**Go** 语⾔编程 **—** 编码规范指南

# ⽬录

### ⽂章⽬录

⽬录

参考

⼯程化要求

编码规范

⼤⼩约定

缩进、括号和空格约定

命名规范

包、⽬录命名规范

⽂件命名规范

标识符命名规范

变量、常量名

函数、⽅法名

结构体、接⼝名

空⾏、注释、⽂档规范

空⾏

注释与⽂档

注释⻛格

包注释

函数、⽅法注释

结构体、接⼝注释

其它说明

导⼊规范

代码逻辑实现规范

变量、常量定义规范

String 类型定义规范

Slice、Map 类型定义规范结构体定义规范

接⼝定义规范

函数、⽅法定义规范

Named Result Parameters

Receiver Names

Receiver Type

错误处理规范

单元测试规范

# 参考

《Effective Go》、 《The Go common mistakes guide》

# ⼯程化要求

建议你在 IDE 中集成下述⼯具插件：

1. 提交代码前，开发人员必须使⽤ gofmt ⼯具格式化代码。注意，gofmt 不识别空⾏，因为 gofmt 不能理解空⾏的意义。

执行：gofmt xxx.go/dir // 输出代码格式化之后的内容

gofmt -s xxx.go/dir // 使用-s参数可以开启简化代码功能

例如：

s := s[1:len(s)]

格式化后：

s := s[1:]

1. 提交代码前，开发人员必须使⽤ goimports ⼯具检查导⼊。

执行：goimports xxx.go // 输出包导入格式化之后的文件（会对包分类换行，删除无用的包等）

1. 提交代码后，CICD需要使⽤ golint ⼯具检查代码规范。

执行：golint xxx.go/dir // 会对命名规范进行检查，例如定义一个包名main\_会有如下报错

root@wangbin:/home/goproject/src/athens/cmd/proxy# golint

main.go:1:1: don't use an underscore in package name

main.go:13:2: a blank import should be only in a main or test package, or have a comment justifying it

1. 提交代码后，CICD需要使⽤ go vet ⼯具静态分析代码实现。

执行：go vet xxx.go/dir // 会对go编码进行静态分析

例如：使用%d 打印一个string类型的变量，会有如下报错

a := "ni hao"

fmt.Printf("print %d", a)

PS E:\gospace\src\mystarter\src> go vet

# mystarter/src

.\main.go:20:2: fmt.Printf format %d has arg a of wrong type string

1. 提交代码后，CID需要使用gocyclo工具分析代码的圈复杂度

执行gocyclo xxx.go/dir

例如：在server.go中的testGocyclo 函数圈复杂度为11

gocyclo -over 10 hsm-os-go-template

11 app testGocyclo hsm-os-go-template/cmd/app/server.go:49:1

1. （可选）提交代码后，CICD需要使用**go test**统计覆盖率，覆盖率不低于50%

PS E:\gospace\src\mystarter\src\controller> go test -cover

--- FAIL: TestCheckHostName (0.00s)

coverage: 40.0% of statements

# 编码规范

## ⼤⼩约定

单个⽂件⻓度尽量不超过 500 ⾏。

单个函数⻓度尽量不超过 50 ⾏。

单个函数圈复杂度尽量不超过 10，禁⽌超过 15。单个函数中嵌套不超过 3 层。

单⾏注释尽量不超过 80 个字符。

单⾏语句尽量不超过 80 个字符。

当单⾏代码超过 80 个字符时，就要考虑分⾏。分⾏的规则是以参数为单位将从较⻓的参数开始换⾏，以此类推直到每⾏⻓度合适。

分⾏的规则是将参数按类型分组。

## 缩进、括号和空格约定

缩进、括号和空格都使⽤ gofmt ⼯具处理。

强制使⽤ tab 缩进。强制左⼤括号不换⾏。

强制所有的运算符和操作数之间要留空格。

## 命名规范

所有命名遵循 “意图” 原则。

### 包、⽬录命名规范

包名和⽬录名保持⼀致。⼀个⽬录尽量维护⼀个包下的所有⽂件。

包名为全⼩写单词， 不使⽤复数，不使⽤下划线。

包名应该尽可能简短。

### ⽂件命名规范

⽂件名为全⼩写单词，使⽤ “\_” 分词。Golang 通常具有以下⼏种代码⽂件类型：

业务代码⽂件

测试代码⽂件

### 标识符命名规范

短名优先，作⽤域越⼤命名越⻓且越有意义。

### 变量、常量名

变量命名遵循驼峰法。

常量使⽤全⼤写单词，使⽤ “\_” 分词。

⾸字⺟根据访问控制原则使⽤⼤写或者⼩写。

对于常规缩略语，⼀旦选择了⼤写或⼩写的⻛格，就应当在整份代码中保持这种⻛格，不要⾸字⺟⼤写和缩写两种⻛格混⽤。以 URL 为例， 如果选择了缩写 URL 这种⻛格，则应在整份代码中保持。错误：UrlArray，正确：urlArray 或 URLArray。再以 ID 为例，如果选择了缩写

