Uber Go 语言编程规范

相信很多人前两天都看到 Uber 在 github 上面开源的 Go 语言编程规范了，原文在这里：https://github.com/uber-go/guide/blob/master/style.md 。我们今天就来简单了解一下国外大厂都是如何来写代码的。行文仓促，错误之处，多多指正。另外如果觉得还不错，也欢迎分享给更多的人。

1. 介绍

英文原文标题是 ***Uber Go Style Guide***，这里的 Style 是指在管理我们代码的时候可以遵从的一些约定。

这篇编程指南的初衷是更好的管理我们的代码，包括去编写什么样的代码，以及不要编写什么样的代码。我们希望通过这份编程指南，代码可以具有更好的维护性，同时能够让我们的开发同学更高效地编写 Go 语言代码。

这份编程指南最初由 Prashant Varanasi 和 Simon Newton 编写，旨在让其他同事快速地熟悉和编写 Go 程序。经过多年发展，现在的版本经过了多番修改和改进了。这是我们在 Uber 遵从的编程范式，但是很多都是可以通用的，如下是其他可以参考的链接：

- Effective Go
- The Go common mistakes guide

所有的提交代码都应该通过 `golint` 和 `go vet` 检测，建议在代码编辑器上面做如下设置：

- 保存的时候运行 `goimports`
- 使用 `golint` 和 `go vet` 去做错误检测。

你可以通过下面链接发现更多的 Go 编辑器的插件: https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins

2. 编程指南

2.1 指向 Interface 的指针

在我们日常使用中，基本上不会需要使用指向 interface 的指针。当我们将 interface 作为值传递的时候，底层数据就是指针。Interface 包括两方面：

- 一个包含 type 信息的指针
- 一个指向数据的指针

如果你想要修改底层的数据，那么你只能使用 pointer。

2.2 Receiver 和 Interface

使用值作为 receiver 的时候 method 可以通过指针调用，也可以通过值来调用。

```
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"}}

// You can only call Read using a value
sVals[1].Read()

// This will not compile:
//  sVals[1].Write("test")

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

// You can call both Read and Write using a pointer
sPtrs[1].Read()
sPtrs[1].Write("test")
```

相似的，pointer 也可以满足 interface 的要求，尽管 method 使用 value 作为 receiver。

```
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

// The following doesn't compile, since s2Val is a value, and there is
no value receiver for f.
//   i = s2Val
```

Effective Go 关于如何使用指针和值也有一些不错的 practice： Pointers vs. Values.

2.3 mutex 默认 0 值是合法的

`sync.Mutex` 和 `sync.RWMutex` 的 0 值也是合法的，所以我们基本不需要声明一个指针指向 mutex。

Bad

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

Good

```
var mu sync.Mutex
mu.Lock()
```

如果 struct 内部使用 mutex，在我们使用 struct 的指针类型时候，mutex 也可以是一个非指针类型的 field，或者直接嵌套在 struct 中。

Mutex 直接嵌套在 struct 中。

```
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]
}
```

将 Mutex 作为一个 struct 内部一个非指针类型 Field 使用。

```
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]
}
```

2.4 拷贝 Slice 和 Map

Slice 和 Map 都包含了对底层存储数据的指针，所以注意在修改 slice 或者 map 数据的场景下，是不是使用了引用。

slice 和 map 作为参数

当把 slice 和 map 作为参数的时候，如果我们对 slice 或者 map 的做了引用操作，那么修改会修改掉原始值。如果这种修改不是预期的，那么要先进行 copy。

Bad

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

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

// Did you mean to modify d1.trips?
trips[0] = ...
```

Good

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

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

// We can now modify trips[0] without affecting d1.trips.
trips[0] = ...
```

slice 和 map 作为返回值

当我们的函数返回 slice 或者 map 的时候，也要注意是不是直接返回了内部数据的引用到外部。

Bad

```go
type Stats struct {
  sync.Mutex

  counters map[string]int
}

// Snapshot returns the current stats.
func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  return s.counters
}

// snapshot is no longer protected by the lock!
snapshot := stats.Snapshot()
```

Good

```go
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 is now a copy.
snapshot := stats.Snapshot()
```

2.5 使用 defer 做资源清理

建议使用 defer 去做资源清理工作，比如文件，锁等。

Bad

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

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

return newCount

// easy to miss unlocks due to multiple returns
```

Good

```
p.Lock()
defer p.Unlock()

