文档

官方文档：https://pkg.go.dev/testing

规则

1. 测试用例文件必须以_test结尾，例如：utilt_test.go
2. 功能测试用例方法必须以Test开头
3. 默认情况下测试用例若正常通过不会打印任何信息，可以通过 -v来打印完整信息
4. 声明后缀为"_test"的包的测试文件将编译为分离包，然后与主测试二进制文件链接并运行
5. go help test 、go help testflag 可查看帮助

go test 模式

1. 本地模式：编译运行当前目录中的测试用例，go test 或 go test -v
2. 列表模式：编译并测试命令行上列出的每一个包，例如：go test util、go test ./... 、go test . 等

flags

常用flag

```
# 运行匹配的基准（性能）测试
-bench regexp

# 指定每一个基准测试的运行时间或者运行次数
# 默认 1s，特殊语法：Nx 表示运行基准测试N次，例如：100x表示运行用例100次
-benchtime t

# 指定测试、基准测试、模糊测试 运行n次，相对于运行了N次go test命令
# 如果指定了-cpu 则为每个GOMAXPROCS值运行n次
-count n

# 启用覆盖率分析
-cover

# go test -cpu=1,2,4 将会执行 3 次，其中 GOMAXPROCS 值分别为 1，2，和 4
-cpu 1,2,4

# 运行匹配的模糊测试，不能对多个包使用
-fuzz regexp

# 设置模糊测试时长（例如:1h30m），默认为一直运行。
# 设置模糊运行次数Nx表示运行模糊测试N次，（例如：1000x）表示运行1000次
-fuzztime t

# 列出匹配的顶层测试，子测试将不被列出
-list regexp

# 允许并行执行 调用了t.Parallel方法的测试用例
# -parallel 的默认值为 GOMAXPROCS
```

```
-parallel n

# 只运行那些与常规测试匹配的测试、示例和模糊测试
#（模糊测试仅用语料库中记录的上一次的异常输入作为测试输入）
-run regexp

# 告诉需要长时间运行的测试用例，缩短运行时间
# 在测试用例当中判定是否需要根据该标记跳过用例
-short

# 执行go test 的超时时长，超时panic，默认为(10m)，设置为0表示禁用该选项
-timeout d

# 打印所有输出
-v
```

基准测试指标导出flag

```
#打印基准的内存分配统计信息。
-benchmem

# 测试完成时将goroutine阻塞数据写入指定文件（block.out）
-blockprofile block.out

# 测试完成将覆盖率数据写入到指定文件（cover.out），需指定 -cover标志
-coverprofile cover.out

# 测试完成时将协程的CPU使用数据写入指定的文件（cpu.out）
-cpuprofile cpu.out

# 测试完成时将内存数据写入到指定文件（mem.out）
-memprofile mem.out

# 测试完成将协程互斥数据写入到指定文件（mutex.out）
-mutexprofile mutex.out

# 退出之前，将执行跟踪写入指定的文件（trace.out）
-trace trace.out

#将分析的输出文件放在指定的目录中，默认情况下，运行“go test”的目录
-outputdir directory
```

TestMain

单元测试运行前，可通过TestMain函数做一些初始化工作

```
func TestMain(m *testing.M) {
    // 如有必要 run 之前可以做一些准备工作
    m.Run()
}
```

功能测试

作用

检查应用程序按预期输入能否产生预期输出

测试用例

1. 一般用例

```
func TestParseSize(t *testing.T) {
    ...
    t.Errorf("测试结果不符合预期：%+v", data)
}
```

2. 子测试与并行测试

```
func TestParseSize(t *testing.T) {
    if testing.Short() {
        t.Skip("short mod 跳过该测试用例")
    }
    t.Run("sub1", func(t *testing.T) {
    // 并行信号表明此测试将与其他并行测试并行运行（且仅与其他并行检测并行运行）
        //  t.Parallel()
        ...
        t.Errorf("测试结果不符合预期：%+v", data)
  })
    t.Run("sub2", func(t *testing.T) {
    // 并行信号表明此测试将与其他并行测试并行运行（且仅与其他并行检测并行运行）
        //  t.Parallel()
        ...
        t.Errorf("测试结果不符合预期：%+v", data)
  })
}
```

运行测试

1. 运行当前目录下测试用例

```
go test
go test -v
```

2. 运行指定目录下测试用例

```
# 当前目录
go test .
# 所有目录
go test ./...
# util 目录
go test ./util
# 运行包含ParseSize 的用例
go test -run ParseSize ./util
```

模糊测试

作用

对应用程序输入预期之外的输入，检查程序是否会出现意料之外的情况，比如：报错，panic等。当模糊测试时发生故障，导致问题的输入将会被写入到下次运行的种子语料库文件中。语料库位置：testdata\fuzz\

测试用例

```
//模糊测试
func FuzzParseSize(f *testing.F) {
    f.Fuzz(func(t *testing.T, a string) {
        t.Error("输入异常导致错误")
    })
}
```

运行测试

1. 运行功能测试，默认也会运行模糊测试，只是模糊测试会从语料库中取值，若语料库为空则直接通过
2. 运行模糊测试，运行模糊测试的同时，功能测试并不受影响，会正常进行

```
# 运行当前目录下模糊测试用例
go test -fuzz .
# 运行./util 目录下的模糊测试，匹配名称包含ParseSize的模糊测试
go test -fuzz ParseSize ./util
# 仅运行模糊测试
go test -run ^$  -fuzz ParseSize ./util
```

3. 模糊测试，只允许匹配一个，否则将无法运行

基准（性能）测试

作用

衡量固定条件下，应用程序的性能

测试用例

1. 一般用例

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

2. 子测试

```
//子测试
func BenchmarkParseSizeSub(b *testing.B) {
    //子测试
    b.Run(k, func(b *testing.B) {
        //重置计时
        b.ResetTimer()
        //暂停计时
        b.StopTimer()
    //开始计时
        b.StartTimer()
        for i := 0; i < b.N; i++ {
        ...
        }
    })
}
```

3. 并行测试

RunParallel 并行运行基准测试。它创建多个 goroutine 并在它们之间分配 b.N 次迭代。 goroutine 的数量默认为 GOMAXPROCS。可通过SetParallelism(p)方法配合-cpu设置goroutine数，
数量为：p * GOMAXPROCS

```
func BenchmarkParseSizeParallel(b *testing.B) {
    //此处 goroutine 数量为 10 * GOMAXPROCS
    b.SetParallelism(10)
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            ...
        }
    })
}
```

运行测试

```
# 运行当前目录下测试用例
go test --bench .
# 运行./util 目录下的基准测试，匹配名称包含ParseSize的基准测试
go test -bench ParseSize ./util
# 仅运行基准测试
go test -run ^$  -bench ParseSize ./util
```

http测试

作用

测试http 请求处理函数的功能、性能等。由net/http/httptest 包提供http测试。