用 Testify 来改善 GO 测试和模拟

(/gctt/csshawn)

csshawn (/gctt/csshawn) · 2018-12-09 18:20:41 · 658 次点击 · 预计阅读时间 8 分钟 · 大约1分钟之前 开始浏览

目录

- **入门指南**
- **一个简单的示例**
  - **否定测试案例和 Nil 测试**
  - **将 Testify 与表驱动测试相结合**
- **模拟**
  - **模拟示例**
  - **用 Mockery 生成模仿对象**
- **关键点**
- **总结**
- **延伸阅读**

断言是我感觉 Go 中的标准库真正缺失的东西。你绝对可以通过 `if` 比较或者其他任何方式获得相同的结果，但那不是写测试文件最简洁的方法。

这就需要诸如 **stretchr/testify (https://github.com/stretchr/testify)** 之类的东西来挽救局面了。如果不是世界各地 Go 开发人员最受欢迎的测试包，这个软件包很快就会成为最流行的测试软件包之一。

它优雅的语法使您能够编写简单得难以置信的断言。

入门指南

想要启动和运行 testify 包，我们要做的第一件事就是安装它。如果你当前正在使用 Go 模块，那么在你导入第一个 `*_test.go` 文件后，只需要执行 `go test ...` 即可。

但是，如果你仍然在使用较旧版本的 Go，你可以通过输入下面指令来获取包：

```
go get GitHub.com/stretchr/testify
```

完成此操作后，我们应该将它很好地并入到我们的各种测试套件中去。

一个简单的示例

让我们先看看传统上如何在 Go 中编写测试。这将让我们对提高可读性的 `testify` 有所了解。

我先来写一个简单的 Go 程序，它有一个输出函数，`Calculate()`。

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  // 计算并返回 x + 2.
8  func Calculate(x int) (result int) {
9      result = x + 2
10     return result
11 }
12
13 func main() {
14     fmt.Println("Hello World")
15 }
16
```

如果我们用传统方法来编写测试，我们通常会得到这样的结果：

```
1  package main
2
3  import (
4      "testing"
5  )
6
7  func TestCalculate(t *testing.T) {
8      if Calculate(2) != 4 {
9          t.Error("Expected 2 + 2 to equal 4")
10     }
11 }
12
```

然后我们通过调用 `go test ./... -v` 来尝试运行这个简单的测试，传入 `-v` 以确保我们可以看到更详细的输出。

如果我们想要更完美，我们可以在这里加入表驱动测试，以确保测试了各种各样的情况。现在，让我们试着修改这个基本的方法，看看 `testify` 是如何工作的：

```
package main

import (
    "testing"
)

func TestCalculate(t *testing.T) {
    assert.Equal(t, Calculate(2), 4)
}
```

很好，如你所见，我们使用 `assert.Equal` 函数简单地测试了相等性。这看起来是一种改进，因为我们需要阅读的代码行数更少了，而且我们可以清楚地看到测试函数想要达到的效果。

否定测试案例和 Nil 测试

我们已经看过了情况较好的测试，但否定断言和 Nil 检查怎么样呢。好吧，幸运的是， `testify` 软件包有方法允许我们测试两者。

假设我们想要测试一个用来返回指定应用程序状态的函数。例如，如果应用程序处于活动状态并等待请求，那么状态将返回 `"waiting"`，如果它已崩溃，那么它将返回 `"down"` 以及其服务请求时或其等待第三方时的各种其他状态等。

当我们执行测试时，我们希望我们的测试在状态不等于 `"down"` 的情况下都能通过，因此我们可以在这个特定的假设情况下使用 `assert.NotEqual()` 。

```
func TestStatusNotDown(t *testing.T) {
    assert.NotEqual(t, status, "down")
}
```

如果我们想测试 `"status"` 是否为 nil，那么我们可以使用 `assert.Nil(status)` 或者 `assert.NotNil(object)`，这取决于我们希望做何反应当它为 `nil` 时。

将 Testify 与表驱动测试相结合

将 `testify` 并入到我们测试套件中并不会妨碍我们使用诸如表驱动测试之类的方法，事实上，它能使测试变得更简单。

