uber-go/guide 的中文翻译

English

Uber Go 语言编码规范

Uber 是一家美国硅谷的科技公司，也是 Go 语言的早期 adopter。其开源了很多 golang 项目，诸如被 Gopher 圈熟知的 zap、jaeger 等。2018 年年末 Uber 将内部的 Go 风格规范 开源到 GitHub，经过一年的积累和更新，该规范已经初具规模，并受到广大 Gopher 的关注。本文是该规范的中文版本。本版本会根据原版实时更新。

版本

- 当前更新版本：2024-08-10 版本地址：commit:#217
- 如果您发现任何更新、问题或改进，请随时 fork 和 PR
- Please feel free to fork and PR if you find any updates, issues or improvement.

目录

介绍

样式 (style) 是支配我们代码的惯例。术语 样式 有点用词不当，因为这些约定涵盖的范围不限于由 gofmt 替我们处理的源文件格式。

本指南的目的是通过详细描述在 Uber 编写 Go 代码的注意事项来管理这种复杂性。这些规则的存在是为了使代码库易于管理，同时仍然允许工程师更有效地使用 Go 语言功能。

该指南最初由 Prashant Varanasi 和 Simon Newton 编写，目的是使一些同事能快速使用 Go。多年来，该指南已根据其他人的反馈进行了修改。

本文档记录了我们在 Uber 遵循的 Go 代码中的惯用约定。其中许多是 Go 的通用准则，而其他扩展准则依赖于下面外部的指南：

1. Effective Go
2. Go Common Mistakes
3. Go Code Review Comments

我们的目标是使代码示例能够准确地用于 Go 的两个发布版本 releases.

所有代码都应该通过 `golint` 和 `go vet` 的检查并无错误。我们建议您将编辑器设置为：

- 保存时运行 `goimports`
- 运行 `golint` 和 `go vet` 检查错误

您可以在以下 Go 编辑器工具支持页面中找到更为详细的信息：
https://go.dev/wiki/IDEsAndTextEditorPlugins

指导原则

指向 interface 的指针

您几乎不需要指向接口类型的指针。您应该将接口作为值进行传递，在这样的传递过程中，实质上传递的底层数据仍然可以是指针。

接口实质上在底层用两个字段表示：

1. 一个指向某些特定类型信息的指针。您可以将其视为"type"。
2. 数据指针。如果存储的数据是指针，则直接存储。如果存储的数据是一个值，则存储指向该值的指针。

如果希望接口方法修改基础数据，则必须使用指针传递 (将对象指针赋值给接口变量)。

```
type F interface {
  f()
}

type S1 struct{}
```

```
func (s S1) f() {}

type S2 struct{}

func (s *S2) f() {}

// f1.f() 无法修改底层数据
// f2.f() 可以修改底层数据，给接口变量 f2 赋值时使用的是对象指针
var f1 F = S1{}
var f2 F = &S2{}
```

永远不要使用指向 interface 的指针，这个是没有意义的。在 go 语言中，接口本身就是引用类型，换句话说，接口类型本身就是一个指针。对于我的需求，其实 test 的参数只要是 myinterface 就可以了，只需要在传值的时候，传*mystruct 类型（也只能传*mystruct 类型）

```
type myinterface interface{
    print()
}
func test(value *myinterface){
    //someting to do ...
}

type mystruct struct {
    i int
}
//实现接口
func (this *mystruct) print(){
    fmt.Println(this.i)
    this.i=1
}
func main(){
    m := &mystruct{0}
    test(m)  // 错误
    test(*m) // 错误
}
```

Interface 合理性验证

在编译时验证接口的符合性。这包括：

- 将实现特定接口的导出类型作为接口 API 的一部分进行检查
- 实现同一接口的 (导出和非导出) 类型属于实现类型的集合

- 任何违反接口合理性检查的场景，都会终止编译，并通知给用户

补充：上面 3 条是编译器对接口的检查机制，
大体意思是错误使用接口会在编译期报错。
所以可以利用这个机制让部分问题在编译期暴露。

Bad	Good

```
// 如果 Handler 没有实现 http.Handler，会在运行时报错
type Handler struct {
  // ...
}
func (h *Handler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  ...
}
```

```
type Handler struct {
  // ...
}
// 用于触发编译期的接口的合理性检查机制
// 如果 Handler 没有实现 http.Handler，会在编译期报错
var _ http.Handler = (*Handler)(nil)
func (h *Handler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  // ...
}
```

如果 `*Handler` 与 `http.Handler` 的接口不匹配，
那么语句 `var _ http.Handler = (*Handler)(nil)` 将无法编译通过。

赋值的右边应该是断言类型的零值。
对于指针类型（如 `*Handler`）、切片和映射，这是 `nil`；
对于结构类型，这是空结构。

```
type LogHandler struct {
  h   http.Handler
  log *zap.Logger
```

```
}
var _ http.Handler = LogHandler{}
func (h LogHandler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  // ...
}
```

接收器 (receiver) 与接口

使用值接收器的方法既可以通过值调用，也可以通过指针调用。

带指针接收器的方法只能通过指针或 addressable values 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"}}