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

p.count++
return p.count

// more readable
```

尽管使用 defer 会导致一定的性能开销，但是大部分情况下这个开销在你的整个链路上所占的比重往往是微乎其微，除非说真的是有非常高的性能需求。另外使用 defer 带来的代码可读性的改进以及减少代码发生错误的概率都是值得的。

2.6 channel 的 size 最好是 1 或者是 unbuffered

在使用 channel 的时候，最好将 size 设置为 1 或者使用 unbuffered channel。其他 size 的 channel 往往都会引入更多的复杂度，需要更多考虑上下游的设计。

Bad

```
// Ought to be enough for anybody!
c := make(chan int, 64)
```

Good

```
// Size of one
c := make(chan int, 1) // or
// Unbuffered channel, size of zero
c := make(chan int)
```

2.7 枚举变量应该从 1 开始

在 Go 语言中枚举值的声明典型方式是通过 `const` 和 `iota` 来声明。由于 0 是默认值，所以枚举值最好从一个非 0 值开始，比如 1。

Bad

```
type Operation int

const (
  Add Operation = iota
  Subtract
  Multiply
)

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

Good

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

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

有一种例外情况：0 值是预期的默认行为的时候，枚举值可以从 0 开始。

```
type LogOutput int

const (
  LogToStdout LogOutput = iota
  LogToFile
  LogToRemote
)

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

2.8 Error 类型

在 Go 语言中声明 error 可以有多种方式：

- `errors.New` 声明包含简单静态字符串的 error
- `fmt.Errorf` 格式化 error string
- 其他自定义类型使用了 `Error()` 方法
- 使用 `"pkg/errors".Wrap`

当要把 error 作为返回值的时候，可以考虑如下的处理方式

- 是不是不需要额外信息，如果是，`errors.New` 就足够了。
- client 需要检测和处理返回的 error 吗？如果是，最好使用实现了 `Error()` 方法的自定义类型，这样可以包含更多的信息。
- error 是不是从下游函数传递过来的？如果是，考虑一下 error wrap，参考：section on error wrapping.
- 其他情况，`fmt.Errorf` 一般足够了。

对于 client 需要检测和处理 error 的情况，这里详细说一下。如果你要通过 `errors.New` 声明一个简单的 error，那么可以使用一个变量声明：`var ErrCouldNotOpen = errors.New("Could not open")`

Bad

```go
// 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")
    }
  }
}
```

Good

```go
// 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")
  }
}
```

如果需要 error 中包含更多的信息，而不仅仅类型原生 error 的这种简单字符串，那么最好使用一个自定义类型。

Bad

```
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")
    }
  }
}
```

Good

```
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")
```

```
    }
  }
}
```

在直接暴露自定义的 error 类型的时候，最好 export 配套的检测自定义 error 类型的函数。

```
// 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")
  }
}
```

2.9 Error Wrapping

在函数调用失败的时候，有三种方式可以将下游的 error 传递出去：

- 直接返回失败函数返回的 error。
- 使用 `"pkg/errors".Wrap` 增加更多的上下文信息，这种情况下可以使用 `"pkg/errors".Cause` 去提取原始的 error 信息。
- 如果调用者不需要检测和处理返回的 error 信息的话，可以直接使用 `fmt.Errorf` 将需要附加的信息进行格式化添加进去。

如果条件允许，最好增加上下文信息。比如 "connection refused" 和 "call service foo: connection refused" 这两种错误信息在可读性上比较也是高下立判。当增加上下文信息的时候，尽量保持简洁。比如像 "failed to" 这种极其明显的信息就没有必要写上去了。

Bad

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

Good

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

另外对于需要传播到其他系统的 error，也要有明显的标识信息，比如在 log 的最前面增加 `err` 等字样。

更多参考：[Don't just check errors, handle them gracefully](#).

2.10 类型转换失败处理

类型转换失败会导致进程 panic，所以对于类型转换，一定要使用 "comma ok" 的范式来处理。

Bad

```
t := i.(string)
```

Good

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

2.11 不要 panic

对于线上环境要尽量避免 panic。在很多情况下，panic 都是引起雪崩效应的罪魁祸首。一旦 error 发生，我们应该向上游调用者返回 error，并且容许调用者对 error 进行检测和处理。

Bad

