链滴

Uber Go 风格指南 (译)

作者：Allenxuxu
原文链接：https://ld246.com/article/1570978892782
来源网站：链滴

Uber Go 风格指南

- 译文： https://github.com/Allenxuxu/uber-go-guide
- 原文： https://github.com/uber-go/guide/blob/master/style.md

简介

风格是指规范代码的共同约定。风格一词其实是有点用词不当的，因为共同约定的范畴远远不止 gofm 所做的源代码格式化这些。

本指南旨在通过详尽描述 Uber 在编写 Go 代码中的注意事项（规定）来解释其中复杂之处。制定这注意事项（规定）是为了提高代码可维护性同时也让工程师们高效的使用 Go 的特性。

这份指南最初由 Prashant Varanasi 和 Simon Newton 编写，目的是让一些同事快速上手 Go 。多来，已经根据其他人的反馈不断修改。

这份文档记录了我们在 Uber 遵守的 Go 惯用准则。其中很多准则是 Go 的通用准则，其他方面依赖外部资源：

1. Effective Go
2. The Go common mistakes guide

所有的代码都应该通过 golint 和 go vet 检查。我们建议您设置编辑器：

- 保存时自动运行 goimports
- 自动运行 golint 和 go vet 来检查错误

您可以在这找到关于编辑器设定 Go tools 的相关信息：

https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins

指南

指向接口（interface）的指针

你基本永远不需要一个指向接口的指针。你应该直接将接口作为值传递，因为接口的底层数据就是指。

一个接口包含两个字段：

1. 类型指针，指向某些特定类型信息的指针。
2. 数据指针。如果存储数据是一个指针变量，那就直接存储。如果存储数据是一个值变量，那就存储向该值的指针。

如果你需要接口方法来修改这些底层数据，那你必须使用指针。

方法接收器和接口

具有值类型接收器的方法可以被值类型和指针类型调用。

例如，

```
type S struct {
  data string
}

func (s S) Read() string {
  return s.data
}

func (s *S) Write(str string) {
  s.data = str
}

sVals := map[int]S{1: {"A"}}

// 值类型变量只能调用 Read 方法
sVals[1].Read()

// 无法编译通过:
//  sVals[0].Write("test")

sPtrs := map[int]*S{1: {"A"}}

// 指针类型变量可以调用 Read 和 Write 方法:
sPtrs[1].Read()
sPtrs[1].Write("test")
```

同理，即使方法是值类型接收器，接口也可以通过指针来满足调用需求。

```
type F interface {
```

```
  f()
}

type S1 struct{}

func (s S1) f() {}

type S2 struct{}

func (s *S2) f() {}

s1Val := S1{}
s1Ptr := &S1{}
s2Val := S2{}
s2Ptr := &S2{}

var i F
i = s1Val
i = s1Ptr
i = s2Ptr

// 无法编译通过, 因为 s2Val 是一个值类型变量, 并且 f 方法不具有值类型接收器。
//   i = s2Val
```

Effective Go 中关于 Pointers vs. Values 写的很棒。

零值Mutexes是有效的

零值的 sync.Mutex 和 sync.RWMutex 是有效的，所以基本是不需要一个指向 Mutex 的指针的。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

mu := new(sync.Mutex)
mu.Lock()

</td><td>

var mu sync.Mutex
mu.Lock()

</td></tr>
</tbody></table>
```

如果你希望通过指针操作结构体，mutex 可以作为其非指针结构体字段，或者最好直接嵌入结构体中。

```
<table>
<tbody>
<tr><td>
```

```
type smap struct {
  sync.Mutex

  data map[string]string
}

func newSMap() *smap {
  return &smap{
    data: make(map[string]string),
  }
}

func (m *smap) Get(k string) string {
  m.Lock()
  defer m.Unlock()

  return m.data[k]
}

</td><td>


type SMap struct {
  mu sync.Mutex

  data map[string]string
}

func NewSMap() *SMap {
  return &SMap{
    data: make(map[string]string),
  }
}

func (m *SMap) Get(k string) string {
  m.mu.Lock()
  defer m.mu.Unlock()

  return m.data[k]
}

</td></tr>


</tr>
<tr>
<td>嵌入到非导出类型或者需要实现 Mutex 接口的类型。</td>
<td>对于导出类型，将 mutex 作为私有成员变量。</td>
</tr>