```
package main

import (
    "testing"

    "github.com/stretchr/testify/assert"
)

func TestCalculate(t *testing.T) {
    assert := assert.New(t)

    var tests = []struct {
        input    int
        expected int
    }{
        {2, 4},
        {-1, 1},
        {0, 2},
        {-5, -3},
        {99999, 100001},
    }

    for _, test := range tests {
        assert.Equal(Calculate(test.input), test.expected)
    }
}
```

请注意我们在此示例中调用的 `assert.Equal()` 与上面例子中的细微差别。我们用 `assert using assert.New(t)` 初始化了断言，然后我们可以多次调用 `assert.Equal()`，只需传入输入值和期望值，而不是每次都将 t 作为第一个参数传入。当然这不是什么大问题，但它确实使我们的测试看起来更简洁明了。

模拟

`testify` 包另外一个优秀的功能就是它的模拟功能。有效的模拟允许我们在代码里创建一个替代的对象，用来模拟对象的某些行为，这样我们在运行测试用例时就不用每次都期望它能够触发。

例如，一个是消息服务或电子邮件服务，无论何时被调用，都会向客户端发送电子邮件。如果我们正在积极地开发我们的代码库，可能每天会运行数百次测试，但我们不希望每天向客户发送数百封电子邮件或消息，因为那样他们可能会不高兴。

那么，我们要如何使用 `testify` 包来模拟呢？

模拟示例

让我们来看一下如何将 `mocks` 应用到一个相当简单的例子中。在这个例子中，我们有一个系统会尝试向客户收取产品或服务的费用。当 `ChargeCustomer()` 被调用时，它将随后调用 Message Service，向客户发送 SMS 文本消息来通知他们已经被收取的金额。

```go
package main

import (
    "fmt"
)

// MessageService 通知客户被收取的费用
type MessageService interface {
    SendChargeNotification(int) error
}

// SMSService 是 MessageService 的实现
type SMSService struct{}

// MyService 使用 MessageService 来通知客户
type MyService struct {
    messageService MessageService
}

// SendChargeNotification 通过 SMS 来告知客户他们被收取费用
// 这就是我们将要模拟的方法
func (sms SMSService) SendChargeNotification(value int) error {
    fmt.Println("Sending Production Charge Notification")
    return nil
}

// ChargeCustomer 向客户收取费用
// 在真实系统中，我们会模拟这个
// 但是在这里，我想在每次运行测试时都赚点钱
func (a MyService) ChargeCustomer(value int) error {
    a.messageService.SendChargeNotification(value)
    fmt.Printf("Charging Customer For the value of %d\n", value)
    return nil
}

// 一个 "Production" 例子
func main() {
    fmt.Println("Hello World")

    smsService := SMSService{}
    myService := MyService{smsService}
    myService.ChargeCustomer(100)
}
```

那么，我们如何进行测试以确保我们不会让客户疯掉？好吧，我们通过创建一个新的 `struct` 称之为 `smsServiceMock` ，用来模拟我们的 SMSService，并且将 mock.Mock 添加到它的字段列表中。

然后我们将改写 `SendChargeNotification` 方法，这样它就不会向我们的客户发送通知并返回 `nil` 错误。

最后，我们创建 `TestChargeCustomer` 测试函数，接着实例化一个新的类型实例 `smsServiceMock` 并指定 `SendChargeNotification` 在被调用时应该做什么。

```go
package main

import (
    "fmt"
    "testing"

    "github.com/stretchr/testify/mock"
)

// smsServiceMock
type smsServiceMock struct {
    mock.Mock
}

// 我们模拟的 smsService 方法
func (m *smsServiceMock) SendChargeNotification(value int) bool {
    fmt.Println("Mocked charge notification function")
    fmt.Printf("Value passed in: %d\n", value)
    // 这将记录方法被调用以及被调用时传进来的参数值
    args := m.Called(value)
    // 它将返回任何我需要返回的
    // 这种情况下模拟一个 SMS Service Notification 被发送出去
    return args.Bool(0)
}

