GoConvey框架使用指南

张晓龙 关注 IP属地: 陕西

3 2017.05.08 00:21:22 字数 2,636 阅读 49,285

序言

在软件开发中，产品代码的正确性通过测试代码来保证，而测试代码的正确性谁来保证？答案是毫无争议的，肯定是程序员自己。这就要求测试代码必须足够简单且表达力强，让错误无处藏身。我们要有一个好鼻子，能够嗅出测试的坏味道，及时的进行测试重构，从而让测试代码易于维护。笔者从大量的编码实践中感悟道：虽然能写出好的产品代码的程序员很牛，但能写出好的测试代码的程序员更牛，尤其对于TDD实践。

要写出好的测试代码，必须精通相关的框架。对于Golang程序员来说，至少需要掌握下面两个框架：

- GoConvey
- GoMonkey

本文将主要介绍GoConvey框架的基本使用方法，从而指导读者更好的进行测试实践，最终写出简单优雅的测试代码。

GoConvey简介

GoConvey是一款针对Golang的测试框架，可以管理和运行测试用例，同时提供了丰富的断言函数，并支持很多 Web 界面特性。Golang虽然自带了单元测试功能，并且在GoConvey框架诞生之前也出现了许多第三方测试框架，但没有一个测试框架像GoConvey一

样能够让程序员如此简洁优雅的编写测试代码。

安装

在命令行运行下面的命令：

```
go get github.com/smartystreets/goconvey
```

运行时间较长，运行完后你会发现：

1. 在$GOPATH/src目录下新增了github.com子目录，该子目录里包含了GoConvey框架的库代码
2. 在$GOPATH/bin目录下新增了GoConvey框架的可执行程序goconvey

> 注：上面是在gopath时代使用GoConvey的API前的安装方法，而在gomod时代一般不需要先显式安装（gomod机制会自动从goproxy拉取依赖到本地cache），除非要使用GoConvey的web界面，这时需要提前安装GoConvey的二进制，命令为go install github.com/smartystreets/goconvey@latest。

基本使用方法

我们通过一个案例来介绍GoConvey框架的基本使用方法，并对要点进行归纳。

产品代码

我们实现一个判断两个字符串切片是否相等的函数StringSliceEqual，主要逻辑包括：

- 两个字符串切片长度不相等时，返回false
- 两个字符串切片一个是nil，另一个不是nil时，返回false
- 遍历两个切片，比较对应索引的两个切片元素值，如果不相等，返回false
- 否则，返回true

根据上面的逻辑，代码实现如下所示：

```
func StringSliceEqual(a, b []string) bool {
    if len(a) != len(b) {
        return false
    }

    if (a == nil) != (b == nil) {
        return false
    }

    for i, v := range a {
        if v != b[i] {
            return false
        }
    }
    return true
}
```

对于逻辑“两个字符串切片一个是nil，另一个不是nil时，返回false”的实现代码有点不好理解：

```
if (a == nil) != (b == nil) {
    return false
}
```

我们实例化一下a和b，即[]string{}和[]string(nil)，这时两个字符串切片的长度都是0，但肯定不相等。

测试代码

先写一个正常情况的测试用例，如下所示：

```
import (
    "testing"
    . "github.com/smartystreets/goconvey/convey"
)

func TestStringSliceEqual(t *testing.T) {
    Convey("TestStringSliceEqual should return true when a != nil  && b != nil", t, func() {
        a := []string{"hello", "goconvey"}
        b := []string{"hello", "goconvey"}
        So(StringSliceEqual(a, b), ShouldBeTrue)
    })
}
```

由于GoConvey框架兼容Golang原生的单元测试，所以可以使用go test -v来运行测试。
打开命令行，进入$GOPATH/src/infra/alg目录下，运行go test -v，则测试用例的执行结果日下：

```
=== RUN   TestStringSliceEqual

  TestStringSliceEqual should return true when a != nil  && b != nil ✓


1 total assertion

--- PASS: TestStringSliceEqual (0.00s)
PASS
ok      infra/alg       0.006s