</tbody></table>
```

Slices和Maps的边界拷贝操作

切片和 map 包含一个指针来指向底层数据，所以当需要复制他们时需要特别注意。

接收Slices和Maps

请记住，如果存储了对 slice 或 map 的引用，那么用户是可以对其进行修改。

```
<table>
<thead><tr><th>Bad</th> <th>Good</th></tr></thead>
<tbody>
<tr>
<td>

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = trips
}

trips := ...
d1.SetTrips(trips)

// 是想修改 d1.trips 吗?
trips[0] = ...

</td>
<td>

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = make([]Trip, len(trips))
  copy(d.trips, trips)
}

trips := ...
d1.SetTrips(trips)

// 修改 trips[0] 并且不影响 d1.trips 。
trips[0] = ...

</td>
</tr>

</tbody>
</table>
```

返回 Slices 和 Maps

同理，谨慎提防用户修改暴露内部状态的 slices 和 maps 。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>
```

```
type Stats struct {
  sync.Mutex

  counters map[string]int
}

// Snapshot 返回当前状态
func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  return s.counters
}

// snapshot 不再受锁保护了!
snapshot := stats.Snapshot()
```

</td><td>

```
type Stats struct {
  sync.Mutex

  counters map[string]int
}

func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  result := make(map[string]int, len(s.counters))
  for k, v := range s.counters {
    result[k] = v
  }
  return result
}

// snapshot 是一分拷贝的内容了
snapshot := stats.Snapshot()
```

</td></tr>
</tbody></table>

使用 defer 来做清理工作

使用 defer 来做资源的清理工作，例如文件的关闭和锁的释放。

<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

```
p.Lock()
```

```
if p.count < 10 {
  p.Unlock()
  return p.count
}

p.count++
newCount := p.count
p.Unlock()

return newCount

// 当有多处 return 时容易忘记释放锁

</td><td>


p.Lock()
defer p.Unlock()

if p.count < 10 {
  return p.count
}

p.count++
return p.count

// 可读性更高

</td></tr>
</tbody></table>
```

defer 只有非常小的性能开销，只有当你能证明你的函数执行时间在纳秒级别时才可以不使用它。使用 defer 对代码可读性的提高是非常值得的，因为使用 defer 的成本真的非常小。特别是在一些主要是内存操作的长函数中，函数中的其他计算操作远比 defer 重要。

Channel 的大小设为 1 还是 None

通道的大小通常应该设为 1 或者设为无缓冲类型。默认情况下，通道是无缓冲类型的，大小为 0 。将道大小设为其他任何数值都应该经过深思熟虑。认真考虑如何确定其大小，是什么阻止了工作中的通被填满并阻塞了写入操作，以及何种情况会发生这样的现象。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>


// 足以满足任何人!
c := make(chan int, 64)

</td><td>


// 大小 为 1
c := make(chan int, 1) // or
```

```
// 无缓冲 channel, 大小为 0
c := make(chan int)

</td></tr>
</tbody></table>
```

枚举类型值从 1 开始

在 Go 中使用枚举的标准方法是声明一个自定义类型并通过 iota 关键字来声明一个 const 组。但是于 Go 中变量的默认值都为该类型的零值，所以枚举变量的值应该从非零值开始。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

type Operation int

const (
  Add Operation = iota
  Subtract
  Multiply
)

// Add=0, Subtract=1, Multiply=2

</td><td>

type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

// Add=1, Subtract=2, Multiply=3

</td></tr>
</tbody></table>
```

在某些情况下，从零值开始也是可以的。例如，当零值是我们期望的默认行为时。

```
type LogOutput int

const (
  LogToStdout LogOutput = iota
  LogToFile
  LogToRemote
)

// LogToStdout=0, LogToFile=1, LogToRemote=2
```

错误类型

有很多种方法来声明 errors:

