



下载APP



## 37 | 代码测试（下）：Go 语言其他测试类型及 IAM 测试介绍

2021-08-19 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 23:35 大小 21.60M



你好，我是孔令飞。

🔗 **上一讲**，我介绍了 Go 中的两类测试：单元测试和性能测试。在 Go 中，还有一些其他的测试类型和测试方法，值得我们去了解和掌握。此外，IAM 项目也编写了大量测试用例，这些测试用例使用了不同的编写方法，你可以通过学习 IAM 的测试用例来验证你学到的测试知识。

今天，我就来介绍下 Go 语言中的其他测试类型：示例测试、TestMain 函数、Mock 测试、Fake 测试等，并且介绍下 IAM 项目是如何编写和运行测试用例的。



### 示例测试

示例测试以Example开头，没有输入和返回参数，通常保存在example\_test.go文件中。示例测试可能包含以Output:或者Unordered output:开头的注释，这些注释放在函数的结尾部分。Unordered output:开头的注释会忽略输出行的顺序。

执行go test命令时，会执行这些示例测试，并且go test会将示例测试输出到标准输出的内容，跟注释作对比（比较时将忽略行前后的空格）。如果相等，则示例测试通过测试；如果不相等，则示例测试不通过测试。下面是一个示例测试（位于example\_test.go文件中）：

[复制代码](#)

```
1 func ExampleMax() {  
2     fmt.Println(Max(1, 2))  
3     // Output:  
4     // 2  
5 }
```

执行go test命令，测试ExampleMax示例测试：

[复制代码](#)


```
1 $ go test -v -run='Example.*'  
2 === RUN    ExampleMax  
3 --- PASS: ExampleMax (0.00s)  
4 PASS  
5 ok      github.com/marmotedu/gopractise-demo/31/test    0.004s
```

可以看到ExampleMax测试通过。这里测试通过是因为fmt.Println(Max(1, 2))向标准输出输出了2，跟// Output:后面的2一致。

当示例测试不包含Output:或者Unordered output:注释时，执行go test只会编译这些函数，但不会执行这些函数。

## 示例测试命名规范

示例测试需要遵循一些命名规范，因为只有这样，Godoc 才能将示例测试和包级别的标识符进行关联。例如，有以下示例测试（位于example\_test.go文件中）：

 复制代码

```
1 package stringutil_test
2
3 import (
4     "fmt"
5
6     "github.com/golang/example/stringutil"
7 )
8
9 func ExampleReverse() {
10     fmt.Println(stringutil.Reverse("hello"))
11     // Output: olleh
12 }
```

Godoc 将在Reverse函数的文档旁边提供此示例，如下图所示：

### func Reverse

```
func Reverse(s string) string
```

Reverse returns its argument string reversed rune-wise left to right.

#### ▼ Example

Code:

```
fmt.Println(stringutil.Reverse("hello"))
```

Output:

```
olleh
```

示例测试名以Example开头，后面可以不跟任何字符串，也可以跟函数名、类型名或者类型\_方法名，中间用下划线\_连接，例如：

 复制代码

```
1 func Example() { ... } // 代表了整个包的示例
2 func ExampleF() { ... } // 函数F的示例
3 func ExampleT() { ... } // 类型T的示例
4 func ExampleT_M() { ... } // 方法T_M的示例
```

当某个函数 / 类型 / 方法有多个示例测试时，可以通过后缀来区分，后缀必须以小写字母开头，例如：

[复制代码](#)

```
1 func ExampleReverse()  
2 func ExampleReverse_second()  
3 func ExampleReverse_third()
```

## 大型示例

有时候，我们需要编写一个大型的示例测试，这时候我们可以编写一个整文件的示例（whole file example），它有这几个特点：文件名以`_test.go`结尾；只包含一个示例测试，文件中没有单元测试函数和性能测试函数；至少包含一个包级别的声明；当展示这类示例测试时，`godoc` 会直接展示整个文件。例如：

[复制代码](#)

```
1 package sort_test  
2  
3 import (  
4     "fmt"  
5     "sort"  
6 )  
7  
8 type Person struct {  
9     Name string  
10    Age  int  
11 }  
12  
13 func (p Person) String() string {  
14     return fmt.Sprintf("%s: %d", p.Name, p.Age)  
15 }  
16  
17 // ByAge implements sort.Interface for []Person based on  
18 // the Age field.  
19 type ByAge []Person  
20  
21 func (a ByAge) Len() int           { return len(a) }  
22 func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }  
23 func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }  
24  
25 func Example() {  
26     people := []Person{  
27         {"Bob", 31},  
28         {"John", 42},  
29         {"Michael", 17},
```

```
30         {"Jenny", 26},
31     }
32
33     fmt.Println(people)
34     sort.Sort(ByAge(people))
35     fmt.Println(people)
36
37     // Output:
38     // [Bob: 31 John: 42 Michael: 17 Jenny: 26]
39     // [Michael: 17 Jenny: 26 Bob: 31 John: 42]
40 }
```

一个包可以包含多个 whole file example，一个示例一个文件，例如 `example_interface_test.go`、`example_keys_test.go`、`example_search_test.go`等。

