

36 | 测试的艺术：依赖注入、表格测试与压力测试

2022-12-31 郑建勋 来自北京



《Go进阶·分布式爬虫实战》

[课程介绍 >](#)



讲述：郑建勋

时长 09:40 大小 8.83M



你好，我是郑建勋。

对代码的功能与逻辑进行测试是项目开发中非常重要的一部分。这节课，我们一起来看看几个在 Go 中进行代码测试的核心技术：单元测试、压力测试与基准测试。它们共同保证了代码的准确性、可靠性与高效性。

单元测试

单元测试又叫做模块测试，它会对程序模块（软件设计的最小单位）进行正确性检验，通常，单元测试是对一个函数封装起来的最小功能进行测试。

在 Go 中，`testing` 包为我们提供了测试的支持。进行代码测试需要将测试函数放置到 `xxx_test.go` 文件中，测试函数以 `TestXxx` 开头，其中 `Xxx` 是测试函数的名称，以大写字母开


头。测试函数以 `testing.T` 类型的指针作为参数，你可以使用这一参数在测试中打印日志、报告测试结果，或者跳过指定测试。



 复制代码

```
1 func TestXxx(t *testing.T)
```

我们用下面这个简单的加法例子来说明一下。首先，在 `add.go` 文件中，写入一个 `Add` 函数实现简单的加法功能。

 复制代码

```
1 // add.go
2 package add
3
4 func Add(a,b int) int{
5     return a+b
6 }
```

接下来在 `add_test.go` 文件中，书写 `TestAdd` 测试函数，并将执行结果与预期进行对比。如果执行结果与预期相符，`t.Log` 打印日志。默认情况下测试是没问题的。但是如果执行结果与预期不符，`t.Fatal` 会报告测试失败。

 复制代码

```
1 // add_test.go
2 package add
3
4 import (
5     "testing"
6 )
7 func TestAdd(t *testing.T) {
8     sum := Add(1, 2)
9     if sum == 3 {
10         t.Log("the result is ok")
11     } else {
12         t.Fatal("the result is wrong")
13     }
14 }
```

要执行测试文件，可以执行 `go test`，如果测试成功，测试结果如下。

```
1 » go test
2 PASS
3 ok      github.com/dreamerjackson/xxx/add    0.013s
```



如果测试结果不符合预期，输出如下。

```
1 === RUN   TestAdd
2     add_test.go:13: the result is wrong
3 --- FAIL: TestAdd (0.00s)
4
5 FAIL
```

根据上面的 **Add** 函数，我们再回顾一下测试需要遵守的规范。

1. 含有单元测试代码的 Go 文件必须以 **_test.go** 结尾，Go 语言的测试工具只认符合这个规则的文件。
2. 单元测试文件名 **_test.go** 前面的部分，最好是被测试的方法所在 Go 文件的文件名。我们例子中，单元测试文件名是 **add_test.go**，这是因为测试的 **Add** 函数在 **add.go** 文件里。
3. 单元测试的函数名必须以 **Test** 开头，是可导出公开的函数。
4. 测试函数的签名必须接收一个指向 **testing.T** 类型的指针，并且不能返回任何值。
5. 函数名最好是 **Test** + 要测试的方法函数名，在我们这个例子中，函数名是 **TestAdd**，表示测试的是 **Add** 这个函数。

下面让我们在项目中对数据库操作的 **sqlldb** 做单元测试，测试一下创建表的功能是否正常。

```
1 func TestSqlldb_CreateTable(t *testing.T) {
2     sqlldb, err := New(
3         WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
4     )
5     assert.Nil(t, err)
6     assert.NotNil(t, sqlldb)
7     // 测试对于无效的配置返回错误
8     name := "test_create_table"
9     var notValidTable = TableData{
```

```

10     TableName: name,
11     ColumnNames: []Field{
12         {Title: "书名", Type: "notValid"},
13         {Title: "URL", Type: "VARCHAR(255)"},
14     },
15     AutoKey: true,
16 }
17 // 延迟删除表
18 defer func() {
19     err := sqldb.DropTable(notValidTable)
20     assert.Nil(t, err)
21 }()
22 // 测试对于有效的配置返回错误
23 err = sqldb.CreateTable(notValidTable)
24 assert.NotNil(t, err)
25 // 测试对于无效的配置返回错误
26 var validTable = TableData{
27     TableName: name,
28     ColumnNames: []Field{
29         {Title: "书名", Type: "MEDIUMTEXT"},
30         {Title: "URL", Type: "VARCHAR(255)"},
31     },
32     AutoKey: true,
33 }
34 err = sqldb.CreateTable(validTable)
35 assert.Nil(t, err)
36 }

