Go编码规范V1.0

1 文件

文件名应一律使用小写， 不同单词之间用下划线分割, 不用驼峰式， 命名应尽可能地见名知意。看见文件名就可以知道这个文件下的大概内容,其中测试文件以_test.go结尾。

① 平台区分
文件名_平台。
例： file_windows.go, file_unix.go
可选为windows, unix, posix, plan9, darwin, bsd, linux, freebsd, nacl, netbsd, openbsd, solaris, dragonfly, bsd, notbsd， android， stubs

② 测试单元
文件名*test.go*或者 文件名平台_test.go。
例： path_test.go, path_windows_test.go

③ 版本区分
文件名_版本号等。
例：trap_windows_1.4.go

④ CPU类型区分, 汇编用的多
文件名_(平台:可选)_CPU类型.
例：vdso_linux_amd64.go
可选为amd64, none, 386, arm, arm64, mips64, s390,mips64x,ppc64x, nonppc64x, s390x, x86,amd64p32

说明

① golang的变量命名需要使用小驼峰命名法，且不能出现下划线

② golang中根据首字母的大小写来确定可以访问的权限。无论是方法名、常量、变量名还是结构体的名称，如果首字母大写，则可以被其他的包访问；如果首字母小写，则只能在本包中使用，可以简单的理解成，首字母大写是公有的，首字母小写是私有的

③ 结构体中属性名的大写
如果属性名小写则在数据解析（如json解析,或将结构体作为请求或访问参数）时无法解析

注意点

① go build 的时候会选择性地编译以系统名结尾的文件(linux、darwin、windows、freebsd)。例如Linux(Unix)系统下编译只会选择array_linux.go文件，其它系统命名后缀文件全部忽略。

② 在xxx.go文件的文件头上添加 // + build !windows (tags)，可以选择在windows系统下面不编译

2 注释

// 注释
注意：// 与注释内容间有一个空格

3 命名规范

① 需要注释来补充的命名就不算是好命名。

② 使用可搜索的名称：单字母名称和数字常量很难从一大堆文字中搜索出来。单字母名称仅适用于短方法中的本地变量，名称长短应与其作用域相对应。若变量或常量可能在代码中多处使用，则应赋其以便于搜索的名称。
③ 做有意义的区分：product 和 productInfo 和 productData 没有区别，nameString 和 name 没有区别，要区分名称，就要以读者能鉴别不同之处的方式来区分 。
④ 函数命名规则：函数命名采用驼峰命名，其中单元测试中的测试函数，以及模块中需要导出的函数必须采用大驼峰命名，其他普通函数可以采用小驼峰命名。名字可以长但是得把功能，必要的参数描述清楚，函数名应当是动词或动词短语，如 postPayment、deletePage、save。并依 Javabean 标准加上 get、set、is前缀。例如：xxx + With + 需要的参数名 + And + 需要的参数名 + ...
⑤ 结构体命名规则：结构体名应该是名词或名词短语，如 Costume、WikiPage、Account、AddressParser，避免使用 Manager、Processor、Data、Info、这样的类名，类名不应当是动词。
⑥ 包名命名规则：包名应该为小写单词，不要使用下划线或者混合大小写。
⑦ 接口命名规则：单个函数的接口名以”er”作为后缀，如 Reader,Writer。接口的实现则去掉“er”。

4 函数

函数声明：

```
func min（x int，y int）int {
        如果x <y {
                返回x
        }
        返回y
}

func flushCache（begin，end uintptr）//外部实现
```

5 packet

Go程序是通过将程序包链接在一起而构造的。一个包又由一个或多个源文件构造而成，这些源文件一起声明了属于该包的常量，类型，变量和函数，并且可以在同一包的所有文件中进行访问。这些元素可以 导出并在另一个包中使用。

使用事例

```
        package name
```

name是导入路径（import path）的默认包名。在一个包中所有文件必须用相同包名。

6 工程组织

参考：https://studygolang.com/articles/1644

src 下面保存的是应用源代码
pkg 下面存放的是函数包
bin 目录下面存的是编译之后可执行的文件

目录结构例子如下：

```
<project>
  |--<src>
     |--<a>
        |--<a1>
           |--a1.go
        |--<a2>
           |--a2.go
     |--<b>
        |--b1.go
        |--b2.go
     |--<c>
        |--c.go
  |--<pkg>
  |—<bin>
```

step 1: 将project加入$GOPATH:

```
ubuntu@VM-0-2-ubuntu:~/project/src/b$ echo $GOPATH
/home/ubuntu/project
```

```go
package a1

import "fmt"

func PrintA1(){
    fmt.Println("a/a1")
}

// a1.go
// package a1
```

```go
package a2
import "fmt"

func PrintA2(){
   fmt.Println("a/a2")
}
// a2.go
// package a2
```

```go
package b

import "fmt"

func printB1(){
    fmt.Println("b.b1")
}
// b1.go
// package b
~
```

```go
package b

import "fmt"

func PrintB() {
        printB1()
        fmt.Println("b.b2")
}
// b2.go
// package b2 注意： 一个文件夹中只能有一个package
```

```go
package main

import(
    "a/a1"
    "a/a2"
    "b"
)
func main(){
    a1.PrintA1()
    a2.PrintA2()

    b.PrintB()
}
// c.go