- errors.New 声明简单的静态字符串错误信息
- fmt.Errorf 声明格式化的字符串错误信息
- 为自定义类型实现 Error() 方法
- 通过 "pkg/errors".Wrap 包装错误类型

返回错误时，请考虑以下因素来作出最佳选择：

- 这是一个不需要其他额外信息的简单错误吗？如果是，使用 error.New。
- 客户需要检测并处理此错误吗？如果是，那应该自定义类型，并实现 Error() 方法。
- 是否是在传递一个下游函数返回的错误？如果是，请查看 error 封装部分。
- 其他，使用 fmt.Errorf 。

如果客户需要检测错误，并且是通过 errors.New 创建的一个简单的错误，请使用var 声明这个错误型。

<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

```
// package foo

func Open() error {
  return errors.New("could not open")
}

// package bar

func use() {
  if err := foo.Open(); err != nil {
    if err.Error() == "could not open" {
      // handle
    } else {
      panic("unknown error")
    }
  }
}
```

</td><td>

```
// package foo

var ErrCouldNotOpen = errors.New("could not open")

func Open() error {
  return ErrCouldNotOpen
```

```
}

// package bar

if err := foo.Open(); err != nil {
  if err == foo.ErrCouldNotOpen {
    // handle
  } else {
    panic("unknown error")
  }
}

</td></tr>
</tbody></table>
```

如果你有一个错误需要客户端来检测，并且你想向其添加更多信息（例如，它不是一个简单的静态字串），那么应该声明一个自定义类型。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func open(file string) error {
  return fmt.Errorf("file %q not found", file)
}

func use() {
  if err := open(); err != nil {
    if strings.Contains(err.Error(), "not found") {
      // handle
    } else {
      panic("unknown error")
    }
  }
}

</td><td>

type errNotFound struct {
  file string
}

func (e errNotFound) Error() string {
  return fmt.Sprintf("file %q not found", e.file)
}

func open(file string) error {
  return errNotFound{file: file}
}

func use() {
  if err := open(); err != nil {
```

```
    if _, ok := err.(errNotFound); ok {
      // handle
    } else {
      panic("unknown error")
    }
  }
}

</td></tr>
</tbody></table>
```

直接将自定义的错误类型设为导出需要特别小心，因为这意味着他们已经成为包的公开 API 的一部分。更好的方式是暴露一个匹配函数来检测错误。

```
// package foo

type errNotFound struct {
  file string
}

func (e errNotFound) Error() string {
  return fmt.Sprintf("file %q not found", e.file)
}

func IsNotFoundError(err error) bool {
  _, ok := err.(errNotFound)
  return ok
}

func Open(file string) error {
  return errNotFound{file: file}
}

// package bar

if err := foo.Open("foo"); err != nil {
  if foo.IsNotFoundError(err) {
    // handle
  } else {
    panic("unknown error")
  }
}
```

Error 封装

下面提供三种主要的方法来传递函数调用失败返回的错误：

- 如果想要维护原始错误类型并且不需要添加额外的上下文信息，就直接返回原始错误。
- 使用 `"pkg/errors".Wrap` 来增加上下文信息，这样返回的错误信息中就会包含更多的上下文信息并且通过 `"pkg/errors".Cause` 可以提取出原始错误信息。
- 如果调用方不需要检测或处理特定的错误情况，就直接使用 `fmt.Errorf` 。

情况允许的话建议增加更多的上下文信息来代替诸如 "connection refused" 之类模糊的错误信息。回 "failed to call service foo: connection refused" 用户可以知道更多有用的错误信息。

在将上下文信息添加到返回的错误时，请避免使用 "failed to" 之类的短语以保持信息简洁，这些短描述的状态是显而易见的，并且会随着错误在堆栈中的传递而逐渐堆积：

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>


s, err := store.New()
if err != nil {
    return fmt.Errorf(
        "failed to create new store: %s", err)
}

</td><td>


s, err := store.New()
if err != nil {
    return fmt.Errorf(
        "new store: %s", err)
}

<tr><td>


failed to x: failed to y: failed to create new store: the error

</td><td>


x: y: new store: the error

</td></tr>
</tbody></table>
```

