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》

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

```
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

```
// add.go
package add

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

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

```
// add_test.go
package add

import (
  "testing"
)
func TestAdd(t *testing.T) {
  sum := Add(1, 2)
  if sum == 3 {
    t.Log("the result is ok")
  } else {
    t.Fatal("the result is wrong")
  }
}
```

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。

复制代码

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

如果测试结果不符合预期，输出如下。

复制代码

```
=== RUN   TestAdd
    add_test.go:13: the result is wrong
--- FAIL: TestAdd (0.00s)

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。我们这个例子中，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 `sqldb` 做单元测试，测试一下创建表的功能是否正常。

复制代码

```
func TestSqldb_CreateTable(t *testing.T) {
  sqldb, err := New(
    WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
  )
  assert.Nil(t, err)
  assert.NotNil(t, sqldb)
  // 测试对于无效的配置返回错误
  name := "test_create_table"
  var notValidTable = TableData{
```

```
    TableName: name,
    ColumnNames: []Field{
      {Title: "书名", Type: "notValid"},
      {Title: "URL", Type: "VARCHAR(255)"},
    },
    AutoKey: true,
  }
 // 延迟删除表
  defer func() {
    err := sqldb.DropTable(notValidTable)
    assert.Nil(t, err)
  }()
  // 测试对于有效的配置返回错误
  err = sqldb.CreateTable(notValidTable)
  assert.NotNil(t, err)
  // 测试对于无效的配置返回错误
  var validTable = TableData{
    TableName: name,
    ColumnNames: []Field{
      {Title: "书名", Type: "MEDIUMTEXT"},
      {Title: "URL", Type: "VARCHAR(255)"},
    },
    AutoKey: true,
  }
  err = sqldb.CreateTable(validTable)
  assert.Nil(t, err)
}
```


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。

复制代码

```go
// split.go
package split

import "strings"
func Split(s, sep string) []string {
  var result []string
  i := strings.Index(s, sep)
  for i > -1 {
    result = append(result, s[:i])
    s = s[i+len(sep):]
    i = strings.Index(s, sep)
  }
  return append(result, s)
}
```


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。

复制代码

```go
package split

import (
  "reflect"
  "testing"
)

//单元测试
func TestSplit(t *testing.T) {
  got := Split("a/b/c", "/")
  want := []string{"a", "b", "c"}
  if !reflect.DeepEqual(want, got) {
    t.Fatalf("expected: %v, got: %v", want, got)
  }
}
```

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。

```go
// split_test.go
package split

import (
  "reflect"
  "testing"
)

func TestSplit(t *testing.T) {
  tests := map[string]struct {
    input string
    sep   string
    want  []string
  }{
    "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
    "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
    "no sep":       {input: "abc", sep: "/",   want: []string{"abc"}},
    "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
  }

  for name, tc := range tests {
    got := Split(tc.input, tc.sep)
    if !reflect.DeepEqual(tc.want, got) {
      t.Fatalf("%s: expected: %v, got: %v", name, tc.want, got)
    }
  }
}
```


我们也可以把之前测试 CreateTable 的函数修改为表格驱动测试。

```go
func TestSqldb_CreateTableDriver(t *testing.T) {
  type args struct {
    t TableData
  }
  name := "test_create_table"

  tests := []struct {
    name    string
    args    args
    wantErr bool
  }{
    {
      name: "create_not_valid_table",
      args: args{TableData{
        TableName: name,
        ColumnNames: []Field{
          {Title: "书名", Type: "not_valid"},
          {Title: "URL", Type: "VARCHAR(255)"},
```

```
        },
      }},
      wantErr: true,
    },
    {
      name: "create_valid_table",
      args: args{TableData{
        TableName: name,
        ColumnNames: []Field{
          {Title: "书名", Type: "MEDIUMTEXT"},
          {Title: "URL", Type: "VARCHAR(255)"},
        },
      }},
      wantErr: false,
    },
    {
      name: "create_valid_table_with_primary_key",
      args: args{TableData{
        TableName: name,
        ColumnNames: []Field{
          {Title: "书名", Type: "MEDIUMTEXT"},
          {Title: "URL", Type: "VARCHAR(255)"},
        },
        AutoKey: true,
      }},
      wantErr: false,
    },
  }

  sqldb, err := New(
    WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
  )

  for _, tt := range tests {
    err = sqldb.CreateTable(tt.args.t)
    if tt.wantErr {
      assert.NotNil(t, err, tt.name)
    } else {
      assert.Nil(t, err, tt.name)
    }
    sqldb.DropTable(tt.args.t)
  }
}
```


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。

