

Go Test 单元测试简明教程

Go 简明教程系列文章链接:

- [Go 语言简明教程](#) (Aug 6, 2019)
- [Go Gin 简明教程](#) (Aug 7, 2019)
- [Go2 新特性简明教程](#) (Aug 15, 2019)
- [Go Protobuf 简明教程](#) (Jan 11, 2020)
- [Go RPC & TLS 鉴权简明教程](#) (Jan 13, 2020)
- [Go WebAssembly \(Wasm\) 简明教程](#) (Jan 23, 2020)
- [Go Test 单元测试简明教程](#) (Feb 10, 2020)
- [Go Mock \(gomock\)简明教程](#) (Feb 14, 2020)
- [Go Mmap 文件内存映射简明教程](#) (Apr 20, 2020)
- [Go Context 并发编程简明教程](#) (Apr 20, 2020)



1 如何写好单元测试

单元测试(Unit Tests, UT) 是一个优秀项目不可或缺的一部分，特别是在一些频繁变动和多人合作开发的项目中尤为重要。你或多或少都会有因为自己的提交，导致应用挂掉或服务宕机的经历。如果这个时候你的修改导致测试用例失败，你再重新审视自己的修改，发现之前的修改还有一些特殊场景没有包含，恭喜你减少了一次上库失误。也会有这样的情况，项目很大，启动环境很复杂，你优化了一个函数的性能，或是添加了某个新的特性，如果部署在正式环境上之后再进行测试，成本太高。对于这种场景，几个小小的测试用例或许就能够覆盖大部分的

测试场景。而且在开发过程中，效率最高的莫过于所见即所得了，单元测试也能够帮助你做到这一点，试想一下，假如你一口气写完一千行代码，debug 的过程也不会轻松，如果在这个过程中，对于一些逻辑较为复杂的函数，同时添加一些测试用例，即时确保正确性，最后集成的时候，会是另外一番体验。

如何写好单元测试呢？

首先，学会写测试用例。比如如何测试单个函数/方法；比如如何做基准测试；比如如何写出简洁精炼的测试代码；再比如遇到数据库访问等的方法调用时，如何 mock。

然后，写可测试的代码。高内聚，低耦合 是软件工程的原则，同样，对测试而言，函数/方法写法不同，测试难度也是不一样的。职责单一，参数类型简单，与其他函数耦合度低的函数往往更容易测试。我们经常会说，“这种代码没法测试”，这种时候，就得思考函数的写法可不可以改得更好一些。为了代码可测试而重构是值得的。

接下来将介绍如何使用 Go 语言的标准库 `testing` 进行单元测试。

2 一个简单例子

Go 语言推荐测试文件和源代码文件放在一块，测试文件以 `_test.go` 结尾。比如，当前 package 有 `calc.go` 一个文件，我们想测试 `calc.go` 中的 `Add` 和 `Mul` 函数，那么应该新建 `calc_test.go` 作为测试文件。

```
1  example/
2      |--calc.go
3      |--calc_test.go
```

假如 `calc.go` 的代码如下：

```
1  package main
2
3  func Add(a int, b int) int {
4      return a + b
5  }
6
7  func Mul(a int, b int) int {
8      return a * b
9  }
```

那么 `calc_test.go` 中的测试用例可以这么写：

```
1  package main
2
3  import "testing"
4
5  func TestAdd(t *testing.T) {
6      if ans := Add(1, 2); ans != 3 {
7          t.Errorf("1 + 2 expected be 3, but %d got", ans)
```

```
8      }
9
10     if ans := Add(-10, -20); ans != -30 {
11         t.Errorf("-10 + -20 expected be -30, but %d got", ans)
12     }
13 }
```

- 测试用例名称一般命名为 `Test` 加上待测试的方法名。
- 测试用的参数有且只有一个，在这里是 `t *testing.T`。
- 基准测试(benchmark)的参数是 `*testing.B`，`TestMain` 的参数是 `*testing.M` 类型。

运行 `go test`，该 `package` 下所有的测试用例都会被执行。

```
1  $ go test
2  ok      example 0.009s
```

或 `go test -v`，`-v` 参数会显示每个用例的测试结果，另外 `-cover` 参数可以查看覆盖率。

```
1  $ go test -v
2  === RUN   TestAdd
3  --- PASS: TestAdd (0.00s)
4  === RUN   TestMul
5  --- PASS: TestMul (0.00s)
6  PASS
7  ok      example 0.007s
```

如果只想运行其中的一个用例，例如 `TestAdd`，可以用 `-run` 参数指定，该参数支持通配符 `*`，和部分正则表达式，例如 `^`、`$`。

```
1  $ go test -run TestAdd -v
2  === RUN   TestAdd
3  --- PASS: TestAdd (0.00s)
4  PASS
5  ok      example 0.007s
```