## TestMain 函数

有时候，我们在做测试的时候，可能会在测试之前做些准备工作，例如创建数据库连接等；在测试之后做些清理工作，例如关闭数据库连接、清理测试文件等。这时，我们可以在 `_test.go` 文件中添加 `TestMain` 函数，其入参为 `*testing.M`。

`TestMain` 是一个特殊的函数（相当于 `main` 函数），测试用例在执行时，会先执行 `TestMain` 函数，然后可以在 `TestMain` 中调用 `m.Run()` 函数执行普通的测试函数。在 `m.Run()` 函数前面我们可以编写准备逻辑，在 `m.Run()` 后面我们可以编写清理逻辑。

我们在示例测试文件 [math\\_test.go](#) 中添加如下 `TestMain` 函数：

```
1 func TestMain(m *testing.M) {
2     fmt.Println("do some setup")
3     m.Run()
4     fmt.Println("do some cleanup")
5 }
```

[复制代码](#)

执行 `go test`，输出如下：

[复制代码](#)

```
1 $ go test -v
2 do some setup
3
4 === RUN    TestAbs
5 --- PASS: TestAbs (0.00s)
6 ...
7 === RUN    ExampleMax
8 --- PASS: ExampleMax (0.00s)
9 PASS
10 do some cleanup
    ok      github.com/marmotodu/go-practice-demo/21/test 0.006s
```

在执行测试用例之前，打印了do some setup，在测试用例运行完成之后，打印了do some cleanup。

IAM 项目的测试用例中，使用 TestMain 函数在执行测试用例前连接了一个 fake 数据库，代码如下（位于 [internal/apiserver/service/v1/user\\_test.go](#) 文件中）：

[复制代码](#)

```
1 func TestMain(m *testing.M) {
2     fakeStore, _ := fake.NewFakeStore()
3     store.SetClient(fakeStore)
4     os.Exit(m.Run())
5 }
```

单元测试、性能测试、示例测试、TestMain 函数是 go test 支持的测试类型。此外，为了测试在函数内使用了 Go Interface 的函数，我们还延伸出了 Mock 测试和 Fake 测试两种测试类型。

## Mock 测试

一般来说，单元测试中是不允许有外部依赖的，那么也就是说，这些外部依赖都需要被模拟。在 Go 中，一般会借助各类 Mock 工具来模拟一些依赖。

GoMock 是由 Golang 官方开发维护的测试框架，实现了较为完整的基于 interface 的 Mock 功能，能够与 Golang 内置的 testing 包良好集成，也能用于其他的测试环境中。GoMock 测试框架包含了 GoMock 包和 mockgen 工具两部分，其中 GoMock 包用来完成对象生命周期的管理，mockgen 工具用来生成 interface 对应的 Mock 类源文件。下面，我来分别详细介绍下 GoMock 包和 mockgen 工具，以及它们的使用方法。

## 安装 GoMock

要使用 GoMock，首先需要安装 GoMock 包和 mockgen 工具，安装方法如下：

[复制代码](#)

```
1 $ go get github.com/golang/mock/gomock
2 $ go install github.com/golang/mock/mockgen
```

下面，我通过一个**获取当前 Golang 最新版本**的例子，来给你演示下如何使用 GoMock。示例代码目录结构如下（目录下的代码见 [gomock](#)）：

[复制代码](#)

```
1 tree .
2 .
3 |— go_version.go
4 |— main.go
5 |— spider
6   |— spider.go
```

spider.go文件中定义了一个Spider接口，spider.go代码如下：

[复制代码](#)

```
1 package spider
2
3 type Spider interface {
4     GetBody() string
5 }
```

Spider接口中的 GetBody 方法可以抓取<https://golang.org>首页的Build version字段，来获取 Golang 的最新版本。

我们在go\_version.go文件中，调用Spider接口的GetBody方法，go\_version.go代码如下：

[复制代码](#)

```
1 package gomock
2
```

```
3 import (
4     "github.com/marmotedu/gopractise-demo/gomock/spider"
5 )
6
7 func GetGoVersion(s spider.Spider) string {
8     body := s.GetBody()
9     return body
10 }
```

GetGoVersion函数直接返回表示版本的字符串。正常情况下，我们会写出如下的单元测试代码：

[复制代码](#)

```
1 func TestGetGoVersion(t *testing.T) {
2     v := GetGoVersion(spider.CreateGoVersionSpider())
3     if v != "go1.8.3" {
4         t.Error("Get wrong version %s", v)
5     }
6 }
```

上面的测试代码，依赖spider.CreateGoVersionSpider()返回一个实现了Spider接口的实例（爬虫）。但很多时候，spider.CreateGoVersionSpider()爬虫可能还没有实现，或者在单元测试环境下不能运行（比如，在单元测试环境中连接数据库），这时候TestGetGoVersion测试用例就无法执行。

那么，如何才能在这种情况下运行TestGetGoVersion测试用例呢？这时候，我们就可以通过 Mock 工具，Mock 一个爬虫实例。接下来我讲讲具体操作。

首先，用 GoMock 提供的 mockgen 工具，生成要 Mock 的接口的实现，我们在 gomock 目录下执行以下命令：

[复制代码](#)

```
1 $ mockgen -destination spider/mock/mock_spider.go -package spider github.com/m
```

