



下载APP



36 | 代码测试（上）：如何编写 Go 语言单元测试和性能测试用例？

2021-08-17 孔令飞

《Go 语言项目开发实战》

[课程介绍 >](#)



讲述：孔令飞

时长 15:53 大小 14.56M



你好，我是孔令飞。

从今天开始，我们就进入了服务测试模块，这一模块主要介绍如何测试我们的 Go 项目。

在 Go 项目开发中，我们不仅要开发功能，更重要的是确保这些功能稳定可靠，并且拥有一个不错的性能。要确保这些，就要对代码进行测试。开发人员通常会进行单元测试和性能测试，分别用来测试代码的功能是否正常和代码的性能是否满足需求。



每种语言通常都有自己的测试包 / 模块，Go 语言也不例外。在 Go 中，我们可以通过 testing 包对代码进行单元测试和性能测试。这一讲，我会用一些示例来讲解如何编写单

元测试和性能测试用例，下一讲则会介绍如何编写其他的测试类型，并介绍 IAM 项目的测试用例。

如何测试 Go 代码？

Go 语言有自带的测试框架 `testing`，可以用来实现单元测试（T 类型）和性能测试（B 类型），通过 `go test` 命令来执行单元测试和性能测试。

`go test` 执行测试用例时，是以 `go` 包为单位进行测试的。执行时需要指定包名，比如 `go test` 包名，如果没有指定包名，默认会选择执行命令时所在的包。`go test` 在执行时，会遍历以 `_test.go` 结尾的源码文件，执行其中以 `Test`、`Benchmark`、`Example` 开头的测试函数。

为了演示如何编写测试用例，我预先编写了 4 个函数。假设这些函数保存在 `test` 目录下的 `math.go` 文件中，包名为 `test`，`math.go` 代码如下：

[复制代码](#)

```
1 package test
2
3 import (
4     "fmt"
5     "math"
6     "math/rand"
7 )
8
9 // Abs returns the absolute value of x.
10 func Abs(x float64) float64 {
11     return math.Abs(x)
12 }
13
14 // Max returns the larger of x or y.
15 func Max(x, y float64) float64 {
16     return math.Max(x, y)
17 }
18
19 // Min returns the smaller of x or y.
20 func Min(x, y float64) float64 {
21     return math.Min(x, y)
22 }
23
24 // RandInt returns a non-negative pseudo-random int from the default Source.
25 func RandInt() int {
26     return rand.Int()
27 }
```

在这一讲后面的内容中，我会演示如何编写测试用例，来对这些函数进行单元测试和性能测试。下面让我们先来看下测试命名规范。

测试命名规范

在我们对 Go 代码进行测试时，需要编写测试文件、测试函数、测试变量，它们都需要遵循一定的规范。这些规范有些来自于官方，有些则来自于社区。这里，我分别来介绍下测试文件、包、测试函数和测试变量的命名规范。

测试文件的命名规范

Go 的测试文件名必须以 `_test.go` 结尾。例如，如果我们有一个名为 `person.go` 的文件，那它的测试文件必须命名为 `person_test.go`。这样做是因为，Go 需要区分哪些文件是测试文件。这些测试文件可以被 `go test` 命令行工具加载，用来测试我们编写的代码，但会被 Go 的构建程序忽略掉，因为 Go 程序的运行不需要这些测试代码。

包的命名规范

Go 的测试可以分为白盒测试和黑盒测试。

白盒测试：将测试和生产代码放在同一个 Go 包中，这使我们可以同时测试 Go 包中可导出和不可导出的标识符。当我们编写的单元测试需要访问 Go 包中不可导出的变量、函数和方法时，就需要编写白盒测试用例。

黑盒测试：将测试和生产代码放在不同的 Go 包中。这时，我们仅可以测试 Go 包的可导出标识符。这意味着我们的测试包将无法访问生产代码中的任何内部函数、变量或常量。

在白盒测试中，Go 的测试包名称需要跟被测试的包名保持一致，例如：`person.go` 定义了一个 `person` 包，则 `person_test.go` 的包名也要为 `person`，这也意味着 `person.go` 和 `person_test.go` 都要在同一个目录中。