但是，如果这个错误信息是会被发送到另一个系统时，必须清楚的表明这是一个错误（例如，日志中 err 标签或者 Failed 前缀）。

另见 Don't just check errors, handle them gracefully。

处理类型断言失败

类型断言的单返回值形式在遇到类型错误时会直接 panic 。因此，请始终使用 "comma ok" 惯用方。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>


t := i.(string)
```

```
</td><td>

t, ok := i.(string)
if !ok {
  // handle the error gracefully
}

</td></tr>
</tbody></table>
```

不要 Panic

生产级的代码必须避免 panics 。panics 是级联故障的主要源头。如果错误发生，函数应该返回错误且允许调用者决定如果处理它。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func foo(bar string) {
  if len(bar) == 0 {
    panic("bar must not be empty")
  }
  // ...
}

func main() {
  if len(os.Args) != 2 {
    fmt.Println("USAGE: foo <bar>")
    os.Exit(1)
  }
  foo(os.Args[1])
}

</td><td>

func foo(bar string) error {
  if len(bar) == 0
    return errors.New("bar must not be empty")
  }
  // ...
  return nil
}

func main() {
  if len(os.Args) != 2 {
    fmt.Println("USAGE: foo <bar>")
    os.Exit(1)
  }
  if err := foo(os.Args[1]); err != nil {
    panic(err)
```

```
  }
}

</td></tr>
</tbody></table>
```

Panic/recover 并不是错误处理策略。程序只有在遇到无法处理的情况下才可以 panic ，例如，nil 用。程序初始化时是一个例外情况：程序启动时遇到需要终止执行的错误可能会 painc 。

```
var _statusTemplate = template.Must(template.New("name").Parse("_statusHTML"))
```

即使是在测试中，也应优先选择 t.Fatal 或 t.FailNow 而非 panic，以确保测试标记为失败。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

// func TestFoo(t *testing.T)

f, err := ioutil.TempFile("", "test")
if err != nil {
  panic("failed to set up test")
}

</td><td>

// func TestFoo(t *testing.T)

f, err := ioutil.TempFile("", "test")
if err != nil {
  t.Fatal("failed to set up test")
}

</td></tr>
</tbody></table>
```

使用 go.uber.org/atomic

Go 的 sync/atomic 包仅仅提供针对原始类型（int32, int64, ...）的原子操作。因此，很容易忘记使原子操作来读写变量。

go.uber.org/atomic 通过隐藏基础类型，使这些操作类型安全。并且，它还提供一个方便的 atomic.ool 类型。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

type foo struct {
  running int32  // atomic
```

```
}

func (f* foo) start() {
  if atomic.SwapInt32(&f.running, 1) == 1 {
     // already running…
     return
  }
  // start the Foo
}

func (f *foo) isRunning() bool {
  return f.running == 1  // race!
}

</td><td>

type foo struct {
  running atomic.Bool
}

func (f *foo) start() {
  if f.running.Swap(true) {
     // already running…
     return
  }
  // start the Foo
}

func (f *foo) isRunning() bool {
  return f.running.Load()
}

</td></tr>
</tbody></table>
```

性能

性能方面的特定准则，仅适用于热路径。

strconv 性能优于 fmt

将原语转换为字符串或从字符串转换时，strconv 速度比 fmt 更快。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

for i := 0; i < b.N; i++ {
  s := fmt.Sprint(rand.Int())
}
```

```
</td><td>

for i := 0; i < b.N; i++ {
  s := strconv.Itoa(rand.Int())
}

</td></tr>
<tr><td>

BenchmarkFmtSprint-4    143 ns/op    2 allocs/op

</td><td>

BenchmarkStrconv-4    64.2 ns/op    1 allocs/op

</td></tr>
</tbody></table>
```

避免 string to byte 的转换

不要反复地从字符串字面量创建 byte 切片。相反，执行一次转换后存储结果供后续使用。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

for i := 0; i < b.N; i++ {
  w.Write([]byte("Hello world"))
}

</td><td>

data := []byte("Hello world")
for i := 0; i < b.N; i++ {
  w.Write(data)
}

</tr>
<tr><td>

BenchmarkBad-4   50000000   22.2 ns/op

</td><td>

BenchmarkGood-4  500000000   3.25 ns/op

</td></tr>
</tbody></table>
```