ID 这种⻛格，错误：appleId，正确：appleID。

对于只在本⽂件中有效的顶级变量、常量，应该使⽤ “\_” 前缀，避免在同⼀个包中的其他⽂件中意外使⽤错误的值。例如：

注：同一个包下可能会有很多文件，通过“\_” 前缀作为规约，变量只能在本文件中使用。

var (

\_defaultPort = 8080

\_defaultUser = "user"

)

若变量、常量为 bool 类型，则名称应以 Has、Is、Can 或 Allow 开头：

var isExist bool

var hasConflict bool var canManage bool var allowGitHook bool

如果模块的功能较为复杂、常量名称容易混淆的情况下，为了更好地区分枚举类型，可以使⽤完整的前缀：

type PullRequestStatus int

const (

PULL\_REQUEST\_STATUS\_CONFLICT PullRequestStatus = iota PULL\_REQUEST\_STATUS\_CHECKING PULL\_REQUEST\_STATUS\_MERGEABLE

)

### 函数、⽅法名

函数、⽅法（结构体或者接⼝下属的函数称为⽅法）命名规则： 动词 + 名词。

若函数、⽅法为判断类型（返回值主要为 bool 类型），则名称应以 Has、Is、Can 或 Allow 等判断性动词开头：

func HasPrefix(name string, prefixes []string) bool { ... } func IsEntry(name string, entries []string) bool { ... } func CanManage(name string) bool { ... }

func AllowGitHook() bool { ... }

### 结构体、接⼝名

结构体、接口命名规则：名词或名词短语。

单个函数和两个函数接⼝命名规则：接口名综合函数名以 ”er” 作为后缀，例如：Reader、Writer。接⼝实现的⽅法则去掉 “er”，例如：Read、Write。

注：接口是一组操作的抽象，接口命名也是最好能贴近体现所有函数的功能。

type Reader interface {

Read(p []byte) (n int, err error)

}

// 多个函数接⼝

type WriteFlusher interface {

Write([]byte) (int, error)

Flush() error

}

## 空⾏、注释、⽂档规范

### 空⾏

空⾏需要体现代码逻辑的关联，所以空⾏不能随意，⾮常严重地影响可读性。 保持函数内部实现的组织粒度是相近的，⽤空⾏分隔。

### 注释与⽂档

Golang 的 go doc ⼯具可以根据注释⽣成代码⽂档，所以注释的质量决定了代码⽂档的质量。

注释⻛格

统⼀使⽤中⽂注释，中西⽂之间严格使⽤空格分隔，严格使⽤中⽂标点符号。

注释应当是⼀个完整的句⼦，以句号结尾。

句⼦类型的注释⾸字⺟均需⼤写，短语类型的注释⾸字⺟需⼩写。 注释的单⾏⻓度不能超过 80 个字符。

包注释

每个包都应该有⼀个包注释。包注释会⾸先出现在 go doc ⽹⻚上。包注释应该包含： 包名，简介。

创建者。

创建时间。

对于 main 包，通常只有⼀⾏简短的注释⽤以说明包的⽤途，且以项⽬名称开头：

// Gogs (Go Git Service) is a painless self-hosted Git Service. package main

对于简单的⾮ main 包，也可⽤⼀⾏注释概括。

对于⼀个复杂项⽬的⼦包，⼀般情况下不需要包级别注释，除⾮是代表某个特定功能的模块。

对于相对功能复杂的⾮ main 包，⼀般都会增加⼀些使⽤⽰例或基本说明，且以 Package 开头：

/\*

Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is: regexp: concatenation { '|' concatenation } concatenation: { closure } cl

\*/

package regexp

对于特别复杂的包说明，⼀般使⽤ doc.go ⽂件⽤于编写包的描述，并提供与整个包相关的信息。函数、⽅法注释

每个主功能函数、⽅法（结构体或者接⼝下属的函数称为⽅法）都应该有注释说明，包括三个⽅⾯（顺序严格）：

注：函数和方法是否是主功能函数，可以参考砍掉这个函数，是否会影响阅读代码业务的完整性，是否有多次调用。

函数、⽅法名，简要说明。参数列表，每⾏⼀个参数。返回值，每⾏⼀个返回值。

// NewtAttrModel，属性数据层操作类的⼯⼚⽅法。

// 参数：

// ctx：上下⽂信息。

// 返回值：

// 属性操作类指针。

func NewAttrModel(ctx \*common.Context) \*AttrModel {}

如果⼀句话不⾜以说明全部问题，则可换⾏继续进⾏更加细致的描述：

// Copy copies file from source to target path.

