Go语言 标准库 testing包（单元测试）

golang提供了标准库`testing`用来支持测试。

Golang支持的测试种类：

类型	格式	作用
单元测试	函数名前缀为Test	测试程序的一些逻辑行为是否正确
基准（压力）测试	函数名前缀为Benchmark	测试函数的性能
示例测试	函数名前缀为Example	为文档提供示例文档
模糊（随机）测试	函数名前缀为Fuzz	生成一个随机测试用例去覆盖认为测不到的各种赋值场景

单元测试

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，那么一个新建`calc_test.go`作为测试文件。

> 注意：创建项目后需要`go mod init`初始化项目

示例：

- calc.go

```
package main

func Add(a int, b int) int {
    return a + b
}

func Mul(a int, b int) int {
    return a * b
}
```

- calc_test.go

```
package main

import "testing"

func TestADD(t *testing.T) {
    if ans := Add(1, 2); ans != 3 {
        t.Errorf("1 + 2 expected be 3, but %d got", ans)
    }

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

> - 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`类型
> - TestMain的参数是`*testing.M`类型

`run test`结果为：

```
Running tool: C:\Go\bin\go.exe test -timeout 30s -run ^TestADD$ go_pro

ok      go_pro  0.272s
```

还可以在终端运行`go test`，则该包下的**所有测试用例都会被执行**，PASS表示测试用例运行成功，FAIL表示测试用例运行失败。

```
PS E:\golang开发学习\go_pro> go test
PASS
ok      go_pro  0.244s
```

或使用`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。

```
PS E:\golang开发学习\go_pro> go test -v
=== RUN   TestADD
--- PASS: TestADD (0.00s)
PASS
ok      go_pro  0.245s
```

```
PS E:\golang开发学习\go_pro> go test -cover
PASS
coverage: 50.0% of statements
ok      go_pro  0.417s
```

如果只想运行其中一个用例，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，如`^`、`$`。

```
PS E:\golang开发学习\go_pro> go test -run TestADD -v
=== RUN   TestADD
--- PASS: TestADD (0.00s)
PASS
ok      go_pro  0.180s
```

单元测试框架提供的日志方法

方法	作用
Log	打印日志，同时结束测试
Logf	格式化打印日志，同时结束测试
Error	打印错误日志，同时结束测试
Errorf	格式化打印错误日志，同时结束测试
Fatal	打印致命日志，同时结束测试
Fatalf	格式化打印致命日志，同时结束测试

子测试(Subtests)

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

calc_test.go

```
func TestMul(t *testing.T) {
    t.Run("pos", func(t *testing.T) { // pos参数为子测试命名
        if Mul(2, 3) != 6 {
            t.Fatal("fail")
        }
    })

    t.Run("neg", func(t *testing.T) {
        if Mul(2, -3) != -6 {
            t.Fatal("fail")
        }
    })
}
```

> `t.Error/t.Errorf`与`t.Fatal/t.Fatalf`的区别在于:
>
> - `t.Error/t.Errorf`遇错不停，还会继续执行其他的测试用例
> - `t.Fatal/t.Fatalf`遇错即停，不会继续执行其他的测试用例

运行结果:

```
Running tool: C:\Go\bin\go.exe test -timeout 30s -run ^TestMul$ go_pro

ok      go_pro  0.288s
```

运行某个测试用例的子测试：

```
PS E:\golang开发学习\go_pro> go test -run TestMul/pos -v
=== RUN   TestMul
=== RUN   TestMul/pos
--- PASS: TestMul (0.00s)
    --- PASS: TestMul/pos (0.00s)
PASS
ok      go_pro  0.256s
```

对于多个子测试的场景，更推荐使用`table-driven tests`的写法，如下：

```
func TestMul(t *testing.T) {
    cases := []struct {
        Name           string
        A, B, Expected int
    }{
        {"pos", 2, 3, 6},
        {"neg", 2, -3, -6},
        {"zero", 2, 0, 0},
    }

    for _, c := range cases {
        t.Run(c.Name, func(t *testing.T) {
            if ans := Mul(c.A, c.B); ans != c.Expected {
                t.Fatalf("%d * %d expected %d, but %d got", c.A, c.B,
c.Expected, ans)
            }
        })
    }
}
```

所有用例的数据组织在切片`cases`中，借助循环创建子测试，这样写的好处有：

- 新增用例方便
- 测试代码可读性好
- 用例失败时，报错信息格式较统一，测试报告易读

> 若数据很大，或者一些二进制数据，推荐使用相对路径从文件中读取。

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加清晰

例如，将创建子测试的逻辑抽取出来：