// 我们将实现 MessageService 接口
// 这就意味着我们不得不改写在接口中定义的所有方法
func (m *smsServiceMock) DummyFunc() {
    fmt.Println("Dummy")
}

// TestChargeCustomer 是个奇迹发生的地方
// 在这里我们将创建 SMSService mock
func TestChargeCustomer(t *testing.T) {
    smsService := new(smsServiceMock)

    // 然后我们将定义当 100 传递给 SendChargeNotification 时，需要返回什么
    // 在这里，我们希望它在成功发送通知后返回 true
    smsService.On("SendChargeNotification", 100).Return(true)

    // 接下来，我们要定义要测试的服务
    myService := MyService{smsService}
    // 然后调用方法
    myService.ChargeCustomer(100)

    // 最后，我们验证 myService.ChargeCustomer 调用了我们模拟的 SendChargeNotification 方法
    smsService.AssertExpectations(t)
}
```

所以，当我们运行 `go test ./... -v` 时，我们应该看到以下输出：

```
go test ./... -v
=== RUN   TestChargeCustomer
Mocked charge notification function
Value passed in: 100
Charging Customer For the value of 100
--- PASS: TestChargeCustomer (0.00s)
    main_test.go:33: PASS:      SendChargeNotification(int)
PASS
ok      _/Users/elliot/Documents/Projects/tutorials/golang/go-testify-tutorial  0.012s
```

正如你所看到的，我们的模拟方法被调用了而不是 “ production ” 方法，这证明我们的 `myService.ChargeCustomer()` 方法按照我们所期望的方式在运行！

高兴的时刻，我们现在已经能够使用模拟的方法来全面测试更复杂的项目。值得注意的是，此技术可用于各种不同的系统，例如模拟数据库查询或者是如何与其他 API 交互。总的来说，模拟是非常强大的手段，如果你要在 Go 中测试产品级系统，那么你应该掌握它。

用 Mockery 生成模仿对象

在上面的例子中，我们自己模拟了所有的方法，但在实际的例子中，这可能意味着有海量的方法和函数需要来模拟。

值得庆幸的是，这里有 **vektra/mockery (https://github.com/vektra/mockery)** 包来当我们的好帮手。

mockry 的二进制文件可以找到任何你在 Go 中定义的 `interfaces` 的名字，然后会自动输出生成模仿对象到 `mocks/InterfaceName.go` 。当你想节省大量时间时，这非常的方便，我强烈建议你使用这个工具！

关键点

- Testify 可以帮助你简化在测试用例中编写断言的方式。
- Testify 还可用于模拟测试框架中的对象，以确保你在测试时不会调用产品端。

总结

希望这能助你揭开 `stretchr/testify` 软件包测试 Go 项目的 神秘面纱。在本教程中，我们已经看到如何使用 `testify` 包中的断言来执行判断事物相等，不相等或者为 nil。

我们还看到如何模拟系统的各个部分，以确保在运行测试时不会与生产系统交互并执行你不想做的事情。

如果你认为这有用，或者你有任何意见或反馈，请随时在下面的评论区告诉我。

延伸阅读

如果你喜欢这篇文章，你可能喜欢我的另外一篇文章关于 Go 测试的：

- **Advanced Testing in Go (https://tutorialedge.net/golang/advanced-go-testing-tutorial/)**

---

via: https://tutorialedge.net/golang/improving-your-tests-with-testify-go/ (https://tutorialedge.net/golang/improving-your-tests-with-testify-go/)

作者：Elliot Forbes (https://tutorialedge.net/about/) 译者：csshawn (https://github.com/csshawn) 校对：polaris1119 (https://github.com/polaris1119)

本文由 GCTT (https://github.com/studygolang/GCTT) 原创编译，Go语言中文网 (/articles/16799) 荣誉推出