```go
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])
}
```

Good

```go
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)
  }
}
```

Panic/Recover 并不是一种 error 处理策略。进程只有在某些不可恢复的错误发生的时候才需要 panic。

在跑 test case 的时候，使用 `t.Fatal` 或者 `t.FailNow`，而不是 panic 来保证这个 test case 会被标记为失败的。

Bad

```go
// func TestFoo(t *testing.T)

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

Good

```go
// func TestFoo(t *testing.T)

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

2.12 使用 go.uber.org/atomic

这个是 Uber 内部对原生包 `sync/atomic` 的一种封装，隐藏了底层数据类型。

Bad

```go
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!
}
```

Good

```go
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()
}
```

3. 性能相关

3.1 类型转换时，使用 strconv 替换 fmt

当基本类型和 string 互转的时候，`strconv` 要比 `fmt` 快。

Bad

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

BenchmarkFmtSprint-4    143 ns/op    2 allocs/op
```

Good

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

BenchmarkStrconv-4    64.2 ns/op    1 allocs/op
```

3.2 避免 string to byte 的不必要频繁转换

在通过 string 创建 byte slice 的时候，不要在循环语句中重复的转换，而是要将重复的转换逻辑提到循环外面，做一次即可。(看上去很 general 的建议)

Bad

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

BenchmarkBad-4   50000000   22.2 ns/op
```

Good

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

BenchmarkGood-4  500000000   3.25 ns/op
```

4. 编程风格

4.1 声明语句分组

import 语句分组

Bad

```
import "a"
import "b"
```

Good

```
import (
  "a"
  "b"
)
```

常量、变量以及 type 声明

Bad

```
const a = 1
const b = 2

var a = 1
var b = 2

type Area float64
type Volume float64
```

Good

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

var (
```

```
  a = 1
  b = 2
)

type (
  Area float64
  Volume float64
)
```

import 根据导入的包进行顺序分组。（其他库我们其实可以再细分 private 库和 public 库）

- 标准库
- 其他库

Bad

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

Good

```
import (
  "fmt"
  "os"

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

4.2 package 命名

package 命名的几条规则：

- 全小写。不包含大写字母或者下划线。
- 简洁。
- 不要使用复数。比如，使用 `net/url`，而不是 `net/urls`。
- 避免："common", "util", "shared", "lib"，不解释。

更多参考：

- Package Names
- Style guideline for Go packages.

4.3 函数命名

函数命名遵从社区规范： MixedCaps for function names 。有一种特例是 TestCase 中为了方便测试做的函数命名，比如： `TestMyFunction_WhatIsBeingTested` 。

4.4 import 别名

当 package 的名字和 import 的 path 的最后一个元素不同的时候，必须要起别名。

```
import (
  "net/http"

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

另外，import 别名要尽量避免，只要在不得不起别名的时候再这么做，比如避免冲突。

Bad

```
import (
  "fmt"
  "os"

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

Good

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

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

4.5 函数分组和排序

- 函数应该按调用顺序排序
- 一个文件中的函数应该按 receiver 排序

`newXYZ/NewXYZ` 最好紧接着类型声明后面，并在其他的 receiver 函数前面。

Bad

```go
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{}
}
```

Good

```go
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 {...}
```

4.6 避免代码块嵌套

优先处理异常情况，快速返回，避免代码块过多嵌套。看下面代码会比较直观。

Bad

```go
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)
  }
}
```

Good

```go
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()
}
```

4.7 避免不必要的 else 语句

很多情况下，if - else 语句都能通过一个 if 语句表达，比如如下代码。

Bad

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

Good

```go
a := 10
if b {
  a = 100
}
```

4.8 两级 (two-level) 变量声明

所有两级变量声明就是一个声明的右值来自另一个表达式，这个时候第一级变量声明就不需要指明类型，除非这两个地方的数据类型不同。看代码会更直观一点。

Bad

```go
var _s string = F()

func F() string { return "A" }
```

Good

```go
var _s = F()
// Since F already states that it returns a string, we don't need to specify
// the type again.

func F() string { return "A" }
```

上面说的第二种两边数据类型不同的情况。

```go
type myError struct{}

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

func F() myError { return myError{} }

var _e error = F()
// F returns an object of type myError but we want error.
```

4.9 对于不做 export 的全局变量使用前缀 _

对于同一个 package 下面的多个文件，一个文件中的全局变量可能会被其他文件误用，所以建议使用 _ 来做前缀。（其实这条规则有待商榷）

Bad