```

上面的测试用例代码有如下几个要点：

1. import goconvey包时，前面加点号"."，以减少冗余的代码。凡是在测试代码中看到Convey和So两个方法，肯定是convey包的，不要在产品代码中定义相同的函数名
2. 测试函数的名字必须以Test开头，而且参数类型必须为*testing.T
3. 每个测试用例必须使用Convey函数包裹起来，它的第一个参数为string类型的测试描述，第二个参数为测试函数的入参（类型为*testing.T），第三个参数为不接收任何参数也不返回任何值的函数（习惯使用闭包）
4. Convey函数的第三个参数闭包的实现中通过So函数完成断言判断，它的第一个参数为实际值，第二个参数为断言函数变量，第三个参数或者没有（当第二个参数为类ShouldBeTrue形式的函数变量）或者有（当第二个函数为类ShouldEqual形式的函数变量）

我们故意将该测试用例改为不过：

```
import (
    "testing"
    . "github.com/smartystreets/goconvey/convey"
)

func TestStringSliceEqual(t *testing.T) {
    Convey("TestStringSliceEqual should return true when a != nil  && b != nil", t, func() {
        a := []string{"hello", "goconvey"}
        b := []string{"hello", "goconvey"}
        So(StringSliceEqual(a, b), ShouldBeFalse)
    })
}
```

测试用例的执行结果日下：

```
=== RUN   TestStringSliceEqual

  TestStringSliceEqual should return true when a != nil  && b != nil ✗


Failures:

  * /Users/zhangxiaolong/Desktop/D/go-workspace/src/infra/alg/slice_test.go
  Line 45:
  Expected: false
  Actual:   true


1 total assertion

--- FAIL: TestStringSliceEqual (0.00s)
FAIL
exit status 1
FAIL    infra/alg       0.006s


```

我们再补充3个测试用例：

```
import (
    "testing"
    . "github.com/smartystreets/goconvey/convey"
)

func TestStringSliceEqual(t *testing.T) {
    Convey("TestStringSliceEqual should return true when a != nil  && b != nil", t, func() {
        a := []string{"hello", "goconvey"}
        b := []string{"hello", "goconvey"}
        So(StringSliceEqual(a, b), ShouldBeTrue)
    })

```

```
    Convey("TestStringSliceEqual should return true when a == nil  && b == nil", t, func() {
        So(StringSliceEqual(nil, nil), ShouldBeTrue)
    })

    Convey("TestStringSliceEqual should return false when a == nil  && b != nil", t, func() {
        a := []string(nil)
        b := []string{}
        So(StringSliceEqual(a, b), ShouldBeFalse)
    })

    Convey("TestStringSliceEqual should return false when a != nil  && b != nil", t, func() {
        a := []string{"hello", "world"}
        b := []string{"hello", "goconvey"}
        So(StringSliceEqual(a, b), ShouldBeFalse)
    })
}
```

从上面的测试代码可以看出，每一个Convey语句对应一个测试用例，那么一个函数的多个测试用例可以通过一个测试函数的多个Convey语句来呈现。

测试用例的执行结果如下：

```
=== RUN   TestStringSliceEqual

  TestStringSliceEqual should return true when a != nil  && b != nil ✓


1 total assertion


  TestStringSliceEqual should return true when a == nil  && b == nil ✓


2 total assertions

```

```
  TestStringSliceEqual should return false when a == nil  && b != nil ✓


3 total assertions


  TestStringSliceEqual should return false when a != nil  && b != nil ✓


4 total assertions

--- PASS: TestStringSliceEqual (0.00s)
PASS
ok      infra/alg       0.006s
```

Convey语句的嵌套

Convey语句可以无限嵌套，以体现测试用例之间的关系。需要注意的是，只有最外层的Convey需要传入*testing.T类型的变量t。我们将前面的测试用例通过嵌套的方式写另一个版本：

```go
import (
    "testing"
    . "github.com/smartystreets/goconvey/convey"
)

func TestStringSliceEqual(t *testing.T) {
    Convey("TestStringSliceEqual", t, func() {
        Convey("should return true when a != nil  && b != nil", func() {
            a := []string{"hello", "goconvey"}
            b := []string{"hello", "goconvey"}
            So(StringSliceEqual(a, b), ShouldBeTrue)
        })

```

```
14
15          Convey("should return true when a == nil  && b == nil", func() {
16              So(StringSliceEqual(nil, nil), ShouldBeTrue)
17          })
18
19          Convey("should return false when a == nil  && b != nil", func() {
20              a := []string(nil)
21              b := []string{}
22              So(StringSliceEqual(a, b), ShouldBeFalse)
23          })
24
25          Convey("should return false when a != nil  && b != nil", func() {
26              a := []string{"hello", "world"}
27              b := []string{"hello", "goconvey"}
28              So(StringSliceEqual(a, b), ShouldBeFalse)
29          })
30      })
  }
```