代码风格

声明分组

Go 支持将相似的声明分组：

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

import "a"
import "b"

</td><td>

import (
  "a"
  "b"
)

</td></tr>
</tbody></table>
```

分组同样适用于常量、变量和类型的声明：

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

const a = 1
const b = 2

var a = 1
var b = 2

type Area float64
type Volume float64

</td><td>

const (
  a = 1
  b = 2
)

var (
  a = 1
  b = 2
)
```

```
type (
  Area float64
  Volume float64
)

</td></tr>
</tbody></table>
```

仅将相似的声明放在同一组。不相关的声明不要放在同一个组内。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
  ENV_VAR = "MY_ENV"
)

</td><td>

type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

const ENV_VAR = "MY_ENV"

</td></tr>
</tbody></table>
```

声明分组可以在任意位置使用。例如，可以在函数内部使用。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func f() string {
  var red = color.New(0xff0000)
  var green = color.New(0x00ff00)
  var blue = color.New(0x0000ff)

  ...
```

```
}

</td><td>

func f() string {
  var (
    red   = color.New(0xff0000)
    green = color.New(0x00ff00)
    blue  = color.New(0x0000ff)
  )

  ...
}

</td></tr>
</tbody></table>
```

Import 组内顺序

import 有两类导入组：

- 标准库
- 其他

goimports 默认的分组如下：

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

import (
  "fmt"
  "os"
  "go.uber.org/atomic"
  "golang.org/x/sync/errgroup"
)

</td><td>

import (
  "fmt"
  "os"

  "go.uber.org/atomic"
  "golang.org/x/sync/errgroup"
)

</td></tr>
</tbody></table>
```

包名

当为包命名时，请注意如下事项：

- 字符全部小写，没有大写或者下划线
- 在大多数情况下引入包不需要去重命名
- 简单明了，命名需要能够在被导入的地方准确识别
- 不要使用复数。例如， net/url, 而不是 net/urls
- 不要使用“common”，“util”，“shared”或“lib”之类的。这些都是不好的，表达信息不明名称

另见 Package Names 和 Style guideline for Go packages

函数命名

我们遵循 Go 社区关于使用的 MixedCaps for function names。有一种情况例外，对相关的测试用进行分组时，函数名可能包含下划线，如: TestMyFunction_WhatIsBeingTested。

包导入别名

如果包的名称与导入路径的最后一个元素不匹配，那必须使用导入别名。

```
import (
  "net/http"

  client "example.com/client-go"
  trace "example.com/trace/v2"
)
```

在其他情况下，除非导入的包名之间有直接冲突，否则应避免使用导入别名。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

import (
  "fmt"
  "os"


  nettrace "golang.net/x/trace"
)

</td><td>

import (
  "fmt"
  "os"
  "runtime/trace"
```

```
  nettrace "golang.net/x/trace"
)

</td></tr>
</tbody></table>
```

函数分组与排布顺序

- 函数应该粗略的按照调用顺序来排布
- 同一文件中的函数应该按照接收器的类型来分组排布

所以，公开的函数应排布在文件首，并在 struct、const 和 var 定义之后。

newXYZ()/ NewXYZ() 之类的函数应该排布在声明类型之后，具有接收器的其余方法之前。

因为函数是按接收器类别分组的，所以普通工具函数应排布在文件末尾。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func (s *something) Cost() {
  return calcCost(s.weights)
}