```

```
ubuntu@VM-0-2-ubuntu:~$ tree project/
project/
├── bin
│   └── c
├── pkg
│   ├── linux_amd64
│   │   ├── a
│   │   │   ├── a1.a
│   │   │   └── a2.a
│   │   └── b.a
│   └── mod
│       └── cache
│           └── lock
└── src
    ├── a
    │   ├── a1
    │   │   └── a1.go
    │   └── a2
    │       └── a2.go
    ├── b
    │   ├── b1.go
    │   └── b2.go
    └── c
        └── c.go
```

step 2:
go build a/a1
go install a/a1
go build a/a2
go install a/a2
go build b
go install b

go build c
go install c

```
ubuntu@VM-0-2-ubuntu:~/project/bin$ ./c
a/a1
a/a2
b.b1
b.b2
```

8 单元测试

参考：https://golang.org/pkg/testing/
https://www.cnblogs.com/Dominic-Ji/articles/11659896.html
https://www.jianshu.com/p/1adc69468b6f　　//包含单元测试和性能测试
https://blog.csdn.net/lichangrui2009/article/details/86563982
https://github.com/ethereum/go-ethereum/blob/master/tests/state_test.go

单元测试文件名命名规范为 example_test.go 测试用例的函数名称必须以 Test 开头。

测试文件名以及函数命名具体要求如下：

① 文件名必须是`_test.go`结尾的(文件名必须是`*_test.go`的类型，*代表要测试的文件名），这样在执行`go test`的时候才会执行到相应的代码
② 你必须import`testing`这个包
③ 所有的测试用例函数必须是`Test`开头（函数名必须以Test开头如：TestXxx或Test_xxx）
④ 测试用例会按照源代码中写的顺序依次执行
⑤ 测试函数`TestXxx()`的参数是`testing.T`，我们可以使用该类型来记录错误或者是测试状态
⑥ 测试格式：`func TestXxx (t *testing.T)`,`Xxx`部分可以为任意的字母数字的组合，但是首字母不能是小写字母[a-z]，例如`Testintdiv`是错误的函数名。
⑦ 函数中通过调用`testing.T`的`Error`,`Errorf`,`FailNow`,`Fatal`,`FatalIf`方法，说明测试不通过，调用`Log`方法用来记录测试的信息。

go test工具

Go语言中的测试依赖`go test`命令。编写测试代码和编写普通的Go代码过程是类似的，并不需要学习新的语法、规则或工具。

go test命令是一个按照一定约定和组织的测试代码的驱动程序。在包目录内，所有以`_test.go`为后缀名的源代码文件都是`go test`测试的一部分，不会被`go build`编译到最终的可执行文件中。

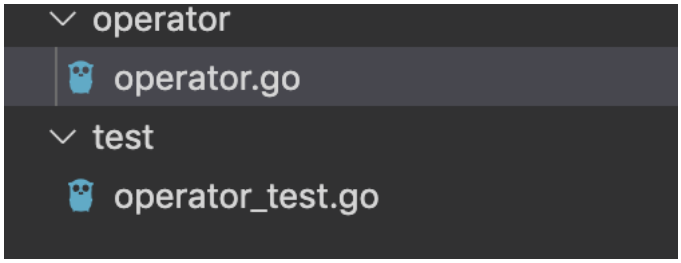
在`*_test.go`文件中有三种类型的函数，单元测试函数、基准测试函数和示例函数。

类型	格式	作用
测试函数	函数名前缀为Test	测试程序的一些逻辑行为是否正确

`go test`命令会遍历所有的`*_test.go`文件中符合上述命名规则的函数，然后生成一个临时的main包用于调用相应的测试函数，然后构建并运行、报告测试结果，最后清理测试中生成的临时文件。