```



在这个单元测试中，我们主要测试了创建表的 **CreateTable** 函数的两个功能，包括“在正常情况下能够创建表”和“在异常情况下不能够创建表”。在这里我们没有直接使用 **t.Fatal** 来报告测试失败，而是借助第三方包 github.com/stretchr/testify/assert 来完成测试。

assert 库对 **testing.T** 进行了封装，例如函数 **assert.Nil** 预期传入的参数为 **nil**，而函数 **assert.NotNil** 预期传入的参数不为 **nil**。如果结果不符合预期，则立即报告测试失败。

不过，这样的单元测试其实并不够清晰，特别是当测试的功能逐渐变多的时候，代码还会变得冗余。那么有没有一种测试方法可以优雅地测试多种功能呢？这就不得不提到表格驱动测试了。

表格驱动测试

表格驱动测试也是单元测试的一种，我们直接用一个例子来说明它。下面是我们写的一个字符串分割函数，它的功能类似于 **strings.Split** 函数。

```
1 // split.go
2 package split
3
4 import "strings"
5 func Split(s, sep string) []string {
6     var result []string
7     i := strings.Index(s, sep)
8     for i > -1 {
9         result = append(result, s[:i])
10        s = s[i+len(sep):]
11        i = strings.Index(s, sep)
12    }
13    return append(result, s)
14 }
```

我们如果要对这个函数进行上述所讲的这种单元测试，测试代码是下面的样子。

`reflect.DeepEqual` 是 Go 标准库提供的深度对比函数，它可以对比两个结构是否一致。而如果有多个要测试的用例，`reflect.DeepEqual` 这段对比函数就会重复多次。

```
1 package split
2
3 import (
4     "reflect"
5     "testing"
6 )
7
8 //单元测试
9 func TestSplit(t *testing.T) {
10     got := Split("a/b/c", "/")
11     want := []string{"a", "b", "c"}
12     if !reflect.DeepEqual(want, got) {
13         t.Fatalf("expected: %v, got: %v", want, got)
14     }
15 }
```

为了解决这个问题，我们来看看表格驱动测试的做法。在表格驱动中，我们使用 `Map` 或者数组来组织用例，我们只需要输入值和期望值，在下面的 `for` 循环中就能够复用对比的函数，这就让表格驱动测试在实践中非常受欢迎了。

```

1 // split_test.go
2 package split
3
4 import (
5     "reflect"
6     "testing"
7 )
8
9 func TestSplit(t *testing.T) {
10     tests := map[string]struct {
11         input string
12         sep    string
13         want   []string
14     }{
15         "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
16         "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
17         "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
18         "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
19     }
20
21     for name, tc := range tests {
22         got := Split(tc.input, tc.sep)
23         if !reflect.DeepEqual(tc.want, got) {
24             t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
25         }
26     }
27 }

```



我们也可以把之前测试 `CreateTable` 的函数修改为表格驱动测试。

```

1 func TestSqlldb_CreateTableDriver(t *testing.T) {
2     type args struct {
3         t TableData
4     }
5     name := "test_create_table"
6
7     tests := []struct {
8         name    string
9         args    args
10        wantErr bool
11    }{
12        {
13            name: "create_not_valid_table",
14            args: args{TableData{
15                TableName: name,
16                ColumnNames: []Field{
17                    {Title: "书名", Type: "not_valid"},
18                    {Title: "URL", Type: "VARCHAR(255)"},

