go test 测试你的代码

在实际开发中，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。不同的语言通常都有自己的测试包/模块，Go 语言也一样，在 Go 中可以通过 testing 包对代码进行单元和性能测试，下面就来详细介绍。

本节核心内容

- 如何进行单元测试
- 如何进行压力/性能测试
- 如何进行性能分析

> 本小节源码下载路径：[demo15 (https://github.com/lexkong/apiserver_demos/tree/mas](https://github.com/lexkong/apiserver_demos/tree/mas)
>
> 可先下载源码到本地，结合源码理解后续内容，边学边练。
>
> 本小节的代码是基于 [demo14 (https://github.com/lexkong/apiserver_demos/tree/mas](https://github.com/lexkong/apiserver_demos/tree/mas) 来开发的。

Go 语言测试支持

Go 语言有自带的测试框架 testing，可以用来实现单元测试和性能测试，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。其中源码文件需要满足以下规范：

- 文件名必须是 _test.go 结尾，跟源文件在同一个包。
- 测试用例函数必须以 Test、Benchmark、Example 开头
- 执行测试用例时的顺序，会按照源码中的顺序依次执行
- 单元测试函数 TestXxx() 的参数是 testing.T，可以使用该类型来记录错误或测试状态
- 性能测试函数 BenchmarkXxx() 的参数是 testing.B，函数内以 b.N 作为循环次数，其中 N 会动态变化
- 示例函数 ExampleXxx() 没有参数，执行完会将输出与注释 // Output: 进行对比
- 测试函数原型：func TestXxx(t *testing.T)，Xxx 部分为任意字母数字组合，首字母大写，例如： TestgenShortId 是错误的函数名，TestGenShortId 是正确的函数名
- 通过调用 testing.T 的 Error、Errorf、FailNow、Fatal、FatalIf 方法来说明测试不通过，通过调用 Log、Logf 方法来记录测试信息：

```
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，而不能在我们在测试代码创建出的 Goroutine 中调用它
t.Skip      # 调用 t.Skip 方法相当于先后对 t.Log 和 t.SkipNow 方法进行调用，而调用 t.Skipf 方法则相当于先后对 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
```

编写测试用例（对 GenShortId 函数进行单元测试）

1. 在 util 目录下创建文件 util_test.go，内容为：

```
package util

import (
    "testing"
)

func TestGenShortId(t *testing.T) {
    shortId, err := GenShortId()
    if shortId == "" || err != nil {
        t.Error("GenShortId failed!")
    }

    t.Log("GenShortId test pass")
}
```

从用例可以看出，如果 GenShortId() 返回的 shortId 为空或者 err 不为空，则调用 t.Error() 函数标明该用例测试不通过。

执行用例

在 util 目录下执行命令 go test：

```
$ cd util/
$ go test
PASS
ok      apiserver/util  0.006s
```

要查看更详细的执行信息可以执行 go test -v：

```
$ go test -v
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
    util_test.go:13: GenShortId test pass
PASS
ok      apiserver/util  0.006s
```

根据 go test 的输出可以知道 TestGenShortId 用例测试通过。

如果要执行测试 N 次可以使用 -count N：

```
$ go test -v -count 2
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
    util_test.go:13: GenShortId test pass
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
    util_test.go:13: GenShortId test pass
PASS
ok      apiserver/util  0.006s
```

编写性能测试用例

在 util/util_test.go 测试文件中，新增两个性能测试函数：BenchmarkGenShortId() 和 BenchmarkGenShortIdTimeConsuming()（详见 [demo15/util/util_test.go (https://github.com/lexkong/apiserver_demos/blob/master/](https://github.com/lexkong/apiserver_demos/blob/master/)

```
func BenchmarkGenShortId(b *testing.B) {
    for i := 0; i < b.N; i++ {
        GenShortId()
    }
}

func BenchmarkGenShortIdTimeConsuming(b
*testing.B) {
    b.StopTimer() // 调用该函数停止压力测试的时间计数

    shortId, err := GenShortId()
    if shortId == "" || err != nil {
        b.Error(err)
    }

    b.StartTimer() // 重新开始时间

    for i := 0; i < b.N; i++ {
        GenShortId()
    }
}
```

说明

- 性能测试函数名必须以 Benchmark 开头，如 BenchmarkXxx 或 Benchmark_xxx
- go test 默认不会执行压力测试函数，需要通过指定参数 -test.bench 来运行压力测试函数，-test.bench 后跟正则表达式，如 go test -test.bench=".*" 表示执行所有的压力测试函数
- 在压力测试中，需要在循环体中指定 testing.B.N 来循环执行压力测试代码