// It returns false and error when error occurs in underlying function calls.

若函数或⽅法为判断类型（返回值主要为 bool 类型），则注释以 <name> returns true if 开头：

// HasPrefix returns true if name has any string in given slice as prefix. func HasPrefix(name string, prefixes []string) bool { ...

### 结构体、接⼝注释

每个⾃定义的结构体、接⼝都应该有注释说明，放在实体定义的前⼀⾏，格式为：名称、说明。同时，结构体内的每个成员都要有说明，该说明放 在成员变量的后⾯（注意对⻬），例如：

// User，⽤⼾实例，定义了⽤⼾的基础信息。

type User struct{

Username string // ⽤⼾名 Email string // 邮箱

}

### 其它说明

当某个部分等待完成时，⽤ TODO(Your name): 开头的注释来提醒维护⼈员。

当某个部分存在已知问题进⾏需要修复或改进时，⽤ FIXME(Your name) : 开头的注释来提醒维护⼈员。

当需要特别说明某个问题时，可⽤ NOTE(You name): 开头的注释。

## 导⼊规范

使⽤ goimports ⼯具，在保存⽂件时⾃动检查 import 规范。

如果使⽤的包没有导⼊，则⾃动导⼊；如果导⼊的包没有被使⽤，则⾃动删除。 强制使⽤分⾏导⼊，即便仅导⼊⼀个包。

导⼊多个包时注意按照类别顺序并使⽤空⾏区分：标准库包、程序内部包、第三⽅包。

禁⽌使⽤相对路径导⼊。

禁⽌使⽤ Import Dot（“.”） 简化导⼊。

注：（“.”） 简化导⼊方式，导入的包，可以当做本本包使用，直接调用包内的接口。这种方式从造成代码混乱。

在所有其他情况下，除⾮导⼊之间有直接冲突，否则应避免导⼊别名。

Import( "fmt"

"os"

"runtime/trace"

nettrace "golang.net/x/trace"

)

代码逻辑实现规范

## 变量、常量定义规范

函数内使⽤短变量声明（:=）。

函数外使⽤⻓变量声明（var 关键字），var 关键字⼀般⽤于包级别变量声明，或者函数内的零值情况。

变量、常量的分组声明⼀般需要按照功能来区分，⽽不是将所有类型都分在⼀组：

const (

// Default section name.

DEFAULT\_SECTION = "DEFAULT" // Maximum allowed depth when recursively substituing variable na

)

type ParseError int

const (

ERR\_SECTION\_NOT\_FOUND ParseError = iota + 1

ERR\_KEY\_NOT\_FOUND ERR\_BLANK\_SECTION\_NAME ERR\_COULD\_NOT\_PARSE

)

如果有可能，尽量缩⼩变量的作⽤范围。

// Bad

err := ioutil.WriteFile(name, data, 0644)

if err != nil {

return err

}

// Good

if err := ioutil.WriteFile(name, data, 0644); err != nil {

return err

}

如果是枚举常量，需要先创建相应类型：

type Scheme string

const (

HTTP Scheme = "http"

HTTPS Scheme = "https"

)

⾃构建的枚举类型应该从 1 开始，除⾮从 0 开始是有意义的：

注：从1开始，可以区分枚举实例于类型零值

// Bad

type Operation int

const (

Add Operation = iota Subtract

Multiply

)

// Good

type Operation int

const (

Add Operation = iota + 1 Subtract

Multiply

)

**String** 类型定义规范

声明 Printf-style String 时，将其设置为 const 常量，这有助于 go vet 对 String 类型实例执⾏静态分析。

注：Printf-style String可以理解为string用途只是用来输出。

// Bad

msg := "unexpected values %v, %v\n" fmt.Printf(msg, 1, 2)

// Good

const msg = "unexpected values %v, %v\n" fmt.Printf(msg, 1, 2)

优先使⽤ strconv ⽽不是 fmt，将原语转换为字符串或从字符串转换时，strconv 速度⽐ fmt 快。

// Bad

for i := 0; i < b.N; i++ {

s := fmt.Sprint(rand.Int())

}

// Good

for i := 0; i < b.N; i++ {

s := strconv.Itoa(rand.Int())