测试用例的执行结果如下：

```
1  === RUN   TestStringSliceEqual
2
3    TestStringSliceEqual
4      should return true when a != nil  && b != nil ✓
5      should return true when a == nil  && b == nil ✓
6      should return false when a == nil  && b != nil ✓
7      should return false when a != nil  && b != nil ✓
8
9
10 4 total assertions
11
12 --- PASS: TestStringSliceEqual (0.00s)
13 PASS
14 ok      infra/alg       0.006s
15
```

可见，Convey语句嵌套的测试日志和Convey语句不嵌套的测试日志的显示有差异，笔者更喜欢这种以测试函数为单位多个测试用例集中显示的形式。

此外，Convey语句嵌套还有一种三层嵌套的惯用法，即按BDD风格来写测试用例，核心点是通过GWT（Given...When...Then）格式来描述测试用例，示例如下：

```
func TestStringSliceEqualIfBothNotNil(t *testing.T) {
    Convey("Given two string slice which are both not nil", t, func() {
        a := []string{"hello", "goconvey"}
        b := []string{"hello", "goconvey"}
        Convey("When the comparision is done", func() {
            result := StringSliceEqual(a, b)
            Convey("Then the result should be true", func() {
                So(result, ShouldBeTrue)
            })
        })
    })
}
```

GWT测试用例的执行结果如下：

```
=== RUN   TestStringSliceEqualIfBothNotNil

  Given two string slice which are both not nil
    When the comparision is done
      Then the result should be true ✓


1 total assertion
```

```
--- PASS: TestStringSliceEqualIfBothNotNil (0.00s)
ok      infra/alg       0.007s
```

按GWT格式写测试用例时，每一组GWT对应一条测试用例，即最内层的Convey语句不像两层嵌套时可以有多个，而是只能有一个Convey语句。

我们依次写出其余三个用例的三层嵌套形式：

```go
func TestStringSliceEqualIfBothNil(t *testing.T) {
    Convey("Given two string slice which are both nil", t, func() {
        var a []string = nil
        var b []string = nil
        Convey("When the comparision is done", func() {
            result := StringSliceEqual(a, b)
            Convey("Then the result should be true", func() {
                So(result, ShouldBeTrue)
            })
        })
    })
}

func TestStringSliceNotEqualIfNotBothNil(t *testing.T) {
    Convey("Given two string slice which are both nil", t, func() {
        a := []string(nil)
        b := []string{}
        Convey("When the comparision is done", func() {
            result := StringSliceEqual(a, b)
            Convey("Then the result should be false", func() {
                So(result, ShouldBeFalse)
            })
        })
    })
}

```

```
func TestStringSliceNotEqualIfBothNotNil(t *testing.T) {
    Convey("Given two string slice which are both not nil", t, func() {
        a := []string{"hello", "world"}
        b := []string{"hello", "goconvey"}
        Convey("When the comparision is done", func() {
            result := StringSliceEqual(a, b)
            Convey("Then the result should be false", func() {
                So(result, ShouldBeFalse)
            })
        })
    })
}
```

我们再将上面的四条用例使用测试套的形式来写，即一个测试函数包含多条用例，每条用例使用Convey语句四层嵌套的惯用法：

```
func TestStringSliceEqual(t *testing.T) {
    Convey("TestStringSliceEqualIfBothNotNil", t, func() {
        Convey("Given two string slice which are both not nil", func() {
            a := []string{"hello", "goconvey"}
            b := []string{"hello", "goconvey"}
            Convey("When the comparision is done", func() {
                result := StringSliceEqual(a, b)
                Convey("Then the result should be true", func() {
                    So(result, ShouldBeTrue)
                })
            })
        })
    })

    Convey("TestStringSliceEqualIfBothNil", t, func() {
        Convey("Given two string slice which are both nil", func() {
            var a []string = nil
            var b []string = nil
            Convey("When the comparision is done", func() {
                result := StringSliceEqual(a, b)
                Convey("Then the result should be true", func() {
```