在黑盒测试中，Go 的测试包名称需要跟被测试的包名不同，但仍然可以存放在同一个目录下。比如，`person.go` 定义了一个 `person` 包，则 `person_test.go` 的包名需要跟

person不同，通常我们命名为person_test。


如果不是需要使用黑盒测试，我们在做单元测试时要尽量使用白盒测试。一方面，这是 go test 工具的默认行为；另一方面，使用白盒测试，我们可以测试和使用不可导出的标识符。

测试文件和包的命名规范，由 Go 语言及 go test 工具来强约束。

函数的命名规范

测试用例函数必须以Test、Benchmark、Example开头，例如TestXxx、BenchmarkXxx、ExampleXxx，Xxx部分为任意字母数字的组合，首字母大写。这是由 Go 语言和 go test 工具来进行约束的，Xxx一般是需要测试的函数名。

除此之外，还有一些社区的约束，这些约束不是强制的，但是遵循这些约束会让我们的测试函数名更加易懂。例如，我们有以下函数：

 复制代码

```
1 package main
2
3 type Person struct {
4     age int64
5 }
6
7 func (p *Person) older(other *Person) bool {
8     return p.age > other.age
9 }
```

很显然，我们可以把测试函数命名为TestOlder，这个名称可以很清晰地说明它是Older函数的测试用例。但是，如果我们想用多个测试用例来测试TestOlder函数，这些测试用例该如何命名呢？也许你会说，我们命名为TestOlder1、TestOlder2不就行了？

其实，还有其他更好的命名方法。比如，这种情况下，我们可以将函数命名为TestOlderXxx，其中Xxx代表Older函数的某个场景描述。例如，strings.Compare函数有如下测试函数：TestCompare、TestCompareIdenticalString、TestCompareStrings。

变量的命名规范

Go 语言和 go test 没有对变量的命名做任何约束。但是，在编写单元测试用例时，还是有一些规范值得我们去遵守。

单元测试用例通常会有一个实际的输出，在单元测试中，我们会将预期的输出跟实际的输出进行对比，来判断单元测试是否通过。为了清晰地表达函数的实际输出和预期输出，可以将这两类输出命名为expected/actual，或者got/want。例如：

[复制代码](#)

```
1 if c.expected != actual {  
2     t.Fatalf("Expected User-Agent '%s' does not match '%s'", c.expected, actual)  
3 }
```

或者：

[复制代码](#)

```
1 if got, want := diags[3].Description().Summary, undeclPlural; got != want {  
2     t.Errorf("wrong summary for diagnostic 3\ngot:  %s\nwant: %s", got, want)  
3 }
```

其他的变量命名，我们可以遵循 Go 语言推荐的变量命名方法，例如：

Go 中的变量名应该短而不是长，对于范围有限的局部变量来说尤其如此。

变量离声明越远，对名称的描述性要求越高。

像循环、索引之类的变量，名称可以是单个字母（i）。如果是不常见的变量和全局变量，变量名就需要具有更多的描述性。

上面，我介绍了 Go 测试的一些基础知识。接下来，我们来看看如何编写单元测试用例和性能测试用例。

单元测试

单元测试用例函数以 Test 开头，例如 TestXxx 或 Test_xxx（Xxx 部分为任意字母数字组合，首字母大写）。函数参数必须是 *testing.T，可以使用该类型来记录错误或测

试状态。

我们可以调用 `testing.T` 的 `Error`、`Errorf`、`FailNow`、`Fatal`、`FatalIf` 方法，来说明测试不通过；调用 `Log`、`Logf` 方法来记录测试信息。函数列表和相关描述如下表所示：



函数	描述
t.Log, t.Logf	正常信息
t.Error, t.Errorf	测试失败信息
t.Fatal, t.Fatalf	致命错误，测试程序退出的信息
t.Fail	当前测试标记为失败
t.Failed	查看失败标记
t.FailNow	标记失败，并终止当前测试函数的执行。需要注意的是，我们只能在运行测试函数的Goroutine中调用t.FailNow方法，而不能在我们测试代码创建出的Goroutine中调用它
t.Skip, t.Skipf, t.Skipped	调用t.Skip方法，相当于先后对t.Log和t.SkipNow方法进行调用；调用t.Skipf方法，相当于先后对t.Logf 和t.SkipNow方法进行调用；方法t.Skipped的结果值会告知我们当前的测试是否已被忽略
t.Parallel	标记为可并行运算