执行压力测试

在 util 目录下执行命令 go test -test.bench=".*":

```
$ go test -test.bench=".*"
goos: linux
goarch: amd64
pkg: apiserver/util
BenchmarkGenShortId-2                          500000
2291 ns/op
BenchmarkGenShortIdTimeConsuming-2             500000
2333 ns/op
PASS
ok      apiserver/util  2.373s
```

- 上面的结果显示，我们没有执行任何 TestXXX 的单元测试函数，只执行了压力测试函数
- 第一条显示了 BenchmarkGenShortId 执行了 500000 次，每次的执行平均时间是 2291 纳秒
- 第二条显示了 BenchmarkGenShortIdTimeConsuming 执行了 500000，每次的平均执行时间是 2333 纳秒
- 最后一条显示总执行时间

> BenchmarkGenShortIdTimeConsuming 比 BenchmarkGenShortId 多了两个调用 b.StopTimer() 和 b.StartTimer()。
>
> - b.StopTimer()：调用该函数停止压力测试的时间计数
> - b.StartTimer()：重新开始时间
>
> 在 b.StopTimer() 和 b.StartTimer() 之间可以做一些准备工作，这样这些时间不影响我们测试函数本身的性能。

查看性能并生成函数调用图

1. 执行命令：

```
$ go test -bench=".*" -cpuprofile=cpu.profile
./util
```

上述命令会在当前目录下生成 cpu.profile 和 util.test 文件。

2. 执行 go tool pprof util.test cpu.profile 查看性能（进入交互界面后执行 top 指令）：

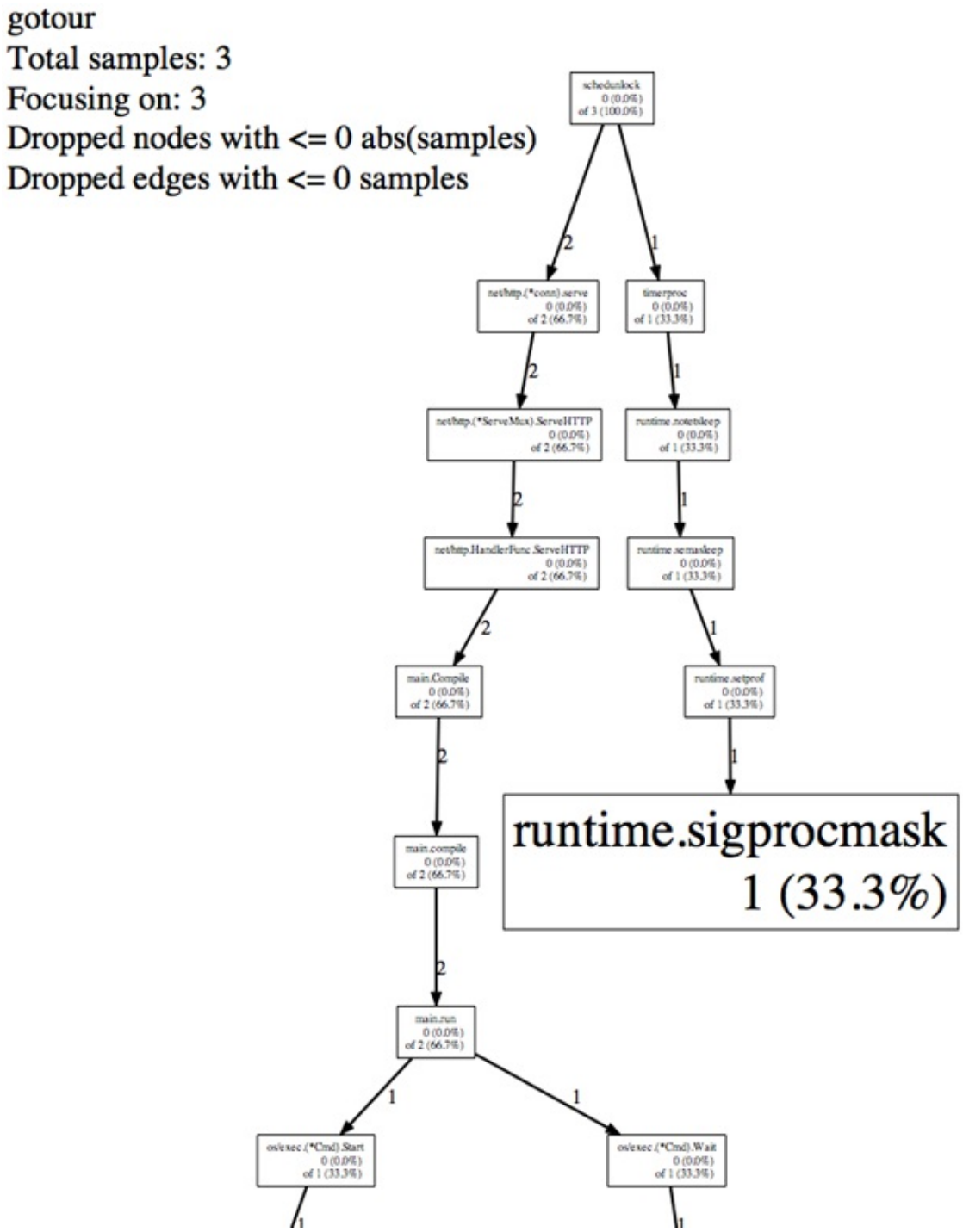
```
$ go tool pprof util.test cpu.profile

File: util.test
Type: cpu
Time: Jun 5, 2018 at 7:28pm (CST)
Duration: 4.93s, Total samples = 4.97s (100.78%)
Entering interactive mode (type "help" for
```

```
commands, "o" for options)
(pprof) top
Showing nodes accounting for 3480ms, 70.02% of
4970ms total
Dropped 34 nodes (cum <= 24.85ms)
Showing top 10 nodes out of 75
      flat  flat%   sum%        cum   cum%
    1890ms 38.03% 38.03%     1900ms 38.23%
syscall.Syscall
     500ms 10.06% 48.09%      620ms 12.47%
runtime.mallocgc
     240ms  4.83% 52.92%     3700ms 74.45%
vendor/github.com/teris-io/shortid.(*Abc).Encode
     150ms  3.02% 55.94%      200ms  4.02%
runtime.scanobject
     140ms  2.82% 58.75%      640ms 12.88%
runtime.makeslice
     140ms  2.82% 61.57%      280ms  5.63%
runtime.slicerunetostring
     120ms  2.41% 63.98%      120ms  2.41%
math.Log
     110ms  2.21% 66.20%     2430ms 48.89%
io.ReadAtLeast
     110ms  2.21% 68.41%      110ms  2.21%
runtime._ExternalCode
      80ms  1.61% 70.02%      140ms  2.82%
runtime.deferreturn
(pprof)
```