上面的命令会在spider/mock目录下生成mock\_spider.go文件：

[复制代码](#)



```
1 $ tree .
2
3 .
4 ├── go_version.go
5 ├── go_version_test.go
6 ├── go_version_test_traditional_method.go~
7 └── spider
8     ├── mock
9     │   └── mock_spider.go
10    └── spider.go
```

mock\_spider.go文件中，定义了一些函数 / 方法，可以支持我们编写 TestGetGoVersion测试函数。这时候，我们的单元测试代码如下（见 [go\\_version\\_test.go](#)文件）：

[复制代码](#)

```
1 package gomock
2
3 import (
4     "testing"
5
6     "github.com/golang/mock/gomock"
7
8     spider "github.com/marmotedu/gopractise-demo/gomock/spider/mock"
9 )
10
11 func TestGetGoVersion(t *testing.T) {
12     ctrl := gomock.NewController(t)
13     defer ctrl.Finish()
14
15     mockSpider := spider.NewMockSpider(ctrl)
16     mockSpider.EXPECT().GetBody().Return("go1.8.3")
17     goVer := GetGoVersion(mockSpider)
18
19     if goVer != "go1.8.3" {
20         t.Errorf("Get wrong version %s", goVer)
21     }
22 }
```

这一版本的TestGetGoVersion通过 GoMock，Mock 了一个Spider接口，而不用去实现一个Spider接口。这就大大降低了单元测试用例编写的复杂度。通过 Mock，很多不能测试的函数也变得可测试了。

通过上面的测试用例，我们可以看到，GoMock 和 [🔗 上一讲](#)介绍的 testing 单元测试框架可以紧密地结合起来工作。

## mockgen 工具介绍

上面，我介绍了如何使用 GoMock 编写单元测试用例。其中，我们使用到了mockgen工具来生成 Mock 代码，mockgen工具提供了很多有用的功能，这里我来详细介绍下。

mockgen工具是 GoMock 提供的，用来 Mock 一个 Go 接口。它可以根据给定的接口，来自动生成 Mock 代码。这里，有两种模式可以生成 Mock 代码，分别是源码模式和反射模式。

### 1. 源码模式

如果有接口文件，则可以通过以下命令来生成 Mock 代码：

```
1 $ mockgen -destination spider/mock/mock_spider.go -package spider -source spid
```

[📄 复制代码](#)

上面的命令，Mock 了spider/spider.go文件中定义的Spider接口，并将 Mock 代码保存在spider/mock/mock\_spider.go文件中，文件的包名为spider。

mockgen 工具的参数说明见下表：

参数	说明
-source	指定需要模拟（Mock）的接口文件
-destination	指定Mock文件输出的地方，若不设置，则打印到标准输出中
-package	指定Mock文件的包名，若不设置，则为mock_前缀加上文件名（例如，这一讲中的包名会为 mock_spider）
-imports	依赖的包
-aux_files	接口文件不止一个文件时，附加文件
-build_flags	传递给build工具的参数

## 2. 反射模式

此外，mockgen 工具还支持通过使用反射程序来生成 Mock 代码。它通过传递两个非标志参数，即导入路径和逗号分隔的接口列表来启用，其他参数和源码模式共用，例如：

[复制代码](#)

```
1 $ mockgen -destination spider/mock/mock_spider.go -package spider github.com/m
```

## 通过注释使用 mockgen

如果有多个文件，并且分散在不同的位置，那么我们要生成 Mock 文件的时候，需要对每个文件执行多次 mockgen 命令（这里假设包名不相同）。这种操作还是比较繁琐的，mockgen 还提供了一种通过注释生成 Mock 文件的方式，此时需要借助go generate工具。

在接口文件的代码中，添加以下注释（具体代码见 [spider.go](#) 文件）：

[复制代码](#)

```
1 //go:generate mockgen -destination mock_spider.go -package spider github.com/c
```

这时候，我们只需要在gomock目录下，执行以下命令，就可以自动生成 Mock 代码：

```
1 $ go generate ./...
```

[复制代码](#)

## 使用 Mock 代码编写单元测试用例

生成了 Mock 代码之后，我们就可以使用它们了。这里我们结合testing来编写一个使用了 Mock 代码的单元测试用例。

**首先**，需要在单元测试代码里创建一个 Mock 控制器：

```
1 ctrl := gomock.NewController(t)
```

[复制代码](#)

将\*testing.T传递给 GoMock，生成一个Controller对象，该对象控制了整个 Mock 的过程。在操作完后，还需要进行回收，所以一般会在NewController后面 defer 一个 Finish，代码如下：

```
1 defer ctrl.Finish()
```

[复制代码](#)

**然后**，就可以调用 Mock 的对象了：


```
1 mockSpider := spider.NewMockSpider(ctrl)
```

[复制代码](#)

这里的spider是 mockgen 命令里面传递的包名，后面是NewMockXxxx格式的对象创建函数，Xxx是接口名。这里，我们需要传递控制器对象进去，返回一个 Mock 实例。