下面的代码是两个简单的单元测试函数（函数位于文件 `math_test.go` 中）：

复制代码

```
1 func TestAbs(t *testing.T) {
2     got := Abs(-1)
3     if got != 1 {
4         t.Errorf("Abs(-1) = %f; want 1", got)
5     }
6 }
7
8 func TestMax(t *testing.T) {
9     got := Max(1, 2)
```

```
10     if got != 2 {  
11         t.Errorf("Max(1, 2) = %f; want 2", got)  
12     }  
13 }
```

执行 `go test` 命令来执行如上单元测试用例：

[复制代码](#)

```
1 $ go test  
2 PASS  
3 ok      github.com/marmotedu/gopractise-demo/31/test    0.002s
```

`go test` 命令自动搜集所有的测试文件，也就是格式为 `*_test.go` 的文件，从中提取全部测试函数并执行。

`go test` 还支持下面三个参数。

`-v`，显示所有测试函数的运行细节：

[复制代码](#)

```
1 $ go test -v  
2 === RUN   TestAbs  
3 --- PASS: TestAbs (0.00s)  
4 === RUN   TestMax  
5 --- PASS: TestMax (0.00s)  
6 PASS  
7 ok      github.com/marmotedu/gopractise-demo/31/test    0.002s
```

`-run < regexp >`，指定要执行的测试函数：

[复制代码](#)

```
1 $ go test -v -run='TestA.*'  
2 === RUN   TestAbs  
3 --- PASS: TestAbs (0.00s)  
4 PASS  
5 ok      github.com/marmotedu/gopractise-demo/31/test    0.001s
```

上面的例子中，我们只运行了以TestA开头的测试函数。

-count N，指定执行测试函数的次数：

[复制代码](#)

```
1 $ go test -v -run='TestA.*' -count=2
2 === RUN    TestAbs
3 --- PASS: TestAbs (0.00s)
4 === RUN    TestAbs
5 --- PASS: TestAbs (0.00s)
6 PASS
7 ok      github.com/marmotedu/gopractise-demo/31/test    0.002s
```


多个输入的测试用例

前面介绍的单元测试用例只有一个输入，但是很多时候，我们需要测试一个函数在多种不同输入下是否能正常返回。这时候，我们可以编写一个稍微复杂点的测试用例，用来支持多输入下的用例测试。例如，我们可以将TestAbs改造成如下函数：

[复制代码](#)

```
1 func TestAbs_2(t *testing.T) {
2     tests := []struct {
3         x      float64
4         want float64
5     }{
6         {-0.3, 0.3},
7         {-2, 2},
8         {-3.1, 3.1},
9         {5, 5},
10    }
11
12    for _, tt := range tests {
13        if got := Abs(tt.x); got != tt.want {
14            t.Errorf("Abs() = %f, want %v", got, tt.want)
15        }
16    }
17 }
```

上述测试用例函数中，我们定义了一个结构体数组，数组中的每一个元素代表一次测试用例。数组元素的值包含输入和预期的返回值：

 复制代码

```
1 tests := []struct {
2     x      float64
3     want   float64
4 }{
5     {-0.3, 0.3},
6     {-2, 2},
7     {-3.1, 3.1},
8     {5, 5},
9 }
```


上述测试用例，将被测函数放在 for 循环中执行：

 复制代码

```
1 for _, tt := range tests {
2     if got := Abs(tt.x); got != tt.want {
3         t.Errorf("Abs() = %f, want %v", got, tt.want)
4     }
5 }
```

上面的代码将输入传递给被测函数，并将被测函数的返回值跟预期的返回值进行比较。如果相等，则说明此次测试通过，如果不相等则说明此次测试不通过。通过这种方式，我们就可以在一个测试用例中，测试不同的输入和输出，也就是不同的测试用例。如果要新增一个测试用例，根据需要添加输入和预期的返回值就可以了，这些测试用例都共享其余的测试代码。

上面的测试用例中，我们通过 `got != tt.want` 来对比实际返回结果和预期返回结果。我们也可以使用 `github.com/stretchr/testify/assert` 包中提供的函数来做结果对比，例如：

 复制代码

```
1 func TestAbs_3(t *testing.T) {
2     tests := []struct {
3         x      float64
4         want   float64
5     }{
6         {-0.3, 0.3},
7         {-2, 2},
8         {-3.1, 3.1},
9         {5, 5},
10    }
```

```
11     for _, tt := range tests {
12         got := Abs(tt.x)
13         assert.Equal(t, got, tt.want)
14     }
15 }
16
```