type something struct{ ... }

func calcCost(n int[]) int {...}

func (s *something) Stop() {...}

func newSomething() *something {
    return &something{}
}

</td><td>

type something struct{ ... }

func newSomething() *something {
    return &something{}
}

func (s *something) Cost() {
  return calcCost(s.weights)
}

func (s *something) Stop() {...}

func calcCost(n int[]) int {...}
```

```
</td></tr>
</tbody></table>
```

减少嵌套

代码应该通过尽可能地先处理错误情况/特殊情况，并且及早返回或继续下一循环来减少嵌套。尽量少嵌套于多个级别的代码数量。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

for _, v := range data {
  if v.F1 == 1 {
    v = process(v)
    if err := v.Call(); err == nil {
      v.Send()
    } else {
      return err
    }
  } else {
    log.Printf("Invalid v: %v", v)
  }
}

</td><td>

for _, v := range data {
  if v.F1 != 1 {
    log.Printf("Invalid v: %v", v)
    continue
  }

  v = process(v)
  if err := v.Call(); err != nil {
    return err
  }
  v.Send()
}

</td></tr>
</tbody></table>
```

不必要的 else

如果一个变量在 if 的两个分支中都设置了，那应该使用单个 if 。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
```

```
<tr><td>

var a int
if b {
  a = 100
} else {
  a = 10
}

</td><td>

a := 10
if b {
  a = 100
}

</td></tr>
</tbody></table>
```

全局变量声明

在顶层使用标准 var 关键字声明变量时，不要显式指定类型，除非它与表达式的返回类型不同。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

var _s string = F()

func F() string { return "A" }

</td><td>

var _s = F()
// F 已经明确声明返回一个字符串类型，我们没有必要显式指定 _s 的类型

func F() string { return "A" }

</td></tr>
</tbody></table>
```

如果表达式的返回类型与所需的类型不完全匹配，请显示指定类型。

```
type myError struct{}

func (myError) Error() string { return "error" }

func F() myError { return myError{} }

var _e error = F()
// F 返回一个 myError 类型的实例，但是我们要 error 类型
```

非导出的全局变量或者常量以 _ 开头

非导出的全局变量和常量前面加上前缀 _，以明确表示它们是全局符号。

例外：未导出的错误类型变量，应以 err 开头。

解释：顶级（全局）变量和常量具有包范围作用域。使用通用名称命名，可能在其他文件中不经意间使用一个错误值。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

// foo.go

const (
  defaultPort = 8080
  defaultUser = "user"
)

// bar.go

func Bar() {
  defaultPort := 9090
  ...
  fmt.Println("Default port", defaultPort)

  // We will not see a compile error if the first line of
  // Bar() is deleted.
}

</td><td>

// foo.go

const (
  _defaultPort = 8080
  _defaultUser = "user"
)

</td></tr>
</tbody></table>
```

结构体中的嵌入类型

嵌入式类型（例如 mutex ）应该放置在结构体字段列表的顶部，并且必须以空行与常规字段隔开。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>
```

```
type Client struct {
  version int
  http.Client
}

</td><td>

type Client struct {
  http.Client

  version int
}

</td></tr>
</tbody></table>
```

使用字段名来初始化结构

初始化结构体时，必须指定字段名称。go vet 强制执行。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

k := User{"John", "Doe", true}

</td><td>

k := User{
    FirstName: "John",
    LastName: "Doe",
    Admin: true,
}

</td></tr>
</tbody></table>
```

例外：在测试文件中，如果结构体只有3个或更少的字段，则可以省略字段名称。

```
tests := []struct{
}{
  op Operation
  want string
}{
  {Add, "add"},
  {Subtract, "subtract"},
}
```

局部变量声明

如果声明局部变量时需要明确设值，应使用短变量声明形式（:=）。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

var s = "foo"

</td><td>

s := "foo"

</td></tr>
</tbody></table>
```

但是，在某些情况下，使用 var 关键字声明变量，默认的初始化值会更清晰。例如，声明空切片。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func f(list []int) {
  filtered := []int{}
  for _, v := range list {
    if v > 10 {
      filtered = append(filtered, v)
    }
  }
}

</td><td>

func f(list []int) {
  var filtered []int
  for _, v := range list {
    if v > 10 {
      filtered = append(filtered, v)
    }
  }
}

