测试指南

Rei 写于 26 Oct 2015

测试有很多种，像用户测试，A/B 测试等等，这里说的是由开发人员自己实行，用于确保开发过程没有引入错误的代码测试。

测试不是一个新概念，相反部分社区可能过度狂热，制造了太多的测试框架和库，增加了很多复杂性，以至于让人敬而远之。其实测试只是一个简单的概念，这篇文章尝试说明这一点。

测试

先看一个例子，假如我们需要实现一个方法 fizz_buzz(n)，要求 n 是一个整数，如果 n 是 3 的倍数，就返回 'Fizz'；如果 n 是 5 的倍数，就返回 'Buzz'；其余则返回 n 本身。这个方法没什么实际作用，但用来做例子很合适，我们假设这个方法是某个生产应用的关键算法。

这个方法很简单，一会就能写出来：

```
# fizz_buzz.rb
def fizz_buzz(n)
  if n % 3 == 0
    'Fizz'
  elsif n % 5 == 0
    'Buzz'
  else
    n
  end
end
```

要验证这个方法是否正确，可以在终端执行这个方法查看结果：

```
> require './fizz_buzz.rb'
> fizz_buzz 1
=> 1
> fizz_buzz 2
=> 2
> fizz_buzz 3
=> "Fizz"
> fizz_buzz 4
=> 4
> fizz_buzz 5
=> "Buzz"
```

看起来没问题，于是就把这个方法用到产品环境中了……然后有一天，需求更改了，要求增加一个逻辑：如果 n 同时是 3 和 5 的倍数，就返回 FizzBuzz。而当前的实现只会返回 Fizz：

```
> fizz_buzz 15
=> "Fizz"
```

于是修改这个方法：

```
# fizz_buzz.rb
def fizz_buzz(n)
  if n % 3 == 0 and n % 5 == 0
    'FizzBuzz'
  elsif n % 3 == 0
    'Fizz'
  elsif n % 5 == 0
    'Buzz'
  else
    n
  end
end
```

然后到终端调试：

```
> require './sum.rb'
> fizz_buzz 15
=> "FizzBuzz"
```

但是这个修改有没有破坏以前的行为呢？这时候再用以前的数据调试一下：

```
> fizz_buzz 1
=> 1
> fizz_buzz 2
=> 2
> fizz_buzz 3
=> "Fizz"
...
```

这里遇到一个问题，我们在重复以前的调试内容。重复一两次还没问题，三次以上就很烦人了。并且随着代码量上升，越来越难确定修改会影响什么地方的逻辑，容易引入 bug。

高效程序员会将调试代码固化下来，写成测试代码。

新建一个文件，写入测试代码：

```
# fizz_buzz_test.rb
require './fizz_buzz.rb'
```

```ruby
fizz_buzz(1) == 1 ? print('.') : raise("fizz_buzz 1 should be 1")
fizz_buzz(3) == 'Fizz' ? print('.') : raise("fizz_buzz 3 should be Fizz")
fizz_buzz(5) == 'Buzz' ? print('.') : raise("fizz_buzz 5 should be Buzz")
puts 'done'
```

这个脚本会对比程序输出和预期结果，如果结果一致就会打印一个点.，否则会抛出异常，中止测试并打印错误信息。

```
$ ruby fizz_buzz_test.rb
...done
```

我们可以故意把方法写错，看看有什么结果：

```ruby
# fizz_buzz.rb
def fizz_buzz(n)
  n
end
```

再次运行，结果就是：

```
$ ruby fizz_buzz_test.rb
.fizz_buzz_test.rb:4:in `<main>': fizz_buzz 3 should be Fizz (RuntimeError)
```

有了测试脚本的帮助，我们就能知道对代码的修改有没有破坏以前的逻辑。修改了代码之后，别忘了加上新增部分功能的测试：

```ruby
fizz_buzz(15) == 'FizzBuzz' ? print('.') : raise("fizz_buzz 15 should be FizzBuzz")
```

再次运行测试：

```
$ ruby fizz_buzz_test.rb
....done
```

assert（断言）

之前的测试代码里面有不少重复代码，例如 print，raise 等等。我们可以把这些跟测试用例没有直接关系的代码抽取出通用方法，这类方法有一个惯用名称 assert，于是测试代码简化成：

```ruby
def assert(test, msg = nil)
  test ? print '.' : raise(msg)
end
```

```ruby
assert fizz_buzz(1) == 1, "fizz_buzz 1 should be 1"
assert fizz_buzz(3) == 'Fizz', "fizz_buzz 3 should be Fizz"
assert fizz_buzz(5) == 'Buzz', "fizz_buzz 5 should be Buzz"
assert fizz_buzz(15) == 'FizzBuzz', "fizz_buzz 15 should be FizzBuzz"
puts 'done'
```