使用assert来对比结果，有下面这些好处：

友好的输出结果，易于阅读。

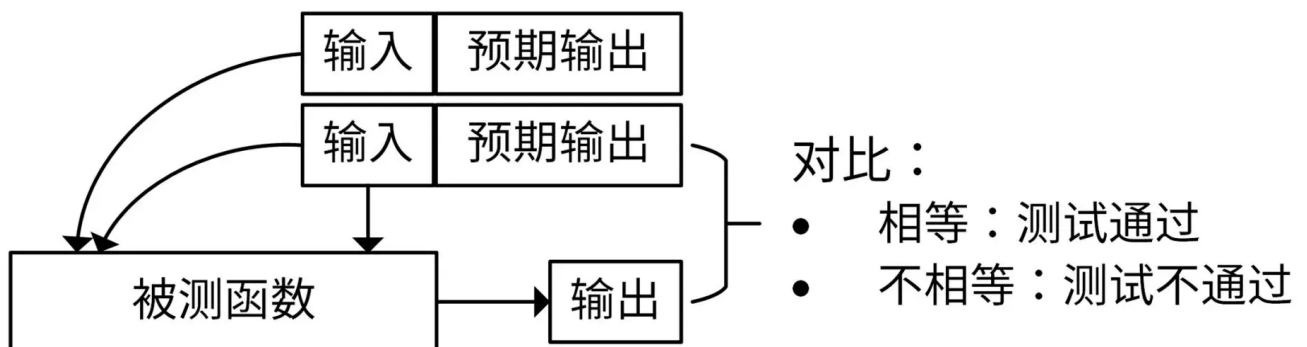
因为少了if got := Xxx(); got != tt.wang {}的判断，代码变得更加简洁。

可以针对每次断言，添加额外的消息说明，例如assert.Equal(t, got, tt.want, "Abs test")。

assert 包还提供了很多其他函数，供开发者进行结果对比，例如Zero、NotZero、Equal、NotEqual、Less、True、Nil、NotNil等。如果想了解更多函数，你可以参考[go doc github.com/stretchr/testify/assert](https://go.dev/doc/gotest/compare)。

自动生成单元测试用例

通过上面的学习，你也许可以发现，测试用例其实可以抽象成下面的模型：



用代码可表示为：

```
1 func TestXxx(t *testing.T) {
2     type args struct {
```

复制代码

```
3      // TODO: Add function input parameter definition.
4  }
5
6  type want struct {
7      // TODO: Add function return parameter definition.
8  }
9  tests := []struct {
10     name string
11     args args
12     want want
13 }{
14     // TODO: Add test cases.
15 }
16 for _, tt := range tests {
17     t.Run(tt.name, func(t *testing.T) {
18         if got := Xxx(tt.args); got != tt.want {
19             t.Errorf("Xxx() = %v, want %v", got, tt.want)
20         }
21     })
22 }
23 }
```

既然测试用例可以抽象成一些模型，那么我们就可以基于这些模型来自动生成测试代码。Go 社区中有一些优秀的工具可以自动生成测试代码，我推荐你使用 [gotests](#) 工具。

下面，我来讲讲 gotests 工具的使用方法，可以分成三个步骤。

第一步，安装 gotests 工具：

```
1 $ go get -u github.com/cweill/gotests/...
```

[复制代码](#)

gotests 命令执行格式为：gotests [options] [PATH] [FILE] ...。gotests 可以为 PATH 下的所有 Go 源码文件中的函数生成测试代码，也可以只为某个 FILE 中的函数生成测试代码。

第二步，进入测试代码目录，执行 gotests 生成测试用例：


```
1 $ gotests -all -w .
```

[复制代码](#)

上面的命令会为当前目录下所有 Go 源码文件中的函数生成测试代码。

第三步，添加测试用例：

生成完测试用例，你只需要添加需要测试的输入和预期的输出就可以了。下面的测试用例是通过 gotests 生成的：

 复制代码

```
1 func TestUnpointer(t *testing.T) {
2     type args struct {
3         offset *int64
4         limit  *int64
5     }
6     tests := []struct {
7         name string
8         args args
9         want *LimitAndOffset
10    }{
11        // TODO: Add test cases.
12    }
13    for _, tt := range tests {
14        t.Run(tt.name, func(t *testing.T) {
15            if got := Unpointer(tt.args.offset, tt.args.limit); !reflect.DeepEqual(
16                t.Errorf("Unpointer() = %v, want %v", got, tt.want)
17            )
18        })
19    }
20 }
```