</td></tr>
</tbody></table>
```

nil是一个有效的slice

nil 是一个有效的长度为 0 的 slice，这意味着：

- 不应明确返回长度为零的切片，而应该直接返回 nil 。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

if x == "" {
  return []int{}
}

</td><td>

if x == "" {
  return nil
}

</td></tr>
</tbody></table>
```

- 若要检查切片是否为空，始终使用 `len(s) == 0`，不要与 nil 比较来检查。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

func isEmpty(s []string) bool {
  return s == nil
}

</td><td>

func isEmpty(s []string) bool {
  return len(s) == 0
}

</td></tr>
</tbody></table>
```

- 零值切片（通过 var 声明的切片）可直接使用，无需调用 make 创建。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

nums := []int{}
// or, nums := make([]int)

if add1 {
  nums = append(nums, 1)
}

if add2 {
```

```
  nums = append(nums, 2)
}

</td><td>

var nums []int

if add1 {
  nums = append(nums, 1)
}

if add2 {
  nums = append(nums, 2)
}

</td></tr>
</tbody></table>
```

缩小变量作用域

如果有可能，尽量缩小变量作用范围，除非这样与减少嵌套的规则冲突。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

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

</td><td>

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

</td></tr>
</tbody></table>
```

如果需要在 if 之外使用函数调用的结果，则不应尝试缩小范围。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

if data, err := ioutil.ReadFile(name); err == nil {
  err = cfg.Decode(data)
  if err != nil {
```

```
    return err
  }

  fmt.Println(cfg)
  return nil
} else {
  return err
}

</td><td>

data, err := ioutil.ReadFile(name)
if err != nil {
   return err
}

if err := cfg.Decode(data); err != nil {
  return err
}

fmt.Println(cfg)
return nil

</td></tr>
</tbody></table>
```

避免裸参数

函数调用中的裸参数可能会降低代码可读性。所以当参数名称的含义不明显时，请为参数添加 C 样式注释（/* ... */）。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

// func printInfo(name string, isLocal, done bool)

printInfo("foo", true, true)

</td><td>

// func printInfo(name string, isLocal, done bool)

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

</td></tr>
</tbody></table>
```

上面更好的作法是将 bool 类型替换为自定义类型，从而使代码更易读且类型安全。将来需要拓展时该参数也可以不止两个状态（true/false）。

```
type Region int

const (
  UnknownRegion Region = iota
  Local
)

type Status int

const (
  StatusReady = iota + 1
  StatusDone
  // 也许将来我们会有 StatusInProgress。
)

func printInfo(name string, region Region, status Status)
```

使用原始字符串字面值，避免使用转义

Go 支持原始字符串字面值，可以多行并包含引号。使用它可以避免使用肉眼阅读较为困难的手工转的字符串。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

wantError := "unknown name:\"test\""

</td><td>

wantError := `unknown error:"test"`

</td></tr>
</tbody></table>
```

初始化结构体引用

在初始化结构引用时，使用 &T{} 而非 new(T)，以使其与结构体初始化方式保持一致。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

sval := T{Name: "foo"}

// 定义方式不一致
sptr := new(T)
sptr.Name = "bar"

</td><td>
```

```
sval := T{Name: "foo"}

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

</td></tr>
</tbody></table>
```

格式化字符串放在 Printf 外部

如果为 Printf-style 函数声明格式化字符串，将格式化字符串放在函数外面 ，并将其设置为 const 常。

这有助于 go vet 对格式字符串进行静态分析。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

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

</td><td>

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

</td></tr>
</tbody></table>
```

为 Printf 样式函数命名

声明 Printf-style 函数时，请确保 go vet 可以检查它的格式化字符串。

这意味着应尽可能使用预定义的 Printf-style 函数名称。go vet 默认会检查它们。更多相关信息，请见 Printf系列。

如果不能使用预定义的名称，请以 f 结尾：Wrapf，而非 Wrap。因为 go vet 可以指定检查特定的 Printf 样式名称，但名称必须以 f 结尾。

```
$ go vet -printfuncs=wrapf,statusf
...
```