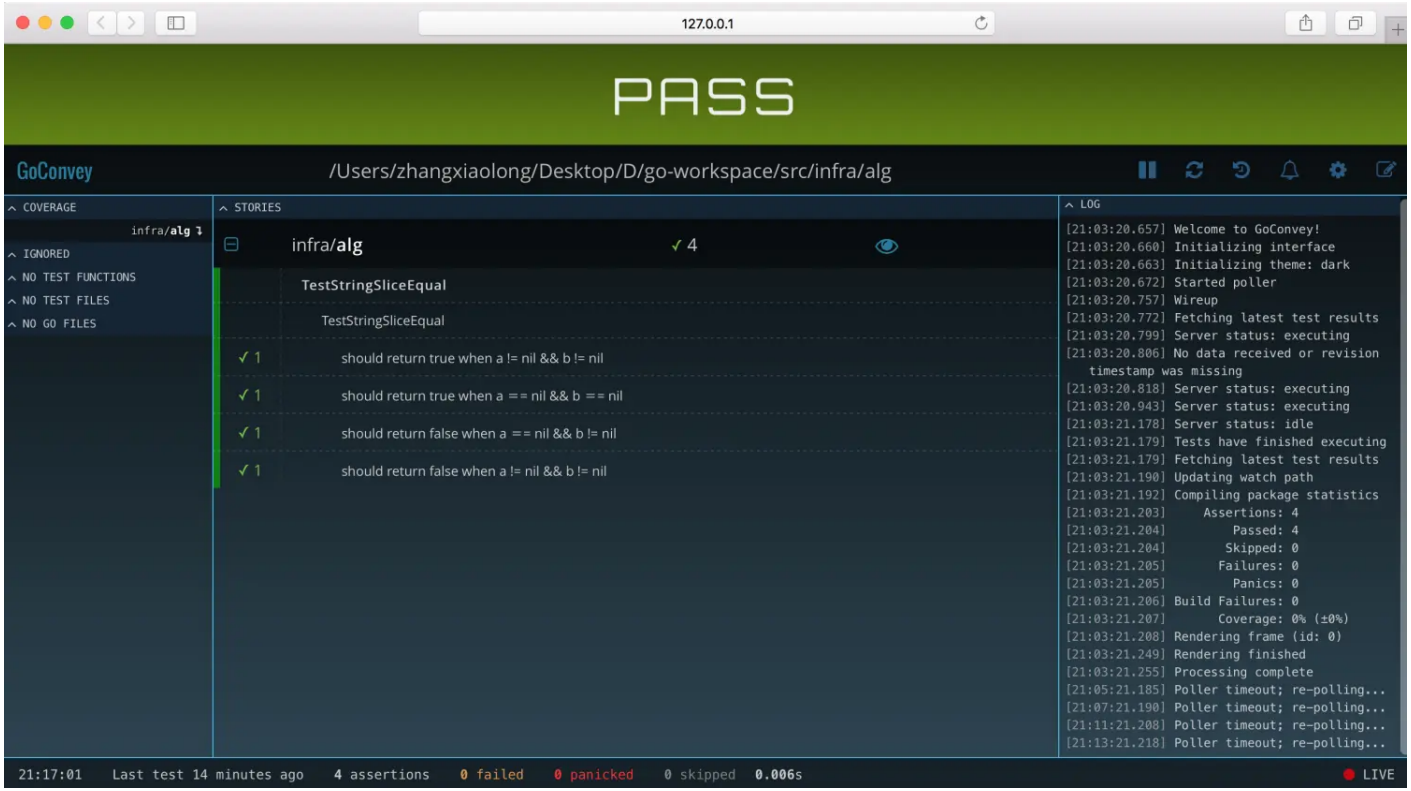
```
                    So(result, ShouldBeTrue)
                })
            })
        })
    })

    Convey("TestStringSliceNotEqualIfNotBothNil", t, func() {
        Convey("Given two string slice which are both nil", func() {
            a := []string(nil)
            b := []string{}
            Convey("When the comparision is done", func() {
                result := StringSliceEqual(a, b)
                Convey("Then the result should be false", func() {
                    So(result, ShouldBeFalse)
                })
            })
        })
    })

    Convey("TestStringSliceNotEqualIfBothNotNil", t, func() {
        Convey("Given two string slice which are both not nil", func() {
            a := []string{"hello", "world"}
            b := []string{"hello", "goconvey"}
            Convey("When the comparision is done", func() {
                result := StringSliceEqual(a, b)
                Convey("Then the result should be false", func() {
                    So(result, ShouldBeFalse)
                })
            })
        })
    })

}
```