```
=== RUN   TestSqldb_CreateTableDriver
    sqldb_test.go:98:
          Error Trace:  /Users/jackson/career/crawler/sqldb/sqldb_test.go:98
          Error:        Expected nil, but got: &mysql.MySQLError{Number:0x428
          Test:         TestSqldb_CreateTableDriver
          Messages:     create_not_valid_table
--- FAIL: TestSqldb_CreateTableDriver (0.06s)

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。


```go
func TestSplit(t *testing.T) {
  tests := map[string]struct {
    input string
    sep   string
    want  []string
  }{
    "simple":       {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
    "wrong sep":    {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
    "no sep":       {input: "abc", sep: "/", want: []string{"abc"}},
    "trailing sep": {input: "a/b/c/", sep: "/", want: []string{"a", "b", "c"}},
  }

  for name, tc := range tests {
    t.Run(name, func(t *testing.T) {
      got := Split(tc.input, tc.sep)
      if !reflect.DeepEqual(tc.want, got) {
        t.Fatalf("expected: %#v, got: %#v", tc.want, got)
      }
    })
  }
```

```
}
```

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

```
func TestSqldb_InsertTable(t *testing.T) {
  type args struct {
    t TableData
  }
  tableName := "test_create_table"
  columnNames := []Field{{Title: "书名", Type: "MEDIUMTEXT"}, {Title: "price", T
  tests := []struct {
    name    string
    args    args
    wantErr bool
  }{
    {
      name: "insert_data",
      args: args{TableData{
        TableName:   tableName,
        ColumnNames: columnNames,
        Args:        []interface{}{"book1", 2},
        DataCount:   1,
      }},
      wantErr: false,
    },
    {
      name: "insert_multi_data",
      args: args{TableData{
        TableName:   tableName,
        ColumnNames: columnNames,
        Args:        []interface{}{"book3", 88.88, "book4", 99.99},
        DataCount:   2,
      }},
      wantErr: false,
    },
    {
      name: "insert_multi_data_wrong_count",
      args: args{TableData{
        TableName:   tableName,
        ColumnNames: columnNames,
        Args:        []interface{}{"book3", 88.88, "book4", 99.99},
        DataCount:   1,
      }},
      wantErr: true,
    },
    {
      name: "insert_wrong_data_type",
```

```
      args: args{TableData{
        TableName:   tableName,
        ColumnNames: columnNames,
        Args:        []interface{}{"book2", "rrr"},
        DataCount:   1,
      }},
      wantErr: true,
    },
  }

  sqldb, err := New(
    WithConnURL("root:123456@tcp(127.0.0.1:3326)/crawler?charset=utf8"),
  )
  err = sqldb.CreateTable(tests[0].args.t)
  defer sqldb.DropTable(tests[0].args.t)
  assert.Nil(t, err)
  for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
      err = sqldb.Insert(tt.args.t)
      if tt.wantErr {
        assert.NotNil(t, err, tt.name)
      } else {
        assert.Nil(t, err, tt.name)
      }
    })
  }
}
```


测试结果如下所示。

复制代码

```
» go test -run=TestSqldb_InsertTable
--- FAIL: TestSqldb_InsertTable (0.07s)
    --- FAIL: TestSqldb_InsertTable/insert_wrong_data_type (0.01s)
        sqldb_test.go:171:
                Error Trace:    /Users/jackson/career/crawler/sqldb/sqldb_test.
                Error:          Expected nil, but got: &mysql.MySQLError{Number
                Test:           TestSqldb_InsertTable/insert_wrong_data_type
                Messages:       insert_wrong_data_type
FAIL
exit status 1
FAIL    github.com/dreamerjackson/crawler/sqldb 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。

当然，我们还可以加入 -v 参数打印出详细的信息。

```
» go test -run=InsertTable -v
=== RUN   TestSqldb_InsertTable
=== RUN   TestSqldb_InsertTable/insert_data
=== RUN   TestSqldb_InsertTable/insert_multi_data
=== RUN   TestSqldb_InsertTable/insert_multi_data_wrong_count
=== RUN   TestSqldb_InsertTable/insert_wrong_data_type
    sqldb_test.go:171:
                Error Trace:    /Users/jackson/career/crawler/sqldb/sqldb_test.
                Error:          Expected nil, but got: &mysql.MySQLError{Number
                Test:           TestSqldb_InsertTable/insert_wrong_data_type
                Messages:       insert_wrong_data_type
--- FAIL: TestSqldb_InsertTable (0.07s)
    --- PASS: TestSqldb_InsertTable/insert_data (0.01s)
    --- PASS: TestSqldb_InsertTable/insert_multi_data (0.01s)
    --- PASS: TestSqldb_InsertTable/insert_multi_data_wrong_count (0.00s)
    --- FAIL: TestSqldb_InsertTable/insert_wrong_data_type (0.01s)
FAIL
exit status 1
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。

```
» go test -run=TestSqldb_InsertTable/insert_multi_data_wrong_count  -v
=== RUN   TestSqldb_InsertTable
=== RUN   TestSqldb_InsertTable/insert_multi_data_wrong_count
--- PASS: TestSqldb_InsertTable (0.04s)
    --- PASS: TestSqldb_InsertTable/insert_multi_data_wrong_count (0.00s)
PASS
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