```

```
19     },
20     },
21     wantErr: true,
22 },
23 {
24     name: "create_valid_table",
25     args: args{TableData{
26         TableName: name,
27         ColumnNames: []Field{
28             {Title: "书名", Type: "MEDIUMTEXT"},
29             {Title: "URL", Type: "VARCHAR(255)"},
30         },
31     }},
32     wantErr: false,
33 },
34 {
35     name: "create_valid_table_with_primary_key",
36     args: args{TableData{
37         TableName: name,
38         ColumnNames: []Field{
39             {Title: "书名", Type: "MEDIUMTEXT"},
40             {Title: "URL", Type: "VARCHAR(255)"},
41         },
42         AutoKey: true,
43     }},
44     wantErr: false,
45 },
46 }
47
48 sqldb, err := New(
49     WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
50 )
51
52 for _, tt := range tests {
53     err = sqldb.CreateTable(tt.args.t)
54     if tt.wantErr {
55         assert.NotNil(t, err, tt.name)
56     } else {
57         assert.Nil(t, err, tt.name)
58     }
59     sqldb.DropTable(tt.args.t)
60 }
61 }
```

一般来说，我们会给每一个测试加上名字，方便我们在测试出错时打印出具体的用例。在上例中，我们在 `assert.NotNil` 的第三个参数中加上了测试的名字，假如测试出错，打印的结果如下所示。

```

1  === RUN   TestSqlDb_CreateTableDriver
2      sqldb_test.go:98:
3          Error Trace:  /Users/jackson/career/crawler/sqlldb/sqlldb_test.go:98
4          Error:         Expected nil, but got: &mysql.MySQLError{Number: 0x428
5          Test:          TestSqlDb_CreateTableDriver
6          Messages:      create_not_valid_table
7  --- FAIL: TestSqlDb_CreateTableDriver (0.06s)
8
9  FAIL

```

错误信息清晰可见，其中的 **Messages** 就是相关测试用例的名字。

子测试

前面我们看到的例子都是串行调用的，**CreateTable** 的例子也确实不太适合使用并发调用。但是在一些场景下，我们需要通过并发调用来加速测试，这就是子测试为我们做的事情。

使用子测试可以调用 **testing.T** 的 **Run** 函数，子测试会新开一个协程，实现并行。除此之外，子测试还有一个特点，就是会运行所有的测试用例（即使某一个测试用例失败了）。这样在出错时，就可以将多个错误都打印出来。

如下所示，我们用 **t.Run** 子测试来测试之前的 **Split** 函数，并发测试所有用例。

```

1  func TestSplit(t *testing.T) {
2      tests := map[string]struct {
3          input string
4          sep   string
5          want  []string
6      }{
7          "simple":      {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
8          "wrong sep":  {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
9          "no sep":     {input: "abc", sep: "/", want: []string{"abc"}},
10         "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
11     }
12
13     for name, tc := range tests {
14         t.Run(name, func(t *testing.T) {
15             got := Split(tc.input, tc.sep)
16             if !reflect.DeepEqual(tc.want, got) {
17                 t.Fatalf("expected: %#v, got: %#v", tc.want, got)
18             }
19         })
20     }

```




下面让我们用子测试来测试我们 MySQL 库的插入功能。这里我并发测试了四个测试用例，`t.run` 的第一个参数为测试用例的名字。

[复制代码](#)

```

1 func TestSqlDb_InsertTable(t *testing.T) {
2     type args struct {
3         t TableData
4     }
5     tableName := "test_create_table"
6     columnNames := []Field{{Title: "书名", Type: "MEDIUMTEXT"}, {Title: "price", 1
7     tests := []struct {
8         name      string
9         args      args
10        wantErr bool
11    }{
12        {
13            name: "insert_data",
14            args: args{TableData{
15                TableName:  tableName,
16                ColumnNames: columnNames,
17                Args:        []interface{}{"book1", 2},
18                DataCount:  1,
19            }},
20            wantErr: false,
21        },
22        {
23            name: "insert_multi_data",
24            args: args{TableData{
25                TableName:  tableName,
26                ColumnNames: columnNames,
27                Args:        []interface{}{"book3", 88.88, "book4", 99.99},
28                DataCount:  2,
29            }},
30            wantErr: false,
31        },
32        {
33            name: "insert_multi_data_wrong_count",
34            args: args{TableData{
35                TableName:  tableName,
36                ColumnNames: columnNames,
37                Args:        []interface{}{"book3", 88.88, "book4", 99.99},
38                DataCount:  1,
39            }},
40            wantErr: true,
41        },
42        {
43            name: "insert_wrong_data_type",