3 子测试(Subtests)

子测试是 Go 语言内置支持的，可以在某个测试用例中，根据测试场景使用 `t.Run` 创建不同的子测试用例：

```
1  // calc_test.go
2
3  func TestMul(t *testing.T) {
```

```

4      t.Run("pos", func(t *testing.T) {
5          if Mul(2, 3) != 6 {
6              t.Fatal("fail")
7          }
8      })
9      t.Run("neg", func(t *testing.T) {
10         if Mul(2, -3) != -6 {
11             t.Fatal("fail")
12         }
13     })
14 }
15 }

```

- 之前的例子测试失败时使用 `t.Error/t.Errorf`，这个例子中使用 `t.Fatal/t.Fatalf`，区别在于前者遇错不停，还会继续执行其他的测试用例，后者遇错即停。

运行某个测试用例的子测试：

```

1  $ go test -run TestMul/pos -v
2  === RUN    TestMul
3  === RUN    TestMul/pos
4  --- PASS: TestMul (0.00s)
5  --- PASS: TestMul/pos (0.00s)
6  PASS
7  ok      example 0.008s

```

对于多个子测试的场景，更推荐如下的写法(table-driven tests)：

```

1  // calc_test.go
2  func TestMul(t *testing.T) {
3      cases := []struct {
4          Name      string
5          A, B, Expected int
6      }{
7          {"pos", 2, 3, 6},
8          {"neg", 2, -3, -6},
9          {"zero", 2, 0, 0},
10     }
11
12     for _, c := range cases {
13         t.Run(c.Name, func(t *testing.T) {
14             if ans := Mul(c.A, c.B); ans != c.Expected {
15                 t.Fatalf("%d * %d expected %d, but %d got",
16                     c.A, c.B, c.Expected, ans)
17             }
18         })
19     }
20 }

```

```
17         }
18     })
19 }
20 }
```

所有用例的数据组织在切片 `cases` 中，看起来就像一张表，借助循环创建子测试。这样写的好处有：

- 新增用例非常简单，只需给 `cases` 新增一条测试数据即可。
- 测试代码可读性好，直观地能够看到每个子测试的参数和期待的返回值。
- 用例失败时，报错信息的格式比较统一，测试报告易于阅读。

如果数据量较大，或是一些二进制数据，推荐使用相对路径从文件中读取。

4 帮助函数(helpers)

对一些重复的逻辑，抽取出来作为公共的帮助函数(helpers)，可以增加测试代码的可读性和可维护性。借助帮助函数，可以让测试用例的主逻辑看起来更清晰。

例如，我们可以将创建子测试的逻辑抽取出来：

```
1  // calc_test.go
2  package main
3
4  import "testing"
5
6  type calcCase struct{ A, B, Expected int }
7
8  func createMulTestCase(t *testing.T, c *calcCase) {
9      // t.Helper()
10     if ans := Mul(c.A, c.B); ans != c.Expected {
11         t.Fatalf("%d * %d expected %d, but %d got",
12             c.A, c.B, c.Expected, ans)
13     }
14
15 }
16
17 func TestMul(t *testing.T) {
18     createMulTestCase(t, &calcCase{2, 3, 6})
19     createMulTestCase(t, &calcCase{2, -3, -6})
20     createMulTestCase(t, &calcCase{2, 0, 1}) // wrong case
21 }
```

在这里，我们故意创建了一个错误的测试用例，运行 `go test`，用例失败，会报告错误发生的文件和行号信息：

```
1  $ go test
2  --- FAIL: TestMul (0.00s)
3      calc_test.go:11: 2 * 0 expected 1, but 0 got
4  FAIL
5  exit status 1
6  FAIL    example 0.007s
```

可以看到，错误发生在第11行，也就是帮助函数 `createMulTestCase` 内部。18, 19, 20行都调用了该方法，我们第一时间并不能够确定是哪一行发生了错误。有些帮助函数还可能在不同的函数中被调用，报错信息都在同一处，不方便问题定位。因此，Go 语言在 1.9 版本中引入了 `t.Helper()`，用于标注该函数是帮助函数，报错时将输出帮助函数调用者的信息，而不是帮助函数的内部信息。

修改 `createMulTestCase`，调用 `t.Helper()`

```
1  func createMulTestCase(c *calcCase, t *testing.T) {
2      t.Helper()
3      t.Run(c.Name, func(t *testing.T) {
4          if ans := Mul(c.A, c.B); ans != c.Expected {
5              t.Fatalf("%d * %d expected %d, but %d got",
6                  c.A, c.B, c.Expected, ans)
7          }
8      })
9  }
```