}

避免字符串到字节的转换，不要反复从固定字符串创建字节 Slice，执⾏⼀次性完成转换。

// Bad

for i := 0; i < b.N; i++ {

w.Write([]byte("Hello world"))

}

// Good

data := []byte("Hello world")

for i := 0; i < b.N; i++ {

w.Write(data)

}

Slice、Map 类型定义规范

尽可能指定容器的容量，以便为容器预先分配内存，向 make() 传⼊容量参数会在初始化时尝试调整 Slice、Map 类型实例的⼤⼩，这将减少在将元素添加到 Slice、Map 类型实例时的重新分配内存造成的损耗。

在追加 Slice 类型变量时优先指定切⽚容量，在初始化要追加的切⽚时为 make() 提供⼀个容量值。

data := make([]int, 0, size)

for k := 0; k < size; k++ {

data = append(data, k)

}

Map 或 Slice 类型实例是引⽤类型，所以在函数调⽤传递时，要注意在函数内外保证实例数据的安全性，除⾮你知道⾃⼰在做什么。这是⼀个深拷⻉和浅拷⻉的问题。

// Bad

func (d \*Driver) SetTrips(trips []Trip) {

d.trips = trips

}

trips := ...

d1.SetTrips(trips)

// 你是要修改 d1.trips 吗？

trips[0] = ...

// Good

func (d \*Driver) SetTrips(trips []Trip) {

d.trips = make([]Trip, len(trips))

copy(d.trips, trips)

}

trips := ...

d1.SetTrips(trips)

// 这⾥我们修改 trips[0]，但不会影响到 d1.trips。

返回 Map 或 Slice 类型实例时，同样要注意⽤⼾对暴露了内部状态的实例的数值进⾏修改：

// Bad

type Stats struct {

mu sync.Mutex

counters map[string]int

}

// Snapshot 返回当前状态。

func (s \*Stats) Snapshot() map[string]int {

s.mu.Lock()

defer s.mu.Unlock()

return s.counters

}

// snapshot 不再受互斥锁保护。

// 因此对 snapshot 的任何访问都将受到数据竞争的影响。

// Good

type Stats struct {

mu sync.Mutex counters map[string]int

}

func (s \*Stats) Snapshot() map[string]int {

s.mu.Lock()

defer s.mu.Unlock()

result := make(map[string]int, len(s.counters))

for k, v := range s.counters {

result[k] = v

}

return result

}

// snapshot 现在是⼀个拷⻉

snapshot := stats.Snapshot()

结构体定义规范

嵌⼊结构体中作为成员的结构体，应位于结构体内的成员列表的顶部，并且必须有⼀个空⾏将嵌⼊式成员与常规成员分隔开。

在初始化 Struct 类型的指针实例时，使⽤ &T{} 代替 new(T) ，使其与初始化 Struct 类型实例⼀致。

注：通过new(T)关键字创建一个类型实例只会申请一块内存不会对类型的属性进行实例化。隐式返回指针也不易于代码阅读。

sval := T{Name: "foo"}

sptr := &T{Name: "bar"}

接⼝定义规范

如果希望通过接⼝的⽅法修改接⼝实例的实际数据，则必须传递接⼝实例的指针（将实例指针赋值给接⼝变量），因为指针指向真正的内存数据：

type F interface {

f()

}

type S1 struct{}

func (s S1) f() {}

type S2 struct{}

func (s \*S2) f() {}

var f1 F := S1{} // f1.f() ⽆法修改底层数据。

var f2 F := &S2{} // f2.f() 可以修改底层数据，给接⼝变量 f2 赋值时使⽤的是实例指针。

### 函数、⽅法定义规范

函数、⽅法的参数排列顺序遵循以下⼏点原则（从左到右）：

1. 参数的重要程度与逻辑顺序。

2. 简单类型优先于复杂类型。

3. 尽可能将同种类型的参数放在相邻位置，则只需写⼀次类型。

避免实参传递时的语义不明确（Avoid Naked Parameters），当参数名称的含义不明显时，使⽤块注释语法：

func printInfo(name string, isLocal, done bool)

// Bad

printInfo("foo", true, true)

// Good

printInfo("foo", true /\* isLocal \*/, true /\* done \*/)

上述例⼦中，更好的做法是将 bool 类型换成⾃定义类型。将来，该参数可以⽀持不仅仅是两个状态（true/false）

避免使⽤ init() 函数，否则 init() 中的代码应该保证。函数定义的内容不对环境或调⽤⽅式有任何依赖，具有完全确定性。避免依赖于其他init()函数的顺序。