```go
// 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.
}
```

Good

```
// foo.go

const (
  _defaultPort = 8080
  _defaultUser = "user"
)
```

4.10 struct 嵌套

struct 中的嵌套类型在 field 列表排在最前面，并且用空行分隔开。

Bad

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

Good

```
type Client struct {
  http.Client

  version int
}
```

4.11 struct 初始化的时候带上 Field

这样会更清晰，也是 go vet 鼓励的方式

Bad

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

Good

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

4.12 局部变量声明

变量声明的时候可以使用 `:=` 以表示这个变量被显示的设置为某个值。

Bad

```
var s = "foo"
```

Good

```
s := "foo"
```

但是对于某些情况使用 var 反而表示的更清晰，比如声明一个空的 slice: Declaring Empty Slices

Bad

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

Good

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

4.13 nil 是合法的 slice

在返回值是 slice 类型的时候，直接返回 nil 即可，不需要显式地返回长度为 0 的 slice。

Bad

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

Good

```
if x == "" {
  return nil
}
```

判断 slice 是不是空的时候，使用 `len(s) == 0`。

Bad

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

Good

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

使用 var 声明的 slice 空值可以直接使用，不需要 `make()`。

Bad

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

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

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

Good

```go
var nums []int

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

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

4.14 避免 scope

Bad

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

Good

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

当然某些情况下，scope 是不可避免的，比如

Bad

```go
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
}
```

Good

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

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

fmt.Println(cfg)
return nil
```

4.15 避免参数语义不明确（Avoid Naked Parameters）

Naked Parameter 指的应该是意义不明确的参数，这种情况会破坏代码的可读性，可以使用 C 分格的注释（`/*...*/`）进行注释。

Bad

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

printInfo("foo", true, true)
```

Good

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

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

对于上面的示例代码，还有一种更好的处理方式是将上面的 bool 类型换成自定义类型。

```go
type Region int

const (
  UnknownRegion Region = iota
  Local
)

type Status int

const (
  StatusReady = iota + 1
  StatusDone
  // Maybe we will have a StatusInProgress in the future.
)

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

4.16 使用原生字符串，避免转义

Go 支持使用反引号，也就是 "`" 来表示原生字符串，在需要转义的场景下，我们应该尽量使用这种方案来替换。

Bad

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

Good

```
wantError := `unknown error:"test"`
```

4.17 Struct 引用初始化

使用 `&T{}` 而不是 `new(T)` 来声明对 T 类型的引用，使用 `&T{}` 的方式我们可以和 struct 声明方式 `T{}` 保持统一。

Bad

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

// inconsistent
sptr := new(T)
sptr.Name = "bar"
```

Good

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

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

4.18 字符串 string format

如果我们要在 Printf 外面声明 format 字符串的话，使用 const，而不是变量，这样 go vet 可以对 format 字符串做静态分析。

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)
```

4.19 Printf 风格函数命名

当声明 `Printf` 风格的函数时，确保 `go vet` 可以对其进行检测。可以参考：Printf family 。

另外也可以在函数名字的结尾使用 f 结尾，比如: `WrapF`，而不是 `Wrap`。然后使用 `go vet`

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

更多参考: go vet: Printf family check.

5. 编程模式（Patterns）

5.1 Test Tables

当测试逻辑是重复的时候，通过 subtests 使用 table 驱动的方式编写 case 代码看上去会更简洁。

Bad

```
// 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)
```

Good

```
// 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)
  })
}
```

很明显，使用 test table 的方式在代码逻辑扩展的时候，比如新增 test case，都会显得更加的清晰。

在命名方面，我们将 struct 的 slice 命名为 `tests`，同时每一个 test case 命名为 `tt`。而且，我们强烈建议通过 `give` 和 `want` 前缀来表示 test case 的 input 和 output 的值。

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

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

5.2 Functional Options

关于 functional options 简单来说就是通过类似闭包的方式来进行函数传参。

Bad

```go
// 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 */)
```

Good

```go
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),
)
```

更多参考：

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

注：关于 functional option 这种方式我本人也强烈推荐，我很久以前也写过一篇类似的文章，感兴趣的可以移步： 写扩展性好的代码：函数

6. 总结

Uber 开源的这个文档，通篇读下来给我印象最深的就是：保持代码简洁，并具有良好可读性。不得不说，相比于国内很多 “代码能跑就完事了” 这种写代码的态度，这篇文章或许可以给我们更多的启示和参考。