我们只需要补全TODO位置的测试数据即可，补全后的测试用例见 [gorm_test.go](#) 文件。

性能测试

上面，我讲了用来测试代码的功能是否正常的单元测试，接下来我们来看下性能测试，它是用来测试代码的性能是否满足需求的。

性能测试的用例函数必须以Benchmark开头，例如BenchmarkXxx或Benchmark_Xxx（Xxx 部分为任意字母数字组合，首字母大写）。

函数参数必须是`*testing.B`，函数内以`b.N`作为循环次数，其中`N`会在运行时动态调整，直到性能测试函数可以持续足够长的时间，以便能够可靠地计时。下面的代码是一个简单的性能测试函数（函数位于文件 [math_test.go](#) 中）：

[复制代码](#)

```
1 func BenchmarkRandInt(b *testing.B) {
2     for i := 0; i < b.N; i++ {
3         RandInt()
4     }
5 }
```

`go test`命令默认不会执行性能测试函数，需要通过指定参数`-bench <pattern>`来运行性能测试函数。`-bench`后可以跟正则表达式，选择需要执行的性能测试函数，例如`go test -bench=".*"`表示执行所有的压力测试函数。执行`go test -bench=".*"`后输出如下：

[复制代码](#)

```
1 $ go test -bench=".*"
2 goos: linux
3 goarch: amd64
4 pkg: github.com/marmotedu/gopractise-demo/31/test
5 BenchmarkRandInt-4          97384827          12.4 ns/op
6 PASS
7 ok      github.com/marmotedu/gopractise-demo/31/test    1.223s
```

上面的结果只显示了性能测试函数的执行结果。BenchmarkRandInt性能测试函数的执行结果如下：

[复制代码](#)

```
1 BenchmarkRandInt-4          90848414          12.8 ns/op
```

每个函数的性能执行结果一共有 3 列，分别代表不同的意思，这里用上面的函数举例子：

BenchmarkRandInt-4，BenchmarkRandInt表示所测试的测试函数名，4 表示有 4 个 CPU 线程参与了此次测试，默认是GOMAXPROCS的值。

90848414，说明函数中的循环执行了90848414次。

12.8 ns/op，说明每次循环的执行平均耗时是 12.8 纳秒，该值越小，说明代码性能越高。

如果我们的性能测试函数在执行循环前，需要做一些耗时的准备工作，我们就需要重置性能测试时间计数，例如：

[复制代码](#)

```
1 func BenchmarkBigLen(b *testing.B) {
2     big := NewBig()
3     b.ResetTimer()
4     for i := 0; i < b.N; i++ {
5         big.Len()
6     }
7 }
```

当然，我们也可以先停止性能测试的时间计数，然后再开始时间计数，例如：

[复制代码](#)

```
1 func BenchmarkBigLen(b *testing.B) {
2     b.StopTimer() // 调用该函数停止压力测试的时间计数
3     big := NewBig()
4     b.StartTimer() // 重新开始时间
5     for i := 0; i < b.N; i++ {
6         big.Len()
7     }
8 }
```

B 类型的性能测试还支持下面 4 个参数。

benchmem，输出内存分配统计：

[复制代码](#)


```
1 $ go test -bench=".*" -benchmem
2 goos: linux
3 goarch: amd64
4 pkg: github.com/marmotedu/gopractise-demo/31/test
5 BenchmarkRandInt-4          96776823          12.8 ns/op      0 B/op
6 PASS
```



```
7 ok      github.com/marmotedu/gopractise-demo/31/test 1.255s
```

指定了`-benchmem`参数后，执行结果中又多了两列：`0 B/op`，表示每次执行分配了多少内存（字节），该值越小，说明代码内存占用越小；`0 allocs/op`，表示每次执行分配了多少次内存，该值越小，说明分配内存次数越少，意味着代码性能越高。