**接着**，有了 Mock 实例，我们就可以调用其断言方法EXPECT()了。


gomock 采用了链式调用法，通过.连接函数调用，可以像链条一样连接下去。例如：

 复制代码

```
1 mockSpider.EXPECT().GetBody().Return("go1.8.3")
```

Mock 一个接口的方法，我们需要 Mock 该方法的入参和返回值。我们可以通过参数匹配来 Mock 入参，通过 Mock 实例的 Return 方法来 Mock 返回值。下面，我们来分别看下如何指定入参和返回值。

先来看如何指定入参。如果函数有参数，我们可以使用参数匹配来指代函数的参数，例如：

 复制代码

```
1 mockSpider.EXPECT().GetBody(gomock.Any(), gomock.Eq("admin")).Return("go1.8.3")
```

gomock 支持以下参数匹配：

gomock.Any()，可以用来表示任意的入参。


gomock.Eq(value)，用来表示与 value 等价的值。

gomock.Not(value)，用来表示非 value 以外的值。

gomock.Nil()，用来表示 None 值。

接下来，我们看如何指定返回值。

EXPECT()得到 Mock 的实例，然后调用 Mock 实例的方法，该方法返回第一个Call对象，然后可以对其进行条件约束，比如使用 Mock 实例的 Return 方法约束其返回值。Call对象还提供了以下方法来约束 Mock 实例：

 复制代码

```
1 func (c *Call) After(preReq *Call) *Call // After声明调用在preReq完成后执行
2 func (c *Call) AnyTimes() *Call // 允许调用次数为 0 次或更多次
3 func (c *Call) Do(f interface{}) *Call // 声明在匹配时要运行的操作
4 func (c *Call) MaxTimes(n int) *Call // 设置最大的调用次数为 n 次
5 func (c *Call) MinTimes(n int) *Call // 设置最小的调用次数为 n 次
```

```
6 func (c *Call) Return(rets ...interface{}) *Call // 声明模拟函数调用返回的值
7 func (c *Call) SetArg(n int, value interface{}) *Call // 声明使用指针设置第 n 个参
8 func (c *Call) Times(n int) *Call // 设置调用次数为 n 次
```

上面列出了多个 Call 对象提供的约束方法，接下来我会介绍 3 个常用的约束方法：指定返回值、指定执行次数和指定执行顺序。

## 1. 指定返回值

我们可以提供调用 Call 的 Return 函数，来指定接口的返回值，例如：

```
1 mockSpider.EXPECT().GetBody().Return("go1.8.3")
```

[复制代码](#)

## 2. 指定执行次数

有时候，我们需要指定函数执行多少次，例如：对于接受网络请求的函数，计算其执行了多少次。我们可以通过 Call 的 Times 函数来指定执行次数：

```
1 mockSpider.EXPECT().Recv().Return(nil).Times(3)
```

[复制代码](#)

上述代码，执行了三次 Recv 函数，这里 gomock 还支持其他的执行次数限制：

AnyTimes()，表示执行 0 到多次。

MaxTimes(n int)，表示如果没有设置，最多执行 n 次。

MinTimes(n int)，表示如果没有设置，最少执行 n 次。

## 3. 指定执行顺序

有时候，我们还要指定执行顺序，比如要先执行 Init 操作，然后才能执行 Recv 操作：

[复制代码](#)

```
1 initCall := mockSpider.EXPECT().Init()
2 mockSpider.EXPECT().Recv().After(initCall)
```

最后，我们可以使用 `go test` 来测试使用了 Mock 代码的单元测试代码：

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

[复制代码](#)