运行 `go test`，报错信息如下，可以非常清晰地知道，错误发生在第 20 行。

```
1  $ go test
2  --- FAIL: TestMul (0.00s)
3      calc_test.go:20: 2 * 0 expected 1, but 0 got
4  FAIL
5  exit status 1
6  FAIL    example 0.006s
```

关于 `helper` 函数的 2 个建议：

- 不要返回错误，帮助函数内部直接使用 `t.Error` 或 `t.Fatal` 即可，在用例主逻辑中不会因为太多的错误处理代码，影响可读性。
- 调用 `t.Helper()` 让报错信息更准确，有助于定位。

5 setup 和 teardown

如果在同一个测试文件中，每一个测试用例运行前后的逻辑是相同的，一般会写在 `setup` 和 `teardown` 函数中。例如执行前需要实例化待测试的对象，如果这个对象比较复杂，很适合将这一部分逻辑提取出来；执行后，

可能会做一些资源回收类的工作，例如关闭网络连接，释放文件等。标准库 `testing` 提供了这样的机制：

```
1 func setup() {
2     fmt.Println("Before all tests")
3 }
4
5 func teardown() {
6     fmt.Println("After all tests")
7 }
8
9 func Test1(t *testing.T) {
10    fmt.Println("I'm test1")
11 }
12
13 func Test2(t *testing.T) {
14    fmt.Println("I'm test2")
15 }
16
17 func TestMain(m *testing.M) {
18    setup()
19    code := m.Run()
20    teardown()
21    os.Exit(code)
22 }
23
```

- 在这个测试文件中，包含有2个测试用例，`Test1` 和 `Test2`。
- 如果测试文件中包含函数 `TestMain`，那么生成的测试将调用 `TestMain(m)`，而不是直接运行测试。
- 调用 `m.Run()` 触发所有测试用例的执行，并使用 `os.Exit()` 处理返回的状态码，如果不为0，说明有用例失败。
- 因此可以在调用 `m.Run()` 前后做一些额外的准备(`setup`)和回收(`teardown`)工作。

执行 `go test`，将会输出

```
1 $ go test
2 Before all tests
3 I'm test1
4 I'm test2
5 PASS
6 After all tests
7 ok      example 0.006s
```

6 网络测试(Network)

6.1 TCP/HTTP

假设需要测试某个 API 接口的 handler 能够正常工作，例如 helloHandler

```
1 func helloHandler(w http.ResponseWriter, r *http.Request) {
2     w.Write([]byte("hello world"))
3 }
```

那我们可以创建真实的网络连接进行测试：

```
1 // test code
2 import (
3     "io/ioutil"
4     "net"
5     "net/http"
6     "testing"
7 )
8
9 func handleError(t *testing.T, err error) {
10     t.Helper()
11     if err != nil {
12         t.Fatal("failed", err)
13     }
14 }
15
16 func TestConn(t *testing.T) {
17     ln, err := net.Listen("tcp", "127.0.0.1:0")
18     handleError(t, err)
19     defer ln.Close()
20
21     http.HandleFunc("/hello", helloHandler)
22     go http.Serve(ln, nil)
23
24     resp, err := http.Get("http://" + ln.Addr().String() + "/hello")
25     handleError(t, err)
26
27     defer resp.Body.Close()
28     body, err := ioutil.ReadAll(resp.Body)
29     handleError(t, err)
30
31     if string(body) != "hello world" {
32         t.Fatal("expected hello world, but got", string(body))
33     }
34 }
```



```
33     }
34 }
```

- `net.Listen("tcp", "127.0.0.1:0")`：监听一个未被占用的端口，并返回 `Listener`。
- 调用 `http.Serve(ln, nil)` 启动 `http` 服务。
- 使用 `http.Get` 发起一个 `Get` 请求，检查返回值是否正确。
- 尽量不对 `http` 和 `net` 库使用 `mock`，这样可以覆盖较为真实的场景。

6.2 httptest

针对 `http` 开发的场景，使用标准库 `net/http/httptest` 进行测试更为高效。

上述的测试用例改写如下：

```
1  // test code
2  import (
3      "io/ioutil"
4      "net/http"
5      "net/http/httptest"
6      "testing"
7  )
8
9  func TestConn(t *testing.T) {
10     req := httptest.NewRequest("GET", "http://example.com/foo", nil)
11     w := httptest.NewRecorder()
12     helloHandler(w, req)
13     bytes, _ := ioutil.ReadAll(w.Result().Body)
14
15     if string(bytes) != "hello world" {
16         t.Fatal("expected hello world, but got", string(bytes))
17     }
18 }
```