Web 界面

GoConvey不仅支持在命令行进行自动化编译测试，而且还支持在 Web 界面进行自动化编译测试。想要使用GoConvey的 Web 界面特性，需要在测试文件所在目录下执行goconvey：

```
$GOPATH/bin/goconvey
```

这时弹出一个页面，如下图所示：



goconvey-web.png

在 Web 界面中:

1. 可以设置界面主题
2. 查看完整的测试结果
3. 使用浏览器提醒等实用功能
4. 自动检测代码变动并编译测试
5. 半自动化书写测试用例
6. 查看测试覆盖率
7. 临时屏蔽某个包的编译测试

Skip

针对想忽略但又不想删掉或注释掉某些断言操作，GoConvey提供了Convey/So的Skip方法：

- SkipConvey函数表明相应的闭包函数将不被执行
- SkipSo函数表明相应的断言将不被执行

当存在SkipConvey或SkipSo时，测试日志中会显式打上"skipped"形式的标记：

- 当测试代码中存在SkipConvey时，相应闭包函数中不管是否为SkipSo，都将被忽略，测试日志中对应的符号仅为一个"⚠"
- 当测试代码Convey语句中存在SkipSo时，测试日志中每个So对应一个"✔"或"✗"，每个SkipSo对应一个"⚠"，按实际顺序排列
- 不管存在SkipConvey还是SkipSo时，测试日志中都有字符串"{n} total assertions (one or more sections skipped)"，其中{n}表示测试中实际已运行的断言语句数

定制断言函数

我们先看一下So函数的声明：

```
func So(actual interface{}, assert Assertion, expected ...interface{})
```

第二个参数为assert，是一个函数变量，它的类型Assertion的定义为：

```
type Assertion func(actual interface{}, expected ...interface{}) string
```

当Assertion的变量的返回值为""时表示断言成功，否则表示失败：

```
const assertionSuccess = ""
```

我们简单实现一个Assertion函数：

```
func ShouldSummerBeComming(actual interface{}, expected ...interface{}) string {
    if actual == "summer" && expected[0] == "comming" {
        return ""
    } else {
        return "summer is not comming!"
    }
}
```

我们仍然在slice_test文件中写一个简单测试：

```
func TestSummer(t *testing.T) {
    Convey("TestSummer", t, func() {
        So("summer", ShouldSummerBeComming, "comming")
        So("winter", ShouldSummerBeComming, "comming")
```

```
    })
}
```

根据ShouldSummerBeComming的实现，Convey语句中第一个So将断言成功，第二个So将断言失败。
我们运行测试，查看执行结果，符合期望：

```
=== RUN   TestSummer

  TestSummer ✓✗


Failures:

  * /Users/zhangxiaolong/Desktop/D/go-workspace/src/infra/alg/slice_test.go
  Line 52:
  summer is not comming!


2 total assertions

--- FAIL: TestSummer (0.00s)
FAIL
exit status 1
FAIL    infra/alg       0.006s
```

小结

Golang虽然自带了单元测试功能，但笔者建议读者使用已经成熟的第三方测试框架。本文主要介绍了GoConvey框架，通过文字结合代码示例讲解基本的使用方法，要点归纳如下：

1. import goconvey包时，前面加点号"."，以减少冗余的代码；

2. 测试函数的名字必须以Test开头，而且参数类型必须为*testing.T；
3. 每个测试用例必须使用Convey语句包裹起来，推荐使用Convey语句的嵌套，即一个函数有一个或多个测试函数，一个测试函数嵌套两层、三层或四层Convey语句；
4. Convey语句的第三个参数习惯以闭包的形式实现，在闭包中通过So语句完成断言；
5. 使用GoConvey框架的 Web 界面特性，作为命令行的补充；
6. 在适当的场景下使用SkipConvey函数或SkipSo函数；
7. 当测试中有需要时，可以定制断言函数。

至此，希望读者已经掌握了GoConvey框架的基本用法，从而可以写出简单优雅的测试代码。

然而，事情并没有这么简单！试想，如果在被测函数中调用了底层rand包的Intn函数，你会如何写测试代码？经过思考，你应该会发现需要给rand包的Intn函数打桩。如何低成本的满足用户各种测试场景的打桩诉求，这正是GoMonkey框架的专长。

最后编辑于： 2022.07.06 23:00:52