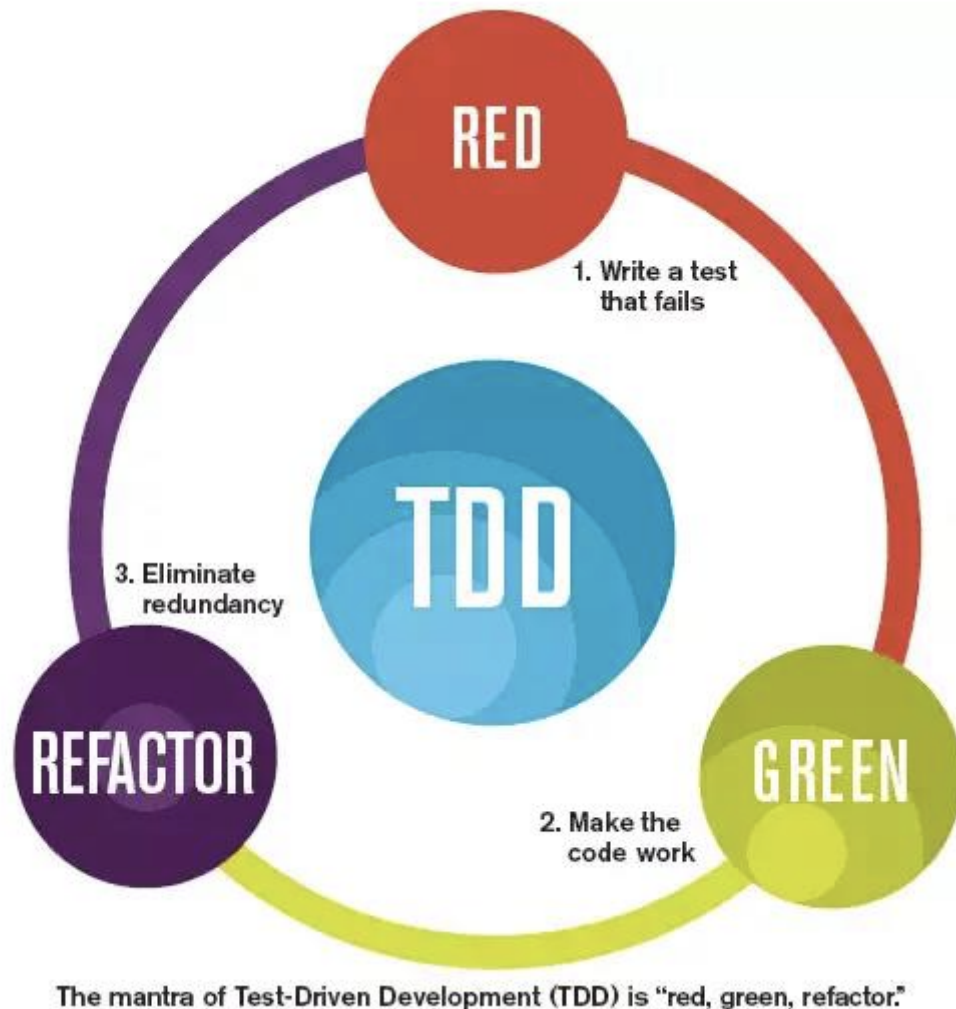
## Fake 测试

在 Go 项目开发中，对于比较复杂的接口，我们还可以 Fake 一个接口实现，来进行测试。所谓 Fake 测试，其实就是针对接口实现一个假（fake）的实例。至于如何实现 Fake 实例，需要你根据业务自行实现。例如：IAM 项目中 iam-apiserver 组件就实现了一个 fake store，代码见 [fake](#) 目录。因为这一讲后面的 IAM 项目测试实战部分有介绍，所以这里不再展开讲解。

## 何时编写和执行单元测试用例？

上面，我介绍了 Go 代码测试的基础知识，这里我再来分享下在做测试时一个比较重要的知识点：何时编写和执行单元测试用例。

## 编码前：TDD



Test-Driven Development，也就是测试驱动开发，是敏捷开发的一项核心实践和技术，也是一种设计方法论。简单来说，TDD 原理就是：开发功能代码之前，先编写测试用例代码，然后针对测试用例编写功能代码，使其能够通过。这样做的好处在于，通过测试的执行代码肯定满足需求，而且有助于面向接口编程，降低代码耦合，也极大降低了 bug 的出现几率。

然而，TDD 的坏处也显而易见：由于测试用例是在进行代码设计之前写的，很有可能限制开发者对代码的整体设计；并且，由于 TDD 对开发人员要求非常高，体现的思想跟传统开发思维也不一样，因此实施起来比较困难；此外，因为要先编写测试用例，TDD 也可能影响项目的研发进度。所以，在客观情况不满足的情况下，不应该盲目追求对业务代码使用 TDD 的开发模式。

## 与编码同步进行：增量

及时为增量代码写单测是一种良好的习惯。一方面是因为，此时我们对需求有一定的理解，能够更好地写出单元测试来验证正确性。并且，在单测阶段就发现问题，而不是等到联调测试中才发现，修复的成本也是最小的。



另一方面，在写单测的过程中，我们也能够反思业务代码的正确性、合理性，推动我们在实现的过程中更好地反思代码的设计，并及时调整。

## 编码后：存量

在完成业务需求后，我们可能会遇到这种情况：因为上线时间比较紧张、没有单测相关规划，开发阶段只手动测试了代码是否符合功能。

如果这部分存量代码出现较大的新需求，或者维护已经成为问题，需要大规模重构，这正是推动补全单测的好时机。为存量代码补充上单测，一方面能够推进重构者进一步理解原先的逻辑，另一方面也能够增强重构者重构代码后的信心，降低风险。

但是，补充存量单测可能需要再次回忆理解需求和逻辑设计等细节，而有时写单测的人并不是原编码的设计者，所以编码后编写和执行单元测试用例也有一定的不足。

## 测试覆盖率

我们写单元测试的时候应该想得很全面，能够覆盖到所有的测试用例，但有时也会漏过一些 case，Go 提供了 cover 工具来统计测试覆盖率。具体可以分为两大步骤。

第一步，生成测试覆盖率数据：

复制代码

```
1 $ go test -coverprofile=coverage.out
2 do some setup
3 PASS
4 coverage: 40.0% of statements
5 do some cleanup
6 ok      github.com/marmotedu/gopractise-demo/test 0.003s
```

上面的命令在当前目录下生成了 coverage.out 覆盖率数据文件。

```
github.com/marmotedu/iam/pkg/shutdown/shutdown.go:250: AddShutdownManager 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdown.go:263: AddShutdownCallback 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdown.go:275: SetErrorHandler 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdown.go:283: StartShutdown 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdown.go:303: ReportError 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal/posixsignal.go:32: NewPosixSignalManager 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal/posixsignal.go:45: GetName 0.0%
github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal/posixsignal.go:50: Start 100.0%
github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal/posixsignal.go:65: ShutdownStart 0.0%
github.com/marmotedu/iam/pkg/shutdown/shutdownmanagers/posixsignal/posixsignal.go:70: ShutdownFinish 0.0%
github.com/marmotedu/iam/pkg/util/genutil/genutil.go:15: OutDir 0.0%
total: 函数的单测覆盖率 80.0%
(statements) 项目的单测覆盖率 64.4%
```