`benchtime`，指定测试时间和循环执行次数（格式需要为 `Nx`，例如 `100x`）：

 复制代码

```
1 $ go test -bench=".*" -benchtime=10s # 指定测试时间
2 goos: linux
3 goarch: amd64
4 pkg: github.com/marmotedu/gopractise-demo/31/test
5 BenchmarkRandInt-4      910328618      13.1 ns/op
6 PASS
7 ok      github.com/marmotedu/gopractise-demo/31/test 13.260s
8 $ go test -bench=".*" -benchtime=100x # 指定循环执行次数
9 goos: linux
10 goarch: amd64
11 pkg: github.com/marmotedu/gopractise-demo/31/test
12 BenchmarkRandInt-4      100      16.9 ns/op
13 PASS
14 ok      github.com/marmotedu/gopractise-demo/31/test 0.003s
```

`cpu`，指定 `GOMAXPROCS`。

`timeout`，指定测试函数执行的超时时间：

 复制代码

```
1 $ go test -bench=".*" -timeout=10s
2 goos: linux
3 goarch: amd64
4 pkg: github.com/marmotedu/gopractise-demo/31/test
5 BenchmarkRandInt-4      97375881      12.4 ns/op
6 PASS
7 ok      github.com/marmotedu/gopractise-demo/31/test 1.224s
```

总结

代码开发完成之后，我们需要为代码编写单元测试用例，并根据需要，给一些函数编写性能测试用例。Go 语言提供了 `testing` 包，供我们编写测试用例，并通过 `go test` 命令

来执行这些测试用例。

`go test` 在执行测试用例时，会查找具有固定格式的 Go 源码文件名，并执行其中具有固定格式的函数，这些函数就是测试用例。这就要求我们的测试文件名、函数名要符合 `go test` 工具的要求：Go 的测试文件名必须以 `_test.go` 结尾；测试用例函数必须以 `Test`、`Benchmark`、`Example` 开头。此外，我们在编写测试用例时，还要注意包和变量的命名规范。

Go 项目开发中，编写得最多的是单元测试用例。单元测试用例函数以 `Test` 开头，例如 `TestXxx` 或 `Test_xxx`（`Xxx` 部分为任意字母数字组合，首字母大写）。函数参数必须是 `*testing.T`，可以使用该类型来记录错误或测试状态。我们可以调用 `testing.T` 的 `Error`、`Errorf`、`FailNow`、`Fatal`、`FatalIf` 方法，来说明测试不通过；调用 `Log`、`Logf` 方法来记录测试信息。

下面是一个简单的单元测试函数：

[复制代码](#)

```
1 func TestAbs(t *testing.T) {
2     got := Abs(-1)
3     if got != 1 {
4         t.Errorf("Abs(-1) = %f; want 1", got)
5     }
6 }
```

编写完测试用例之后，可以使用 `go test` 命令行工具来执行这些测试用例。

此外，我们还可以使用 [gotests](#) 工具，来自动地生成单元测试用例，从而减少编写测试用例的工作量。

我们在 Go 项目开发中，还经常需要编写性能测试用例。性能测试用例函数必须以 `Benchmark` 开头，以 `*testing.B` 作为函数入参，通过 `go test -bench <pattern>` 运行。

课后练习

1. 编写一个 PrintHello 函数，该函数会返回 Hello World 字符串，并编写单元测试用例，对 PrintHello 函数进行测试。
2. 思考一下，哪些场景下采用白盒测试，哪些场景下采用黑盒测试？

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

👍 赞 0 💡 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 | 效率神器：如何设计和实现一个命令行客户端工具？

下一篇 37 | 代码测试（下）：Go 语言其他测试类型及 IAM 测试介绍

专栏上新

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

早鸟优惠 **¥99** 原价¥129



陈天
Tubi TV
研发副总裁

精选留言 (3)

💬 写留言



daz2yy

2021-08-17

老师，问下，测试的代码建议放在代码相同目录包下还是放在项目根目录下的 test 目录呢？

作者回复: 放在跟被测代码相同的目录下，便于维护



1

**Sch0ng**

2021-08-19

go自带测试框架testing。

使用gotests工具自动生成测试代码。

单元测试的价值是提高代码的可靠性，重构的时候多一层保障。

遇到单元测试不知道怎么写的情况，首先考虑函数的粒度是不是太粗，能不能拆成更小的函数。

展开 ∨

**lianyz**

2021-08-17

老师，什么时候使用ExampleXxx呢？

展开 ∨

作者回复: 有fmt.Println/fmt.Printf这类输出的时候

