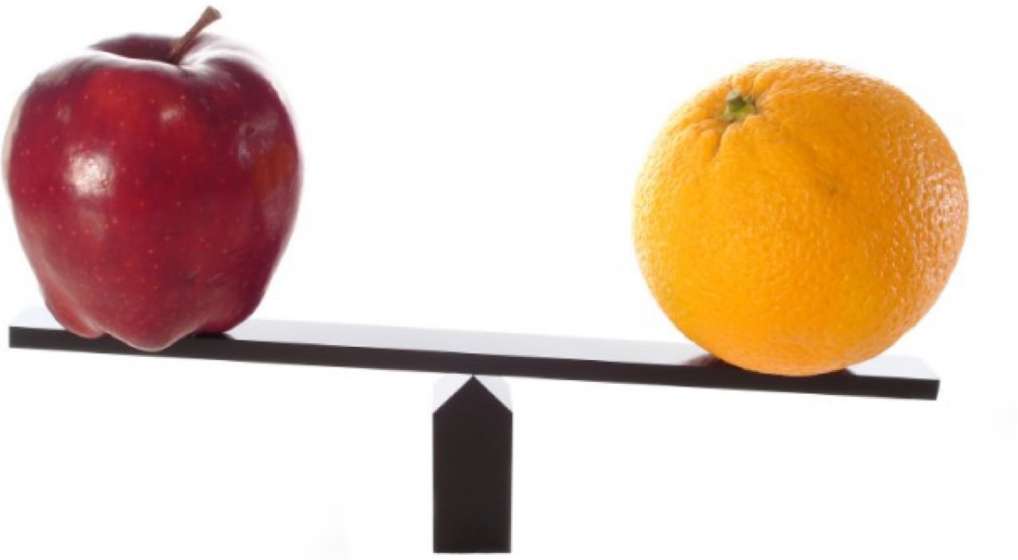
写给精明Java开发者的测试技巧

by Wing, importnew.com



我们都会为我们的代码编写测试，不是吗？毫无疑问，我知道这个问题的答案可能会从“当然，但你知道怎样才能避免写测试吗？”到“**必须的！我爱测试**”都有。接下来我会给你几个小建议，它们可以让你编写测试变得更容易。那会帮助你减少脆弱的测试，并保证应用程序更加健壮。

与此同时，如果你的答案是“*不，我不编写测试。*”，那么我希望这些简单但有效的技术可以让你了解编写测试带来的好处。你也会看到，编写一个复杂、没有价值的测试集（test suit）并没有你认为的那么难。

如何编写测试、有哪些用于管理测试集合的最佳实践这些主题并不新鲜。我们在过去已经就这个问题的某些方面讨论了很多次。从“在构建过程中使用集成测试的正确方式”到谈论“在单元测试中恰当地模拟环境”，再到“代码覆盖率以及如何找到哪些是你真正需要测试的代码”。

但是，今天我想和你谈论一系列小建议，这些建议可以帮助你在头脑中理清测试自下而上是

如何运作的。从如何构造一个简单的单元测试到对 mock（模拟） 和 spy（监视） 以及复制粘贴测试代码更高层次的理解。那我们开始吧。

AAArrr，像海盗一样说话?

和大部分软件开发一样，模式通常都是一个不错的开始。无论是想要通过工厂来创建对象，或者希望将web应用程序中的关注点分散到Model、View和Controller中，在它们背后通常都会有一个模式，帮助你理解正在发生什么并解决困难。 那么，一个典型的测试看上去应该是怎么样的?

当我们编写测试时，其中一个最有用但却极其简单的模式是计划-执行-断言（Arrange-Act-Assert），简称AAA。

这个模式的前提是所有测试都应该遵循默认布局。测试系统所必需的全部条件和输入都应该在测试方法开始的时候被设置（Arrange）。在计划好所有前置条件后，我们通过触发一个方法或者检查系统的某些状态的方式，在测试系统上运行（Act）。最后，我们需要断言（Assert）测试系统是否已经生成了期望的结果。

让我们来看一个Java JUnit测试的示例，它展示了这种模式:

```
@Test
public void testAddition() {
    // Arrange
    Calculator calculator = new Calculator();

    // Act
    int result = calculator.add(1, 2);

    // Assert
    assertEquals(&amp;quot;Calculator.add returns invalid result&amp;quot;, 3, result);
}
```

看看代码流多么精准！计划-执行-断言模式可以让你快速理解测试的功能。偏离了这个模式后会很容易写出非常糟糕的代码。

牢记迪米特法则

迪米特法则在软件上面应用了最小知识原则，减小了单元的耦合——这一直是在开发软件的设计目标。

迪米特法则可以表述为一系列的规则：

- 在方法中，一个类的实例可以调用该类的其它方法；
- 在方法中，实例可以查询自己的数据，但不能查询数据的数据（译者注：即实例的数据比较复杂时，不能进行嵌套查询）；
- 当方法接收参数时，可以调用参数的第一级方法；
- 当方法创建了一些局部变量的实例后，这个类的实例可以调用这些局部变量的方法；
- 不要调用全局对象的方法。