## 第二步，分析覆盖率文件：

 复制代码

```
1 $ go tool cover -func=coverage.out
2 do some setup
3 PASS
4 coverage: 40.0% of statements
5 do some cleanup
6 ok      github.com/marmotedu/gopractise-demo/test 0.003s
7 [colin@dev test]$ go tool cover -func=coverage.out
8 github.com/marmotedu/gopractise-demo/test/math.go:9: Abs      100.0%
9 github.com/marmotedu/gopractise-demo/test/math.go:14: Max     100.0%
10 github.com/marmotedu/gopractise-demo/test/math.go:19: Min      0.0%
11 github.com/marmotedu/gopractise-demo/test/math.go:24: RandInt   0.0%
12 github.com/marmotedu/gopractise-demo/test/math.go:29: Floor     0.0%
13 total:                (statements) 40.0%
```

在上述命令的输出中，我们可以查看到哪些函数没有测试，哪些函数内部的分支没有测试完全。cover 工具会根据被执行代码的行数与总行数的比例计算出覆盖率。可以看到，Abs 和 Max 函数的测试覆盖率为 100%，Min 和 RandInt 的测试覆盖率为 0。

我们还可以使用 `go tool cover -html` 生成 HTML 格式的分析文件，可以更加清晰地展示代码的测试情况：

 复制代码

```
1 $ go tool cover -html=coverage.out -o coverage.html
```

上述命令会在当前目录下生成一个 `coverage.html` 文件，用浏览器打开 `coverage.html` 文件，可以更加清晰地看到代码的测试情况，如下图所示：

```
github.com/marmotedu/gopractise-demo/31/test/math.go (50.0%) ▼ not tracked not covered covered

package test

import (
    "math"
    "math/rand"
)

// Abs returns the absolute value of x.
func Abs(x float64) float64 {
    return math.Abs(x)
}

// Max returns the larger of x or y.
func Max(x, y float64) float64 {
    return math.Max(x, y)
}

// Min returns the smaller of x or y.
func Min(x, y float64) float64 {
    return math.Min(x, y)
}

// RandInt returns a non-negative pseudo-random int from the default Source.
func RandInt() int {
    return rand.Int()
}
```

通过上图，我们可以知道红色部分的代码没有被测试到，可以让我们接下来有针对性地添加测试用例，而不是一头雾水，不知道需要为哪些代码编写测试用例。

在 Go 项目开发中，我们往往会把测试覆盖率作为代码合并的一个强制要求，所以需要在进行代码测试时，同时生成代码覆盖率数据文件。在进行代码测试时，可以通过分析该文件，来判断我们的代码测试覆盖率是否满足要求，如果不满足则代码测试失败。


## IAM 项目测试实战

接下来，我来介绍下 IAM 项目是如何编写和运行测试用例的，你可以通过 IAM 项目的测试用例，加深对上面内容的理解。

### IAM 项目是如何运行测试用例的？

首先，我们来看下 IAM 项目是如何执行测试用例的。

在 IAM 项目的源码根目录下，可以通过运行 `make test` 执行测试用例，`make test` 会执行 `iam/scripts/make-rules/golang.mk` 文件中的 `go.test` 伪目标，规则如下：


 复制代码

```

1 .PHONY: go.test
2 go.test: tools.verify.go-junit-report
3   @echo "=====> Run unit test"
4   @set -o pipefail;$(GO) test -race -cover -coverprofile=$(OUTPUT_DIR)/coverag
5     -timeout=10m -short -v `go list ./...|\
6     egrep -v $(subst $(SPACE),'|',$(sort $(EXCLUDE_TESTS)))` 2>&1 | \
7     tee >(go-junit-report --set-exit-code >$(OUTPUT_DIR)/report.xml)
8   @sed -i '/mock_*.go/d' $(OUTPUT_DIR)/coverage.out # remove mock_*.go files
9   @$ (GO) tool cover -html=$(OUTPUT_DIR)/coverage.out -o $(OUTPUT_DIR)/coverage

```

在上述规则中，我们执行go test时设置了超时时间、竞态检查，开启了代码覆盖率检查，覆盖率测试数据保存在了coverage.out文件中。在 Go 项目开发中，并不是所有的包都需要单元测试，所以上面的命令还过滤掉了一些不需要测试的包，这些包配置在EXCLUDE\_TESTS变量中：

 复制代码


```

1 EXCLUDE_TESTS=github.com/marmotedu/iam/test github.com/marmotedu/iam/pkg/log g

```

同时，也调用了go-junit-report将 go test 的结果转化成了 xml 格式的报告文件，该报告文件会被一些 CI 系统，例如 Jenkins 拿来解析并展示结果。上述代码也同时生成了coverage.html 文件，该文件可以存放在制品库中，供我们后期分析查看。