另见 go vet: Printf family check

模式

测试表

在核心测试逻辑重复时，将表驱动测试与子测试一起使用，以避免重复代码。

<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

```
// func TestSplitHostPort(t *testing.T)

host, port, err := net.SplitHostPort("192.0.2.0:8000")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "8000", port)

host, port, err = net.SplitHostPort("192.0.2.0:http")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "http", port)

host, port, err = net.SplitHostPort(":8000")
require.NoError(t, err)
assert.Equal(t, "", host)
assert.Equal(t, "8000", port)

host, port, err = net.SplitHostPort("1:8")
require.NoError(t, err)
assert.Equal(t, "1", host)
assert.Equal(t, "8", port)
```

</td><td>

```
// func TestSplitHostPort(t *testing.T)

tests := []struct{
  give     string
  wantHost string
  wantPort string
}{
  {
    give:     "192.0.2.0:8000",
    wantHost: "192.0.2.0",
    wantPort: "8000",
  },
  {
    give:     "192.0.2.0:http",
    wantHost: "192.0.2.0",
    wantPort: "http",
  },
  {
    give:     ":8000",
    wantHost: "",
    wantPort: "8000",
  },
  {
    give:     "1:8",
```

```
    wantHost: "1",
    wantPort: "8",
  },
}

for _, tt := range tests {
  t.Run(tt.give, func(t *testing.T) {
    host, port, err := net.SplitHostPort(tt.give)
    require.NoError(t, err)
    assert.Equal(t, tt.wantHost, host)
    assert.Equal(t, tt.wantPort, port)
  })
}

</td></tr>
</tbody></table>
```

测试表使得向错误消息注入上下文信息，减少重复的逻辑，添加新的测试用例变得更加容易。

我们遵循这样的约定：将结构体切片称为 tests。 每个测试用例称为 tt 。此外，我们鼓励使用 give 和 want 前缀说明每个测试用例的输入和输出值。

```
tests := []struct{
  give     string
  wantHost string
  wantPort string
}{
  // ...
}

for _, tt := range tests {
  // ...
}
```

功能选项

功能选项是一种模式，声明一个不透明 Option 类型，该类型记录某些内部结构体的信息。您的函数受这些不定数量的选项参数，并将选项参数上的信息作用于内部结构上。

此模式可用于扩展构造函数和实现其他公共 API 中的可选参数，特别是这些参数已经有三个或者超过个的情况下。

```
<table>
<thead><tr><th>Bad</th><th>Good</th></tr></thead>
<tbody>
<tr><td>

// package db

func Connect(
  addr string,
  timeout time.Duration,
  caching bool,
```

```
) (*Connection, error) {
  // ...
}

// Timeout and caching must always be provided,
// even if the user wants to use the default.

db.Connect(addr, db.DefaultTimeout, db.DefaultCaching)
db.Connect(addr, newTimeout, db.DefaultCaching)
db.Connect(addr, db.DefaultTimeout, false /* caching */)
db.Connect(addr, newTimeout, false /* caching */)

</td><td>


type options struct {
  timeout time.Duration
  caching bool
}

// Option overrides behavior of Connect.
type Option interface {
  apply(*options)
}

type optionFunc func(*options)

func (f optionFunc) apply(o *options) {
  f(o)
}

func WithTimeout(t time.Duration) Option {
  return optionFunc(func(o *options) {
    o.timeout = t
  })
}

func WithCaching(cache bool) Option {
  return optionFunc(func(o *options) {
    o.caching = cache
  })
}

// Connect creates a connection.
func Connect(
  addr string,
  opts ...Option,
) (*Connection, error) {
  options := options{
    timeout: defaultTimeout,
    caching: defaultCaching,
  }

  for _, o := range opts {
    o.apply(&options)
```

```
  }

  // ...
}

// Options must be provided only if needed.

db.Connect(addr)
db.Connect(addr, db.WithTimeout(newTimeout))
db.Connect(addr, db.WithCaching(false))
db.Connect(
  addr,
  db.WithCaching(false),
  db.WithTimeout(newTimeout),
)

</td></tr>
</tbody></table>
```

另见，

- Self-referential functions and the design of options
- Functional options for friendly APIs