```

```

44     args: args{TableData{
45         TableName:  tableName,
46         ColumnNames: columnNames,
47         Args:       []interface{}{"book2", "rrr"},
48         DataCount:  1,
49     }},
50     wantErr: true,
51 },
52 }
53
54 sqldb, err := New(
55     WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
56 )
57 err = sqldb.CreateTable(tests[0].args.t)
58 defer sqldb.DropTable(tests[0].args.t)
59 assert.Nil(t, err)
60 for _, tt := range tests {
61     t.Run(tt.name, func(t *testing.T) {
62         err = sqldb.Insert(tt.args.t)
63         if tt.wantErr {
64             assert.NotNil(t, err, tt.name)
65         } else {
66             assert.Nil(t, err, tt.name)
67         }
68     })
69 }
70 }

```



天下无鱼
<https://shikey.com/>

测试结果如下所示。


 复制代码

```

1  » go test -run=TestSqlldb_InsertTable
2  --- FAIL: TestSqlldb_InsertTable (0.07s)
3      --- FAIL: TestSqlldb_InsertTable/insert_wrong_data_type (0.01s)
4          sqlldb_test.go:171:
5              Error Trace:    /Users/jackson/career/crawler/sqlldb/sqlldb_test.
6              Error:         Expected nil, but got: &mysql.MySQLError{Number
7              Test:          TestSqlldb_InsertTable/insert_wrong_data_type
8              Messages:      insert_wrong_data_type
9  FAIL
10 exit status 1
11 FAIL    github.com/dreamerjackson/crawler/sqlldb 0.085s

```

可以看到，当检测到错误时，能够清晰展示出错误用例的信息。

在这里，我们使用了 `go test -run xxx` 参数来指定我们要运行的程序。`-run` 后面跟的是要测试的函数名，测试时会模糊匹配该函数名，符合条件的函数都将被测试。所以在这个例子中，`go test -run=TestSqlDb_InsertTable` 与 `go test -run=InsertTable` 的执行效果是一致的。 <https://shike.com/>

当然，我们还可以加入 `-v` 参数打印出详细的信息。

 复制代码

```
1 » go test -run=InsertTable -v
2 === RUN    TestSqlDb_InsertTable
3 === RUN    TestSqlDb_InsertTable/insert_data
4 === RUN    TestSqlDb_InsertTable/insert_multi_data
5 === RUN    TestSqlDb_InsertTable/insert_multi_data_wrong_count
6 === RUN    TestSqlDb_InsertTable/insert_wrong_data_type
7     sqldb_test.go:171:
8         Error Trace:    /Users/jackson/career/crawler/sqldb/sqldb_test.
9         Error:          Expected nil, but got: &mysql.MySQLError{Number
10        Test:           TestSqlDb_InsertTable/insert_wrong_data_type
11        Messages:       insert_wrong_data_type
12 --- FAIL: TestSqlDb_InsertTable (0.07s)
13 --- PASS: TestSqlDb_InsertTable/insert_data (0.01s)
14 --- PASS: TestSqlDb_InsertTable/insert_multi_data (0.01s)
15 --- PASS: TestSqlDb_InsertTable/insert_multi_data_wrong_count (0.00s)
16 --- FAIL: TestSqlDb_InsertTable/insert_wrong_data_type (0.01s)
17 FAIL
18 exit status 1
19 FAIL     github.com/dreamerjackson/crawler/sqldb 0.084s
```

`-run` 后还可以只指定运行某一个特定的子测试。例如，我们可以只运行 `TestSqlDb_InsertTable` 测试函数下的 `insert_multi_data_wrong_count` 子测试。

 复制代码

```
1 » go test -run=TestSqlDb_InsertTable/insert_multi_data_wrong_count -v
2 === RUN    TestSqlDb_InsertTable
3 === RUN    TestSqlDb_InsertTable/insert_multi_data_wrong_count
4 --- PASS: TestSqlDb_InsertTable (0.04s)
5 --- PASS: TestSqlDb_InsertTable/insert_multi_data_wrong_count (0.00s)
6 PASS
7 ok       github.com/dreamerjackson/crawler/sqldb 0.055s
```