使用 `httptest` 模拟请求对象(`req`)和响应对象(`w`)，达到了相同的目的。

7 Benchmark 基准测试

基准测试用例的定义如下：

```
1  func BenchmarkName(b *testing.B){
2      // ...
3  }
```

- 函数名必须以 `Benchmark` 开头，后面一般跟待测试的函数名

- 参数为 `b *testing.B`。
- 执行基准测试时，需要添加 `-bench` 参数。

例如：

```
1 func BenchmarkHello(b *testing.B) {
2     for i := 0; i < b.N; i++ {
3         fmt.Sprintf("hello")
4     }
5 }
```

```
1 $ go test -benchmem -bench .
2 ...
3 BenchmarkHello-16    15991854    71.6 ns/op    5 B/op    1 allocs/op
4 ...
```

基准测试报告每一列值对应的含义如下：

```
1 type BenchmarkResult struct {
2     N          int          // 迭代次数
3     T          time.Duration // 基准测试花费的时间
4     Bytes      int64        // 一次迭代处理的字节数
5     MemAllocs  uint64       // 总的分配内存的次数
6     MemBytes   uint64       // 总的分配内存的字节数
7 }
```

如果在运行前基准测试需要一些耗时的配置，则可以使用 `b.ResetTimer()` 先重置定时器，例如：

```
1 func BenchmarkHello(b *testing.B) {
2     ... // 耗时操作
3     b.ResetTimer()
4     for i := 0; i < b.N; i++ {
5         fmt.Sprintf("hello")
6     }
7 }
```

使用 `RunParallel` 测试并发性能

```
1 func BenchmarkParallel(b *testing.B) {
2     templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
3     b.RunParallel(func(pb *testing.PB) {
4         var buf bytes.Buffer
```

```
5         for pb.Next() {
6             // 所有 goroutine 一起，循环一共执行 b.N 次
7             buf.Reset()
8             templ.Execute(&buf, "World")
9         }
10    })
11 }
```

```
1  $ go test -benchmem -bench .
2  ...
3  BenchmarkParallel-16    3325430      375 ns/op    272 B/op    8 allocs/op
4  ...
```

附 参考

- [Go Mock \(gomock\) 简明教程](#)
- [testing - golang.org](#)
- [Advanced Testing in Go - sourcegraph.com](#)

专题: [Go 简明教程](#)

本文发表于 2020-02-10, 最后修改于 2021-02-06。

本站永久域名「geektutu.com」, 也可搜索「极客兔兔」找到我。

[上一篇](#) « 7天用Go从零实现分布式缓存GeeCache

[下一篇](#) » 动手写分布式缓存 - GeeCache第一天 LRU 缓存淘汰策略

赞赏支持



推荐阅读

Go Reflect 提高反射性能

发表于2020-12-06, 阅读约28分钟

动手写RPC框架 - GeeRPC第六天 负载均衡(load balance)

发表于2020-10-08, 阅读约38分钟

博客折腾记(七) - Gitalk Plus

发表于2019-08-23, 阅读约9分钟

#关于我 (9) #Go (48) #百宝箱 (2) #Cheat Sheet (1) #Go语言高性能编程 (20) #友链 (1) #Pandas (3)
#机器学习 (9) #TensorFlow (9) #mnist (5) #Python (10) #强化学习 (3) #OpenAI gym (4) #DQN (1)
#Q-Learning (1) #CNN (1) #TensorFlow 2 (10) #官方文档 (10) #Rust (1)

1 条评论

未登录用户 ▾



说点什么

① 支持 Markdown 语法

使用 GitHub 登录

预览

wilgx0 发表于 5 个月前



您的爱，太阳一般温暖，春风一般和煦，清泉一般甘甜。您的爱，比父爱更严峻，比母爱更细腻，比友爱更纯洁。您的爱，天下最伟大，最高洁。

Go语言动手写Web框架 - Gee第一天 http.Handler

17 评论 • 1天前

 mesiyar —— mark 跟进学习

...

Go语言动手写Web框架 - Gee第三天 路由 Router

30 评论 • 2天前

 GaloisZhou —— 很棒的学习资料！ 有一个问题 get /a/:b get /a/c /a/x 也是去到 /a/c -

...

Go 空结构体 struct{} 的使用


2 评论 • 5天前

 xiezhenyu19970913 —— 学到了很多细节，感谢!

on

动手写ORM框架 - GeeORM第五天 实现钩子 (Hooks)


3 评论 • 24天前

 shiluoye —— 用MethodByName(method) 实在太骚了，为什么不用interface，gorm的

...

Gitalk Plus

© 2021 - 极客兔兔 - 沪ICP备18001798号-1

Powered by [Hexo](#) | Theme [Geektutu](#)  [Star](#)