那么，就测试而言，这些意味着什么呢？好吧，由于迪米特法则减少了应用程序各部分之间的耦合，这意味着测试应用程序中的各个部分变得更加容易。为了要查看该法则如何为测试提供帮助，我们来看一个定义非常糟糕的类，它违背了迪米特法则：

考虑下面这个我们要测试的类：

```
public class Foo() {
    public Bar doSomething(Baz aParameter) {
        Bar bar = null;
        if (aParameter.getValue().isValid()) {
            aParameter.getThing().increment();
            bar = BarManager.getBar(new Thing());
        }
        return bar;
    }
}
```

如果我们试着去测试这个方法，很快就会发现一些问题。这些问题是由于定义方法的方式导致的。

我们在测试这个方法时会遇到的第一个困难是，我们调用了一个静态方法——*BarManager.getBar()*。我们没有办法在单元测试中简单指定如何操作这个方法。还记得我们提过的**计划-执行-断言**模式吗？但在这里，在通过调用 doSomething() 执行这个方法之前，我们没有一种简单的方式来设置 BarManager。如果 BarManager.getBar() 不是一个静态方法，那么可以向 doSomething() 方法中传入一个 BarManager 实例。在测试集中，传递一个样本值（sample value）是非常容易的，并且我们也可以更好地控制和预测方法的执行过程。

我们还可以看到，在这个示例方法中调用了方法链：aParameter.getValue().isValid() 和 aParameter.getThing().increment()。为了测试它们，我们需要明确地知道aParameter.getValue() 和 aParameter.getThing() 的返回结果类型，然后才可以在测试中构建恰当的模拟值。

如果要做这些，那么我们不得不去了解这些方法返回对象的详细信息。而我们的单元测试就会开始变形，逐渐成为一大堆不能维护的、脆弱的代码。我们正在破坏单元测试中一个基本规则：只测试单独的单元，而不是这个单元的实现细节。

我并不是在说单元测试**只能**测试单独的类。然而在大多数情况下，把类作为一个单独的单元考虑，可能是一个好主意。但是有些情况下，我们会将两个或者更多的类看做是一个单元。

在这里我为各位读者留下一个练习：对这个方法进行完全重构，使其更容易被测试。但对于新手来说，我们可能会将 aParameter.getValue() 对象作为一个参数传递给这个方法。这样会满足一些规则，提升方法的可测试性。

了解何时使用断言

对于编写应用程序测试来说，JUnit和TestNG都是非常优秀的框架，它们提供了许多不同的方法在测试中对一个值进行断言。例如，检查两个值是相同还是不同，或者值是否为空。

好，既然已经同意断言很酷，那么让我们随时随地使用它们吧！等一下，过度使用断言会使得测试变得脆弱，从而导致无法维护。一旦这样，我们很清楚后面的结果是怎样的——不能被测试和不稳定的代码。

考虑下面的测试示例：

```
@Test

public void testFoo {
    // Arrange
    Foo foo = new Foo();
    double result = ...;

    // Act
    double value = foo.bar( 100.0 );

    // Assert
    assertEquals(value, result);
    assertNotNull( foo.getBar() );
    assertTrue( foo.isValid() );
}
```

乍一看，这段代码没有什么问题。我们遵循了AAA模式，并断言了一些发生了的事情——那么哪里错了？

首先，我们看到这个测试的名字：testFoo，它并没有真正告诉我们这个测试在做什么事情，并且没有匹配任何一个我们在检查的断言。

然后，如果其中一个断言失败了，我们能够确定测试系统中的哪部分失败了吗？是 foo.bar(100.0) 方法失败了？还是 foo.getBar() 或者 foo.isValid() 方法失败了？如果不通过测试内部调试来试着找出到底发生了什么，我们是无从知道的。

单云测试的目的在于，我们想要一个可信赖的、健壮的测试集。通过快速运行它们，我们可以知道应用程序的状态。而示例中的产生的这种麻烦，已经使得我们的目的落空。如果测试失败，我们不得不运行调试器来找到到底什么地方失败了，那么我们的处境也会变得困难。

通常来说，一种最佳实践是在一个特定的测试中，只有一个最合适的断言。这样我们可以确保测试是明确地，目标是应用程序的单个功能点。

Spy、Mock和Stub，天哪！

有时，Spy应用程序在做什么，或者验证程序使用特定参数调用了特定方法并调用了指定次数，是很有用的。有时，我们想触发数据库层，但又想模拟数据库返回给我们的响应。在Spy、Mock和Stub的帮助下，我们可以实现所有这些功能。