依赖注入

前面我们介绍了单元测试的几种技术。当我们进行单元测试的时候，可能还会遇到一些棘手的依赖问题。例如一个函数需要从下游的多个服务中获取信息并完成后续的操作。在测试时，如果我们需要启动这些依赖，步骤会非常繁琐，有时候甚至无法在本地实现。因此，我们可以使用依赖注入的方式对这些依赖进行 **Mock**，这种方式也能够让我们灵活地控制下游返回的数据。

我们以项目中的 **Flush()** 为例，在这个例子中，最后的 **s.db.Insert** 需要我们把数据插入数据库。

 复制代码

```
1 func (s *SQLStorage) Flush() error {
2     if len(s.dataDocker) == 0 {
3         return nil
4     }
5
6     defer func() {
7         s.dataDocker = nil
8     }()
9     ...
10    return s.db.Insert(sqldb.TableData{
11        TableName:    s.dataDocker[0].GetTableName(),
12        ColumnNames:  getFields(s.dataDocker[0]),
13        Args:          args,
14        DataCount:    len(s.dataDocker),
15    })
16 }
```

但我们其实并不是真的需要一个数据库。让我们新建一个测试文件 **sqlstorage_test.go**，然后实现数据库 **Dber** 接口。

 复制代码

```
1 // sqlstorage_test.go
2 type mysqlldb struct {
3 }
4
5 func (m mysqlldb) CreateTable(t sqldb.TableData) error {
6     return nil
7 }
8
9 func (m mysqlldb) Insert(t sqldb.TableData) error {
10    return nil
11 }
```

接着，我们就可以将 `mysqlldb` 注入到 `SQLStorage` 结构中，单元测试如下所示。



```
1 func TestSQLStorage_Flush(t *testing.T) {
2     type fields struct {
3         dataDocke []*spider.DataCell
4         options    options
5     }
6     tests := []struct {
7         name      string
8         fields     fields
9         wantErr   bool
10    }{
11        {name: "empty", wantErr: false},
12        {name: "no Rule filed", fields: fields{dataDocke: []*spider.DataCell{
13            {Data: map[string]interface{}{"url": "<http://xxx.com>"}}},
14        }, wantErr: true},
15    }
16    for _, tt := range tests {
17        t.Run(tt.name, func(t *testing.T) {
18            s := &SQLStorage{
19                dataDocke: tt.fields.dataDocke,
20                db:        mysqlldb{},
21                options:   tt.fields.options,
22            }
23            if err := s.Flush(); (err != nil) != tt.wantErr {
24                t.Errorf("Flush() error = %v, wantErr %v", err, tt.wantErr)
25            }
26            assert.Nil(t, s.dataDocke)
27        })
28    }
29 }
```

测试用例中测试了没有 `Rule` 字段时的情形，但是程序却直接 `panic` 了。这就是单元测试的意义所在，它可以为我们找到一些特殊的输入，确认它们是否仍然符合预期。

经过测试我们发现，由于我们将接口强制转换为了 `string`，当接口类型不匹配时就会直接 `panic`。

复制代码

```
1 ruleName := datacell.Data["Rule"].(string)
2 taskName := datacell.Data["Task"].(string)
```

要避免这种情况，我们可以对异常情况进行判断，完整的测试你可以查看 [🔗 最新的项目代码](#)。



```
1 if ruleName, ok = datacell.Data["Rule"].(string); !ok {
2     return errors.New("no rule field")
3 }
4
5 if taskName, ok = datacell.Data["Task"].(string); !ok {
6     return errors.New("no task field")
7 }
```

压力测试

有时候，我们还希望对程序进行压力测试，它可以测试随机场景、排除偶然因素、测试函数稳定性等等。

实现压力测试的方法和工具有很多，例如 **ab**、**wrk**。合理的压力测试通常需要结合实际项目来设计。我们也可以通过书写 **Shell** 脚本来进行压力测试，如下脚本中，我们可以用 **go test -c** 为测试函数生成二进制文件，并循环调用测试函数。

复制代码

```
1 # pressure.sh
2 go test -c # -c会生成可执行文件
3
4 PKG=$(basename $(pwd)) # 获取当前路径的最后一个名字，即为文件夹的名字
5 echo $PKG
6 while true ; do
7     export GOMAXPROCS=$(( 1 + $(RANDOM % 128) )) # 随机的GOMAXPROCS
8     ./${PKG}.test $@ 2>&1 # $@代表可以加入参数 2>&1代表错误输出到控制台
9 done
```