Go自带了测试框架和工具，在testing包中，以便完成单元测试（T类型）和性能测试（B类型）。

测试函数事例:



operator > operator.go

```go
package operator

// 导出的函数首字母必须大写
func Add(a, b int) int {
    c:= a+b
    return c
}

func Subtract(a , b int) int {
    c := a-b
    return c
}

func Multiply(a , b int) int {
    c := a*b
    return c
}

func Divide(a , b int) int {
    c := a/b
    return c
}
```

test > operator_test.go

```go
package test

import (
    "testing"
    "operator"
)

func TestAdd(t *testing.T) {
    t.Log(t.Name(), "Begin...")
    a := 2
    b := 3
    if operator.Add(a, b) != 5 {
        t.Errorf("Error: add(2, 3) should be 5 ,but result is : %d", operator
    }
    t.Logf("a + b is : %d ", a+b)
    t.Log(t.Name(), "End...")
}

func TestSubtract(t *testing.T) {
    t.Log(t.Name(), "Begin...")
    a := 2
    b := 3
    if operator.Subtract(a, b) != -1 {
        t.Errorf("Error: subtract(a, b) should be -1 ,but result is : %d", op
    }
    t.Logf("a - b is : %d ", a-b)
    t.Log(t.Name(), "End...")
}
```

```
ubuntu@VM-0-2-ubuntu:~/project/src/test$ go test -v
=== RUN   TestAdd
    TestAdd: operator_test.go:9: TestAdd Begin...
    TestAdd: operator_test.go:15: a + b is : 5
    TestAdd: operator_test.go:16: TestAdd End...
--- PASS: TestAdd (0.00s)
=== RUN   TestSubtract
    TestSubtract: operator_test.go:20: TestSubtract Begin...
    TestSubtract: operator_test.go:26: a - b is : -1
    TestSubtract: operator_test.go:27: TestSubtract End...
--- PASS: TestSubtract (0.00s)
=== RUN   TestMultiply
    TestMultiply: operator_test.go:31: TestMultiply Begin...
    TestMultiply: operator_test.go:37: a * b is : 6
    TestMultiply: operator_test.go:38: TestMultiply End...
--- PASS: TestMultiply (0.00s)
=== RUN   TestDivide
    TestDivide: operator_test.go:42: TestDivide Begin...
    TestDivide: operator_test.go:48: a / b is : 0
    TestDivide: operator_test.go:49: TestDivide End...
--- PASS: TestDivide (0.00s)
PASS
ok      test    0.003s
```

注意：

① 需要import导入源文件所在的包，使用接口时，也需要使用“package_name.API”的方式引用。

② 一个目录下只能有一个包名。

9 gofmt介绍

参考： https://golang.org/cmd/gofmt/
https://www.jianshu.com/p/104b33439ac2

Golang的开发团队制定了统一的官方代码风格，并且推出了gofmt工具（gofmt或go fmt）来帮助开发者格式化他们的代码到统一的风格。gofmt是一个cli程序，会优先读取标准输入，如果传入了文件路径的话，会格式化这个文件，如果传入一个目录，会格式化目录中所有.go文件，如果不传参数，会格式化当前目录下的所有.go文件。

gofmt默认不对代码进行简化，使用-s参数可以开启简化代码功能，具体来说会进行如下的转换：

① 去除数组、切片、Map初始化时不必要的类型声明：

如下形式的切片表达式：
```
[]T{T{}, T{}}
```
将被简化为：
```
[]T{{}, {}}
```

② 去除数组切片操作时不必要的索引指定

如下形式的切片表达式：
```
s[a:len(s)]
```
将被简化为：
```
s[a:]
```

③ 去除迭代时非必要的变量赋值

如下形式的迭代：
```
for x, _ = range v {...}
```
将被简化为：
```
for x = range v {...}
```
如下形式的迭代：
```
for _ = range v {...}
```
将被简化为：
```
for range v {...}
```

④ 以及自定义的重写规则，使用-r参数，按照pattern -> replacement的格式传入规则。
main.go中代码

```
package main

import "fmt"

func main() {
  a := 1
  b := 2
  c := a + b
  fmt.Println(c)
}
```

使用如下规则格式化上面的代码

```
gofmt -r "a + b -> b + a"
```

格式化后

```
package main

import "fmt"

func main() {
   a := 1
   b := 2
   c := b + a
   fmt.Println(c)
}
```

10 参考：

[1]https://golang.org/ref/spec
[2]https://golang.org/doc/effective_go.html
[3]https://github.com/golang/go/wiki/CodeReviewComments
[4]https://www.cnblogs.com/Dominic-Ji/articles/11659896.html
[5]https://www.jianshu.com/p/1adc69468b6f　　//包含单元测试和性能测试
[6]https://blog.csdn.net/lichangrui2009/article/details/86563982
[7]https://www.jianshu.com/p/104b33439ac2
[8]https://github.com/ethereum/go-ethereum/blob/master/tests/state_test.go
[9]https://golang.org/cmd/gofmt/