在Java中，我们有很多不同的库，可以用来Spy、Mock和Stub，例如Mockito、EasyMock和JMockit。那么Spy、Mock和Stub之间有什么区别？我们应该在何时使用它们呢？

Spy可以让你很容易检查程序是否使用正确的参数调用了某些方法，并且会记录这些参数以供后面的验证使用。例如，如果你在代码中有一个循环，在每次循环中会触发一个方法，那么Spy可以用来验证该方法被触发的次数是正确的，并且每次触发时都使用了正确的传入参数。对于某些特定类型的存根来说，Spy是至关重要的。

Stub（存根）是一个对象，它可以在客户端触发某种请求时，提供特定的已经存储的响应，例如，针对输入存根已经有通过预编程生成的响应。当你想在代码片段中强行设定某些条件时，存根会很有用，例如，如果数据库调用失败，而你希望在测试中触发数据库异常处理。存根是模拟对象个一个特例。

Mock（模拟）对象提供了存根对象的所有功能，而且它还提供了预编程的期望结果。这就是说模拟对象和真实对象非常接近，它可以根据之前设定的状态来执行不同的行为。例如，我们可以用模拟对象来表示一个安全系统，它根据登录的不同用户，提供不同的访问控制。就我们的测试而言，它会和一个真实的安全系统交互，而我们可以在应用程序中测试很多不同的路径。

有时，我们会使用**Test Double**（测试替身）一词来表示如上所述的任何类型的对象，我们在

测试中会和这些对象进行交互。

通常来说，spy提供了最少的功能，因为它的目的就在于捕捉方法是否被调用。如果被调用，传入的是什么参数。

Stub是下一个级别的测试替身，它通过设置预定义的方法调用返回值的方式，来设定测试系统的执行流程。一个特定的存根对象通常可以在很多测试中使用。

最后，mock object（模拟对象）提供了远比比存根对象更多的行为。就这一点而言，一种最佳实践是针对特定测试开发特定存根对象，否则存根对象就会想真实对象那样开始变得复杂。

不要让你的测试过度DRY

在软件开发过程中，通常让你的应用程序DRY（不要重复自己，Don't Repeat Yourself）是一种最佳实践。

在测试中，情况并不总是这样。当编写软件时，一种最佳实践是重构那些通用的代码片段，将其放入单独的方法中，那么这些方法就可以在代码中被调用很多次。这样做很有意义，因为我们只编写一次代码，然后也只需要测试一次。另外，如果我们只需要将代码片段编写一次，我们也可以避免由于编写很多次带来的拼写错误。**要当心复制粘贴！**

2006年，Jay Fields创造了一个新词：DAMP（Descriptive And Meaningful Phrases，描述性和有意义的短语），它用来指代那些设计良好的领域特定语言。如果你想再次回忆，可以参考最初的邮件：DRY code, DAMP DSL。

DAMP背后的原理是这样的，对于一个好的领域特定语言来说，它会使用描述性和有意义的短语来增加语言的可读性，并降低高效使用该语言所需要的学习和培训时间。

通常，在一个测试集中的许多单元测试可能都非常类似，唯一的微小区别就在于如何针对测试准备测试系统。因此，对于软件开发人员来说，将这些重复的代码从单元测试重构到帮助函数中是很自然的。同样将实例变量重构成静态变量也是很自然的，这样它们就可以只针对每一个测试类声明一次——再一次从测试中移除重复代码。

尽管在做出如上重构后，代码会变得更加"整洁"，但这些单元测试作为一个单独的部分会变得更难读懂。如果一个单元测试调用了其它几个方法，并且在使用非局部变量，那么单元测试的流程就变得不直观，并且你也不能够像之前那样容易理解单元测试的基本流程。

至关重要的是，如果我们让我们的单元测试DRY，那么测试的复杂度反而会变得更高，而测试的维护工作也会变得更加困难——这正好和让测试DRY的初衷相违背。对于单元测试来

说，让它们更DAMP、而不是DRY，这会增加测试的可读性和可维护性。

关于应该在多大程度上重构你的测试，我们并没有正确或者错误的答案，但**我们要努力在让测试过于DRY和过于DAMP之间做一个平衡，这通常肯定会让我们的测试变得更加容易维护**。

结论

在这篇文章中，我介绍了五个基本原则，这些原则会帮助我们针对应用程序编写单元测试。如果你有任何想法，欢迎通过下面的评论进行分享，或者你可以在Twitter上找到我：@cocoadavid。

希望你能够希望我们讨论过的这些原则，并且能够看到它们是如何潜移默化地让你热爱编写单元测试。是的，我是说“热爱”，因为我相信编写单元测试是高品质软件的基本要求。

高品质软件意味着满意的用户，而满意的用户意味着幸福的开发人员。

开发快乐！

原文链接： zeroturnaround 翻译： ImportNew.com - Wing
译文链接： http://www.importnew.com/16392.html