注：虽然顺序是明确的，但代码可以更改，因此 init() 函数之间的关系可能会使代码变得脆弱，容易出错。

init() 函数应避免访问或操作全局或环境状态，如：机器信息、环境变量、⼯作⽬录、程序参数/输⼊等。避免 I/O 操作，包括：⽂件系统、⽹络和系统调⽤。不能满⾜上述要求的代码应该被定义在 main 中（或程序⽣命周期中的其他地⽅）。

### 函数返回值命名

当你需要在函数结束的 defer 中对返回值做⼀些事情，返回值名字是必要的。

// 错误

func (n \*Node) Parent1() \*Node

func (n \*Node) Parent2() (\*Node, error)

// 正确

func (n \*Node) Parent1() (node \*Node)

func (n \*Node) Parent2() (node \*Node, err error)

### 函数接收者规范

函数接收者命名（Receiver Names）

结构体⽅法中，接受者的命名（Receiver Names）不应该采⽤ me，this，self 等通⽤的名字，⽽应该采⽤简短的（1 或 2 个字符）并且能反映出结构体名的命名⻛格，它不必像参数命名那么具体，因为我们⼏乎不关⼼接受者的名字。

例如：Struct Client，接受者可以命名为 c 或者 cl。这样做的好处是，当⽣成了 go doc 后，过⻓或者过于具体的命名，会影响搜索体验。

函数接收者类型（Receiver Type）

编写结构体⽅法时，接受者的类型（Receiver Type）到底是选择值还是指针通常难以决定。

建议：

当接受者是 map、chan、func，不要使⽤指针传递，因为它们本⾝就是引⽤类型。

当接受者是 slice，⽽函数内部不会对 slice 进⾏切⽚或者重新分配空间，不要使⽤指针传递。

当函数内部需要修改接受者，必须使⽤指针传递。

当接受者类型是⼀个 struct 并且很庞⼤，或者是⼀个⼤的 array，建议使⽤指针传递来提⾼性能。

当接受者是⼩型 struct，⼩ array，并且不需要修改⾥⾯的元素，⾥⾯的元素⼜是⼀些基础类型，使⽤值传递是个不错的选择。

当接受者是指针类型需要进行判空，或者添加注释说明不会出现为空的调用场景。

### 错误处理规范

err 总是作为函数返回值列表的最后⼀个。

如果⼀个函数 return error，⼀定要检查它是否为空，判断函数调⽤是否成功。如果不为空，说明发⽣了错误，⼀定要处理它。 不能使⽤ \_ 丢弃任何 return 的 err。若不进⾏错误处理，要么再次向上游 return err，或者使⽤ log 记录下来。

尽早 return err，函数中优先进⾏ return 检测，遇⻅错误则⻢上 return err。

错误提⽰（Error Strings）不需要⼤写字⺟开头的单词，即使是句⼦的⾸字⺟也不需要。除⾮那是个专有名词或者缩写。同时，错误提⽰也不需要以句号结尾，因为通常在打印完错误提⽰后还需要跟随别的提⽰信息。

采⽤独⽴的错误流进⾏处理。尽可能减少正常逻辑代码的缩进，这有利于提⾼代码的可读性，便于快速分辨出哪些还是正常逻辑代码，例如：

// 错误写法

if err != nil { // error handling

} else { // normal code

}

// 正确写法

if err != nil { // error handling return // or continue, etc.

}

// normal code

另⼀种常⻅的情况，如果我们需要⽤函数的返回值来初始化某个变量，应该把这个函数调⽤单独写在⼀⾏，例如：

// 错误写法

if x, err := f(); err != nil { // error handling return

} else { // use x

}

// 正确写法

x, err := f()

if err != nil { // error handling return

}

// use x

尽量不要使⽤ panic，除⾮你知道你在做什么。只有当实在不可运⾏的情况下采⽤ panic，例如：⽂件⽆法打开，数据库⽆法连接导致程序⽆法正常运⾏。但是对于可导出的接⼝不能有 panic，不要抛出 panic 只能在包内采⽤。建议使⽤ log.Fatal 来记录错误，这样就可以由 log来结束程序。

### 单元测试规范

业务代码⽂件和单元测试⽂件放在同⼀⽬录下。

单元测试⽂件名以 \*\_test.go 为后缀，例如：example\_test.go。

测试⽤例的函数名称必须以 Test 开头，例如：TestLogger。

如果为结构体的⽅法编写测试⽤例，则需要以 Text\_<Struct>\_<Method> 的形式命名，例如：Test\_Macaron\_Run。

每个重要的函数都要同步编写测试⽤例。

测试⽤例和业务代码同步提交，⽅便进⾏回归测试。