这里需要注意，Mock 的代码是不需要编写测试用例的，为了避免影响项目的单元测试覆盖率，需要将 Mock 代码的单元测试覆盖率数据从coverage.out文件中删除掉，go.test规则通过以下命令删除这些无用的数据：


 复制代码

```

1 sed -i '/mock_*.go/d' $(OUTPUT_DIR)/coverage.out # remove mock_*.go files fr

```

另外，还可以通过make cover来进行单元测试覆盖率测试，make cover会执行iam/scripts/make-rules/golang.mk文件中的go.test.cover伪目标，规则如下：

 复制代码

```
1 .PHONY: go.test.cover
2
3 go.test.cover: go.test
4     @$(GO) tool cover -func=$(OUTPUT_DIR)/coverage.out | \
    awk -v target=$(COVERAGE) -f $(ROOT_DIR)/scripts/coverage.awk
```

上述目标依赖`go.test`，也就是说执行单元测试覆盖率目标之前，会先进行单元测试，然后使用单元测试产生的覆盖率数据`coverage.out`计算出总的单元测试覆盖率，这里是通过 [coverage.awk](#) 脚本来计算的。

如果单元测试覆盖率不达标，`Makefile` 会报错并退出。可以通过 `Makefile` 的 [COVERAGE](#) 变量来设置单元测试覆盖率阈值。

`COVERAGE` 的默认值为 60，我们也可以在命令行手动指定，例如：

```
1 $ make cover COVERAGE=80
```

[复制代码](#)

为了确保项目的单元测试覆盖率达标，需要设置单元测试覆盖率质量红线。一般来说，这些红线很难靠开发者的自觉性去保障，所以好的方法是将质量红线加入到 `CICD` 流程中。

所以，在`Makefile`文件中，我将`cover`放在`all`目标的依赖中，并且位于 `build` 之前，也就是`all: gen add-copyright format lint cover build`。这样每次当我们执行 `make` 时，会自动进行代码测试，并计算单元测试覆盖率，如果覆盖率不达标，则停止构建；如果达标，继续进入下一步的构建流程。


## IAM 项目测试案例分享

接下来，我会给你展示一些 IAM 项目的测试案例，因为这些测试案例的实现方法，我在 [36 讲](#) 和这一讲的前半部分已有详细介绍，所以这里，我只列出具体的实现代码，不会再介绍这些代码的实现方法。

### 1. 单元测试案例

我们可以手动编写单元测试代码，也可以使用 `gotests` 工具生成单元测试代码。

先来看手动编写测试代码的案例。这里单元测试代码见 [Test\\_Option](#)，代码如下：

 复制代码


```
1 func Test_Option(t *testing.T) {
2     fs := pflag.NewFlagSet("test", pflag.ExitOnError)
3     opt := log.NewOptions()
4     opt.AddFlags(fs)
5
6     args := []string{"--log.level=debug"}
7     err := fs.Parse(args)
8     assert.Nil(t, err)
9
10    assert.Equal(t, "debug", opt.Level)
11 }
```

上述代码中，使用了 [github.com/stretchr/testify/assert](https://github.com/stretchr/testify/assert) 包来对比结果。

再来看使用 `gotests` 工具生成单元测试代码的案例（Table-Driven 的测试模式）。出于效率上的考虑，IAM 项目的单元测试用例，基本都是使用 `gotests` 工具生成测试用例模板代码，并基于这些模板代码填充测试 Case 的。代码见 [service\\_test.go](#) 文件。

## 2. 性能测试案例

IAM 项目的性能测试用例，见 [BenchmarkListUser](#) 测试函数。代码如下：

 复制代码

```
1 func BenchmarkListUser(b *testing.B) {
2     opts := metav1.ListOptions{
3         Offset: pointer.ToInt64(0),
4         Limit:   pointer.ToInt64(50),
5     }
6     storeIns, _ := fake.GetFakeFactoryOr()
7     u := &userService{
8         store: storeIns,
9     }
10
11    for i := 0; i < b.N; i++ {
12        _, _ = u.List(context.TODO(), opts)
13    }
14 }
```

### 3. 示例测试案例

IAM 项目的示例测试用例见 [example\\_test.go](#) 文件。example\_test.go 中的一个示例测试代码如下：

```
1 func ExampleNew() {
2     err := New("whoops")
3     fmt.Println(err)
4
5     // Output: whoops
6 }
```

[复制代码](#)

### 4. TestMain 测试案例

IAM 项目的 TestMain 测试案例，见 [user\\_test.go](#) 文件中的 TestMain 函数：

```
1 func TestMain(m *testing.M) {
2     _, _ = fake.GetFakeFactoryOr()
3     os.Exit(m.Run())
4 }
```

[复制代码](#)

TestMain 函数初始化了 fake Factory，然后调用 m.Run 执行测试用例。

### 5. Mock 测试案例

Mock 代码见 [internal/apiserver/service/v1/mock\\_service.go](#)，使用 Mock 的测试用例见 [internal/apiserver/controller/v1/user/create\\_test.go](#) 文件。因为代码比较多，这里建议你打开链接，查看测试用例的具体实现。

我们可以在 IAM 项目的根目录下执行以下命令，来自动生成所有的 Mock 文件：

```
1 $ go generate ./...
```

[复制代码](#)