测试代码多了之后，会发现有一类测试有固定的模式，例如上面的测试就是判断一个方法的输出跟另一个值是否相等，这样又可以抽取出一个 assert_equal 方法：

```ruby
def assert(test, msg = nil)
  test ? print '.' : raise(msg)
end

def assert_equal(except, actual, msg = nil)
  assert(except == actual, msg)
end

assert_equal 1, fizz_buzz(1), "fizz_buzz 1 should be 1"
assert_equal 'Fizz', fizz_buzz(3), "fizz_buzz 3 should be Fizz"
assert_equal 'Buzz', fizz_buzz(5), "fizz_buzz 5 should be Buzz"
assert_equal 'FizzBuzz', fizz_buzz(15), "fizz_buzz 15 should be FizzBuzz"
puts 'done'
```

这里看起来好像没减少代码，不过起码不用怕写错 == 号了。常见的 assert_* 方法还有：

- assert_nil(object, msg) 测试对象是否 nil。
- assert_empty(object, msg) 测试对象调用 .empty? 是否返回 true。
- assert_includes(collection, object, msg) 测试集合 collection 是否包含 object。
- assert_throws(exception) { } 测试执行一个 block，是否会抛出某个异常。

这些方法都不过是 assert 的包装，只要知道 assert 的原理，这些辅助方法都能自己实现，或者实现其他适合场景的断言方法。

现在每个主流语言都会有一个测试库，在 Ruby 中就是 Minitest。测试库除了包含一些断言方法外，还提供测试代码隔离、测试环境重置、更好的错误提示等功能，你可以阅读文档了解详情。

TDD

TDD 是 Test-driven development（测试驱动）的缩写，它是一种开发方法，提倡在实现功能之前先写测试，从而实现更好的程序质量和接口设计。它的流程可以简化如下：

1. 写测试
2. 运行测试（失败）
3. 写功能
4. 运行测试（通过）
5. 回到 1

这种方法在一定程度上是有效的，先写测试可以提供高覆盖度的代码测试，减少 bug。并且写测试的过程就是在实际调用程序接口，有助于理清设计思路。要详细了解 TDD，可以读《测试驱动开发》这本书。

不过如果片面追求测试覆盖率，就很容易写出比功能代码多几倍的测试代码。测试也是代码，代码就有维护成本，过多的测试反而降低编码效率。对此《测试驱动开发》的作者 Kent Beck 有过解释：

> I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence. （我的薪水是付给能用的代码，而不是给测试的，所以我的哲学是在可信赖的程度上，尽量少写测试。）
> http://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests

另外，TDD 也可能对设计产生误导，变成追求测试的简洁而不是接口简洁。DHH 在 RailsConf 2014 上举了一个例子：

```
class Person
  def age
    Date.today.year - birthday.year
  end
end

test "a person's age is determined by birthday" do
  sevent_niner = Person.new birthday: Date.new(1979)
  travel_to Date.new(2009)
  assert_equal 30, sevent_niner.age
end
```

在这个例子中，Person#age 的返回结果是会随着时间变化的，为了保证测试的可靠，需要使用 travel_to 这个 hack，把测试用例的时间固定在某个值。

这时候为了写出优美的测试代码，可能会想到另一个接口设计：

```
class Person
  def age(now = Date.today)
    now.year - birthday.year
  end
end

test "a person's age is determined by birthday" do
  sevent_niner = Person.new birthday: date.new(1979)
  assert_equal 30, sevent_niner.age Date.new(2009)
end
```

通过给 age 添加一个参数，让测试代码避免了 hack，似乎是一个更好的方案。但实际上，在正常的调用中，age 并不需要参数，这个参数完全是为了测试而添加的，这给原先的设计增加了不必要的复杂性。这就是测试驱动开发误导了设计的例子。

总结

以上内容就是关于测试必须了解的内容，至于 DSL、Mock/Stub、Factory……之类的工具都是锦上添花，不是必须的。

为了项目的可维护性，也为了节约自己的时间，应该积极的拥抱测试。但也不要忘了测试只是辅助开发的工具，不要本末倒置，使用太复杂的测试工具增加维护难度。