pprof 程序中最重要的命令就是 topN，此命令用于显示 profile 文件中的最靠前的 N 个样本（sample），它的输出格式各字段的含义依次是：

1. 采样点落在该函数中的总时间
2. 采样点落在该函数中的百分比
3. 上一项的累积百分比
4. 采样点落在该函数，以及被它调用的函数中的总时间
5. 采样点落在该函数，以及被它调用的函数中的总次数百分比
6. 函数名

此外，在 pprof 程序中还可以使用 svg 来生成函数调用关系图（需要安装 graphviz），例如：



该调用图生成方法如下：

1. 安装 graphviz 命令

```
# yum -y install graphviz.x86_64
```

2. 执行 go tool pprof 生成 svg 图：

```
$ go tool pprof util.test cpu.profile
File: util.test
Type: cpu
Time: Jun 5, 2018 at 7:28pm (CST)
Duration: 4.93s, Total samples = 4.97s (100.78%)
Entering interactive mode (type "help" for
commands, "o" for options)
(pprof) svg
Generating report in profile001.svg
```

svg 子命令会提示在 $GOPATH/src 目录下生成了一个 svg 文件 profile001.svg。

> 关于如何看懂 pprof 信息，请参考官方文档 Profiling Go Programs (https://blog.golang.org/profiling-go-programs)。
>
> 关于如何做性能分析，请参考郝林大神的文章 go tool pprof (https://github.com/hyper0x/go_command_tutorial/blo

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。

`go test -coverprofile=cover.out`：在测试文件目录下运行测试并统计测试覆盖率

go tool cover -func=cover.out：分析覆盖率文件，可以看出哪些函数没有测试，哪些函数内部的分支没有测试完全，cover 工具会通过执行代码的行数与总行数的比例表示出覆盖率

测试覆盖率

```
$ go test -coverprofile=cover.out
PASS
coverage: 14.3% of statements
ok      apiserver/util  0.006s
[api@centos util]$ go tool cover -func=cover.out
apiserver/util/util.go:8:   GenShortId  100.0%
apiserver/util/util.go:12:  GetReqID    0.0%
total:                  (statements)    14.3%
```

可以看到 GenShortId() 函数测试覆盖率为 100%，GetReqID() 测试覆盖率为 0%。

小结

本小节简单介绍了如何用 testing 包做单元和性能测试。在实际的开发中，要养成编写单元测试代码的好习惯，在项目上线前，最好对一些业务逻辑比较复杂的函数做一些性能测试，提前发现性能问题。

至于怎么去分析性能，比如查找耗时最久的函数等，笔者链接了郝林大神专业的分析方法（[go tool pprof (https://github.com/hyper0x/go_command_tutorial/blob/ma](https://github.com/hyper0x/go_command_tutorial/blob/ma)
更深的分析技巧需要读者在实际开发中自己去探索。