## 6. Fake 测试案例

fake store 代码实现位于 [internal/apiserver/store/fake](#) 目录下。fake store 的使用方式，见 [user\\_test.go](#) 文件：

[复制代码](#)

```
1 func TestMain(m *testing.M) {
2     _, _ = fake.GetFakeFactoryOr()
3     os.Exit(m.Run())
4 }
5
6 func BenchmarkListUser(b *testing.B) {
7     opts := metav1.ListOptions{
8         Offset: pointer.ToInt64(0),
9         Limit:  pointer.ToInt64(50),
10    }
11    storeIns, _ := fake.GetFakeFactoryOr()
12    u := &userService{
13        store: storeIns,
14    }
15
16    for i := 0; i < b.N; i++ {
17        _, _ = u.List(context.TODO(), opts)
18    }
19 }
```

上述代码通过TestMain初始化 fake 实例（[store.Factory](#)接口类型）：

[复制代码](#)

```
1 func GetFakeFactoryOr() (store.Factory, error) {
2     once.Do(func() {
3         fakeFactory = &datastore{
4             users:    FakeUsers(ResourceCount),
5             secrets:  FakeSecrets(ResourceCount),
6             policies: FakePolicies(ResourceCount),
7         }
8     })
9
10    if fakeFactory == nil {
11        return nil, fmt.Errorf("failed to get mysql store factory, mysqlFactory")
12    }
13
14    return fakeFactory, nil
15 }
```



GetFakeFactoryOr函数，创建了一些 fake users、secrets、policies，并保存在了 fakeFactory变量中，供后面的测试用例使用，例如 BenchmarkListUser、Test\_newUsers 等。

## 其他测试工具 / 包

最后，我再来分享下 Go 项目测试中常用的工具 / 包，因为内容较多，我就不详细介绍了，如果感兴趣你可以点进链接自行学习。我将这些测试工具 / 包分为了两类，分别是测试框架和 Mock 工具。

### 测试框架

🔗 **Testify 框架**：Testify 是 Go test 的预判工具，它能让你的测试代码变得更优雅和高效，测试结果也变得更详细。

🔗 **GoConvey 框架**：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。

### Mock 工具

这一讲里，我介绍了 Go 官方提供的 Mock 框架 GoMock，不过还有一些其他的优秀 Mock 工具可供我们使用。这些 Mock 工具分别用在不同的 Mock 场景中，我在 🔗10 讲中已经介绍过。不过，为了使我们这一讲的测试知识体系更加完整，这里我还是再提一次，你可以复习一遍。

🔗 **sqlmock**：可以用来模拟数据库连接。数据库是项目中比较常见的依赖，在遇到数据库依赖时都可以用它。

🔗 **httpmock**：可以用来 Mock HTTP 请求。

🔗 **bouk/monkey**：猴子补丁，能够通过替换函数指针的方式来修改任意函数的实现。如果 golang/mock、sqlmock 和 httpmock 这几种方法都不能满足我们的需求，我们可以尝试用猴子补丁的方式来 Mock 依赖。可以这么说，猴子补丁提供了单元测试 Mock 依赖的最终解决方案。

## 总结

这一讲，我介绍了除单元测试和性能测试之外的另一些测试方法。

除了示例测试和 `TestMain` 函数，我还详细介绍了 Mock 测试，也就是如何使用 GoMock 来测试一些在单元测试环境下不好实现的接口。绝大部分情况下，可以使用 GoMock 来 Mock 接口，但是对于一些业务逻辑比较复杂的接口，我们可以通过 Fake 一个接口实现，来对代码进行测试，这也称为 Fake 测试。

此外，我还介绍了何时编写和执行测试用例。我们可以根据需要，选择在编写代码前、编写代码中、编写代码后编写测试用例。

为了保证单元测试覆盖率，我们还应该为整个项目设置单元测试覆盖率质量红线，并将该质量红线加入到 CICD 流程中。我们可以通过 `go test -coverprofile=coverage.out` 命令来生成测试覆盖率数据，通过 `go tool cover -func=coverage.out` 命令来分析覆盖率文件。

IAM 项目中使用了大量的测试方法和技巧来测试代码，为了加深你对测试知识的理解，我也列举了一些测试案例，供你参考、学习和验证。具体的测试案例，你可以返回前面查看下。

除此之外，我们还可以使用其他一些测试框架，例如 Testify 框架和 GoConvey 框架。在 Go 代码测试中，我们最常使用的是 Go 官方提供的 Mock 框架 GoMock，但仍然有其他优秀的 Mock 工具，可供我们在不同场景下使用，例如 `sqlmock`、`httpmock`、`bouk/monkey` 等。

## 课后习题

1. 请使用 [sqlmock](#) 来 Mock 一个 GORM 数据库实例，并完成 GORM 的 CURD 单元测试用例编写。
2. 思考下，在 Go 项目开发中，还有哪些优秀的测试框架、测试工具、Mock 工具以及测试技巧？欢迎你在留言区分享。

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**

© 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 36 | 代码测试（上）：如何编写 Go 语言单元测试和性能测试用例？

下一篇 特别放送 | 给你一份清晰、可直接套用的Go编码规范

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  
TubiTV 研发副总裁

### 精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。