以之前的加法函数为例，执行下面的命令即可对测试函数进行压力测试。其中，**-test.v** 为运行参数，用于输出详细信息。

复制代码

```
1 > /pressure.sh -test.v
2
3 PASS
4 === RUN    TestAdd
5 --- PASS: TestAdd (0.00s)
6     add_test.go:17: the result is ok
```

```
7 PASS
8 === RUN    TestAdd
9 --- PASS: TestAdd (0.00s)
10      add_test.go:17: the result is ok
```



基准测试

Go 测试包中内置了 **Benchmarks** 基准测试，它可以对比改进后和改进前的函数，查看性能提升效果，也可以供我们探索一些 Go 的特性。

我们可以用基准测试来对比之前的接口调用与直接函数调用。

 复制代码

```
1 package escape
2
3 import "testing"
4
5 type Sumifier interface{ Add(a, b int32) int32 }
6
7 type Sumer struct{ id int32 }
8
9 func (math Sumer) Add(a, b int32) int32 { return a + b }
10
11 type SumerPointer struct{ id int32 }
12
13 func (math *SumerPointer) Add(a, b int32) int32 { return a + b }
14
15 func BenchmarkDirect(b *testing.B) {
16     adder := Sumer{id: 6754}
17     b.ResetTimer()
18     for i := 0; i < b.N; i++ {
19         adder.Add(10, 12)
20     }
21 }
22
23 func BenchmarkInterface(b *testing.B) {
24     adder := Sumer{id: 6754}
25     b.ResetTimer()
26     for i := 0; i < b.N; i++ {
27         Sumifier(adder).Add(10, 12)
28     }
29 }
30
31 func BenchmarkInterfacePointer(b *testing.B) {
32     adder := &SumerPointer{id: 6754}
33     b.ResetTimer()
34     for i := 0; i < b.N; i++ {
```

```
35     Sumifier(adder).Add(10, 12)
36 }
37 }
```



天下无鱼

<https://shikey.com/>

`go test` 可以加入 `-gcflags` 指定编译器的行为。例如这里的 `-gcflags "-N -l"` 表示禁止编译器的优化与内联，`-bench=.` 表示执行基准测试，这样我们就可以对比前后几个函数的性能差异了。

 复制代码

```
1 » go test -gcflags "-N -l" -bench=.
2 BenchmarkDirect-12                535487740          1.95 ns/op
3 BenchmarkInterface-12             76026812          14.6 ns/op
4 BenchmarkInterfacePointer-12      517756519          2.37 ns/op
```

Benchmark 测试时还可以指定一些其他运行参数，例如 `-benchmem` 可以打印每次函数的内存分配情况，`-cpuprofile`、`-memprofile` 还能收集程序的 CPU 和内存的 profile 文件。

 复制代码

```
1 go test ./fibonacci \
2     -bench BenchmarkSuite \
3     -benchmem \
4     -cpuprofile=cpu.out \
5     -memprofile=mem.out
```

这些生成的样本文件我们可以使用 `pprof` 工具进行可视化分析。关于 `pprof` 工具，我们在之后还会做详细介绍。

总结

这节课，我们介绍了 Go 中的多种测试技术，包括单元测试、表格驱动测试、子测试、基准测试、压力测试、依赖注入等。灵活地使用这些测试技术可以提前发现系统存在的性能问题，在后面的课程中，我们还会介绍代码覆盖率和模糊测试等新的测试技术。

课后题

你觉得 `reflect.DeepEqual` 的缺点是什么，有其他的替代方案吗？对于一个复杂的结构，如果 `reflect.DeepEqual` 返回了 `false`，怎么知道是哪一個字段不一致呢？

欢迎你在留言区与我交流讨论，我们下节课见！



天下无鱼

<https://shikey.com/>

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享



赞 2



提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇

35 | 未雨绸缪：怎样通过静态与动态代码扫描保证代码质量？

下一篇

37 | 工具背后的工具：从代码覆盖率到模糊测试

精选留言 (1)



写留言



拾掇拾掇

2023-01-09 来自浙江

goland 会自动帮你建好表格测试