```
func (s *SQLStorage) Flush() error {
  if len(s.dataDocker) == 0 {
    return nil
  }

  defer func() {
    s.dataDocker = nil
  }()
  ...
  return s.db.Insert(sqldb.TableData{
    TableName:   s.dataDocker[0].GetTableName(),
    ColumnNames: getFields(s.dataDocker[0]),
    Args:        args,
    DataCount:   len(s.dataDocker),
  })
}
```

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

```
// sqlstorage_test.go
type mysqldb struct {
}

func (m mysqldb) CreateTable(t sqldb.TableData) error {
  return nil
}

func (m mysqldb) Insert(t sqldb.TableData) error {
  return nil
}
```

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


```
func TestSQLStorage_Flush(t *testing.T) {
  type fields struct {
    dataDocker []*spider.DataCell
    options    options
  }
  tests := []struct {
    name    string
    fields  fields
    wantErr bool
  }{
    {name: "empty", wantErr: false},
    {name: "no Rule filed", fields: fields{dataDocker: []*spider.DataCell{
      {Data: map[string]interface{}{"url": "<http://xxx.com>"}},
    }}, wantErr: true},
  }
  for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
      s := &SQLStorage{
        dataDocker: tt.fields.dataDocker,
        db:         mysqldb{},
        options:    tt.fields.options,
      }
      if err := s.Flush(); (err != nil) != tt.wantErr {
        t.Errorf("Flush() error = %v, wantErr %v", err, tt.wantErr)
      }
      assert.Nil(t, s.dataDocker)
    })
  }
}
```

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

```
ruleName := datacell.Data["Rule"].(string)
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。


```
if ruleName, ok = datacell.Data["Rule"].(string); !ok {
      return errors.New("no rule field")
    }

  if taskName, ok = datacell.Data["Task"].(string); !ok {
    return errors.New("no task field")
  }
```

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

```
# pressure.sh
go test -c # -c会生成可执行文件

PKG=$(basename $(pwd))   # 获取当前路径的最后一个名字，即为文件夹的名字
echo $PKG
while true ; do
        export GOMAXPROCS=$[ 1 + $[ RANDOM % 128 ]] # 随机的GOMAXPROCS
        ./$PKG.test $@ 2>&1    # $@代表可以加入参数    2>&1代表错误输出到控制台
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

```
> /pressure.sh -test.v

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

```
PASS
=== RUN   TestAdd
--- PASS: TestAdd (0.00s)
    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

```go
package escape

import "testing"

type Sumifier interface{ Add(a, b int32) int32 }

type Sumer struct{ id int32 }

func (math Sumer) Add(a, b int32) int32 { return a + b }

type SumerPointer struct{ id int32 }

func (math *SumerPointer) Add(a, b int32) int32 { return a + b }

func BenchmarkDirect(b *testing.B) {
  adder := Sumer{id: 6754}
  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    adder.Add(10, 12)
  }
}

func BenchmarkInterface(b *testing.B) {
  adder := Sumer{id: 6754}
  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    Sumifier(adder).Add(10, 12)
  }
}

func BenchmarkInterfacePointer(b *testing.B) {
  adder := &SumerPointer{id: 6754}
  b.ResetTimer()
  for i := 0; i < b.N; i++ {
```

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

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

```
» go test -gcflags "-N -l"   -bench=.
BenchmarkDirect-12                      535487740                1.95 ns/op
BenchmarkInterface-12                   76026812                 14.6 ns/op
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

```
go test ./fibonacci \\
  -bench BenchmarkSuite \\
  -benchmem \\
  -cpuprofile=cpu.out \\
  -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 fasle，怎么知道是哪一个字段不一致呢？

欢迎你在留言区与我交流讨论，我们下节课见！

上一篇 35丨未雨绸缪：怎样通过静态与动态代码扫描保证代码质量？

下一篇 37丨工具背后的工具：从代码覆盖率到模糊测试

精选留言 (1)

拾掇拾掇

2023-01-09 来自浙江

goland 会自动帮你建好表格测试