```
type calcCase struct {
    A, B, Expected int
}

func createMulTestCase(t *testing.T, c *calcCase) {
    // t.Helper()
    if ans := Mul(c.A, c.B); ans != c.Expected {
```

```
        t.Fatalf("%d * %d expected %d, but %d got", c.A, c.B, c.Expected, ans)
    }
}

func TestMul(t *testing.T) {
    createMulTestCase(t, &calcCase{2, 3, 6})
    createMulTestCase(t, &calcCase{2, -3, -6})
    createMulTestCase(t, &calcCase{2, 0, 1}) // 错误case
}
```

这里故意创建了一个错误的测试用例，运行之后会报告发生错误的文件信息：

```
Running tool: C:\Go\bin\go.exe test -timeout 30s -run ^TestMul$ go_pro

--- FAIL: TestMul (0.00s)
    e:\golang开发学习\go_pro\factory_test.go:22: 2 * 0 expected 1, but 0 got
FAIL
FAIL    go_pro  0.259s
FAIL
```

可以看到，错误信息提示在帮助函数createMulTestCase内部，我们在TestMul函数内调用了三次createMulTestCase，则第一时间并不能确定哪一行发生了错误。有些帮助函数还可能在不同的函数中被调用，因此会导致报错信息都在同一处，不能方便准确定位。因此引用了t.Helper()用于标注该函数是帮助函数，报错时则将输出帮助函数的调用者信息。

调用`t.Helper()`：

```
func createMulTestCase(t *testing.T, c *calcCase) {
    t.Helper()
    if ans := Mul(c.A, c.B); ans != c.Expected {
        t.Fatalf("%d * %d expected %d, but %d got", c.A, c.B, c.Expected, ans)
    }
}
```

运行结果：

```
Running tool: C:\Go\bin\go.exe test -timeout 30s -run ^TestMul$ go_pro

--- FAIL: TestMul (0.00s)
    e:\golang开发学习\go_pro\factory_test.go:29: 2 * 0 expected 1, but 0 got
FAIL
FAIL    go_pro  0.208s
FAIL
```

> 使用帮助函数注意：
>
> - 不要返回错误，帮助函数内部直接使用`t.Error`或`t.Fatal`即可，在用例的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()`能让错误信息更好定位

setup和teardown

如果在同一个测试文件中，每一个测试用例的运行前后逻辑是相同的，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，`testing`标准库提供了这样的机制：

```
func setup() {
    fmt.Println("before all tests")
}

func teardown() {
    fmt.Println("after all tests")
}

func Test1(t *testing.T) {
    fmt.Println("test1")
}

func Test2(t *testing.T) {
    fmt.Println("test2")
}

func TestMain(m *testing.M) {

    setup()
    code := m.Run()
    teardown()

    os.Exit(code)
}
```

> - 这个测试文件中，包含2个测试用例Test1和Test2
> - 如果测试文件包含函数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(code)处理返回状态码，若部位0，说明用例失败
> - 因此可以再调用m.Run()前后做一些额外的准备(setup)和回收(teardown)工作

执行`go test`输出：

```
$ go test
before all tests
test1
test2
PASS
after all tests
ok      example 0.003s
```

网络测试(Network)

TCP/HTTP

假设要测试一个API接口的handler能够正常工作，例如helloHandler。

```
package main

import "net/http"

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

则可以创建真是的网络链接来进行测试：

```
package main

import (
    "io/ioutil"
    "net"
    "net/http"
    "testing"
)

func handlerError(t *testing.T, err error) {
    t.Helper()
    if err != nil {
        t.Fatal("failed", err)
    }
}

func TestConn(t *testing.T) {
    l, err := net.Listen("tcp", "127.0.0.1:0")
    handlerError(t, err)
    defer l.Close()

    http.HandleFunc("/hello", helloHandler)
    go http.Serve(l, nil)

    resp, err := http.Get("http://" + l.Addr().String() + "/hello")
    handlerError(t, err)

    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    handlerError(t, err)

    if string(body) != "hello world" {
        t.Fatal("expected hello world but got", string(body))
    }
}
```

> - 调用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(l, nil)启动http服务
> - 使用http.Get发送一个Get请求，检查返回值是否正确
> - 尽量不对http和net使用mock，这样可以覆盖较为真实的场景

httptest

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

可将上述测试用例修改为：

```
package main

import (
    "io/ioutil"
    "net/http/httptest"
    "testing"
)

func TestConn(t *testing.T) {
    req := httptest.NewRequest("GET", "http://example.com/foo", nil)
    w := httptest.NewRecorder()
    helloHandler(w, req)
    bytes, _ := ioutil.ReadAll(w.Result().Body)

    if string(bytes) != "hello world" {
        t.Fatal("expected hello world but got", string(bytes))
    }
}
```

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