// 你通过值只能调用 Read
sVals[1].Read()

// 这不能编译通过:
//  sVals[1].Write("test")

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

// 通过指针既可以调用 Read，也可以调用 Write 方法
sPtrs[1].Read()
sPtrs[1].Write("test")
```

类似的，即使方法有了值接收器，也同样可以用指针接收器来满足接口。

```
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

//  下面代码无法通过编译。因为 s2Val 是一个值，而 S2 的 f 方法中没有使用值接收器
//   i = s2Val
```

Effective Go 中有一段关于 pointers vs. values 的精彩讲解。

补充：

- 一个类型可以有值接收器方法集和指针接收器方法集
  - 值接收器方法集是指针接收器方法集的子集，反之不是
- 规则
  - 值对象只可以使用值接收器方法集
  - 指针对象可以使用 值接收器方法集 + 指针接收器方法集
- 接口的匹配 (或者叫实现)
  - 类型实现了接口的所有方法，叫匹配
  - 具体的讲，要么是类型的值方法集匹配接口，要么是指针方法集匹配接口

具体的匹配分两种：

- 值方法集和接口匹配
  - 给接口变量赋值的不管是值还是指针对象，都 ok，因为都包含值方法集
- 指针方法集和接口匹配

- 只能将指针对象赋值给接口变量，因为只有指针方法集和接口匹配
- 如果将值对象赋值给接口变量，会在编译期报错 (会触发接口合理性检查机制)

为啥 i = s2Val 会报错，因为值方法集和接口不匹配。

零值 Mutex 是有效的

零值 `sync.Mutex` 和 `sync.RWMutex` 是有效的。所以指向 mutex 的指针基本是不必要的。

Bad **Good**

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

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

如果你使用结构体指针，mutex 应该作为结构体的非指针字段。即使该结构体不被导出，也不要直接把 mutex 嵌入到结构体中。

Bad **Good**

```
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]
}

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]
}
```

`Mutex` 字段，`Lock` 和 `Unlock` 方法是 `SMap` 导出的 API 中不刻意说明的一部分。

mutex 及其方法是 `SMap` 的实现细节，对其调用者不可见。

在边界处拷贝 Slices 和 Maps

slices 和 maps 包含了指向底层数据的指针，因此在需要复制它们时要特别注意。

接收 Slices 和 Maps

请记住，当 map 或 slice 作为函数参数传入时，如果您存储了对它们的引用，则用户可以对其进行修改。

Bad	Good

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

trips := ...
```

```
d1.SetTrips(trips)

// 你是要修改 d1.trips 吗?
trips[0] = ...

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
trips[0] = ...
```

返回 slices 或 maps

同样，请注意用户对暴露内部状态的 map 或 slice 的修改。

Bad | **Good**

```
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
// 影响 stats.counters
snapshot := stats.Snapshot()

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 现在是一个拷贝
snapshot := stats.Snapshot()
```

使用 defer 释放资源

使用 defer 释放资源，诸如文件和锁。

Bad	Good

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

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

return newCount

// 当有多个 return 分支时，很容易遗忘 unlock

p.Lock()
defer p.Unlock()

if p.count < 10 {
```

```
    return p.count
}

p.count++
return p.count

// 更可读
```

Defer 的开销非常小，只有在您可以证明函数执行时间处于纳秒级的程度时，才应避免这样做。使用 defer 提升可读性是值得的，因为使用它们的成本微不足道。尤其适用于那些不仅仅是简单内存访问的较大的方法，在这些方法中其他计算的资源消耗远超过 `defer` 。

Channel 的 size 要么是 1，要么是无缓冲的

channel 通常 size 应为 1 或是无缓冲的。默认情况下，channel 是无缓冲的，其 size 为零。任何其他尺寸都必须经过严格的审查。我们需要考虑如何确定大小，考虑是什么阻止了 channel 在高负载下和阻塞写时的写入，以及当这种情况发生时系统逻辑有哪些变化。(翻译解释：按照原文意思是需要界定通道边界，竞态条件，以及逻辑上下文梳理)

Bad **Good**

```
// 应该足以满足任何情况!
c := make(chan int, 64)

// 大小：1
c := make(chan int, 1) // 或者
// 无缓冲 channel，大小为 0
c := make(chan int)
```

枚举从 1 开始

在 Go 中引入枚举的标准方法是声明一个自定义类型和一个使用了 iota 的 const 组。由于变量的默认值为 0，因此通常应以非零值开头枚举。

Bad **Good**

```
type Operation int

const (
  Add Operation = iota
  Subtract
  Multiply
)

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

type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

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

在某些情况下，使用零值是有意义的（枚举从零开始），例如，当零值是理想的默认行为时。

```
type LogOutput int

const (
  LogToStdout LogOutput = iota
  LogToFile
  LogToRemote
)

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

使用 time 处理时间

时间处理很复杂。关于时间的错误假设通常包括以下几点。

1. 一天有 24 小时
2. 一小时有 60 分钟
3. 一周有七天
4. 一年 365 天
5. 还有更多

例如，*1* 表示在一个时间点上加上 24 小时并不总是产生一个新的日历日。

因此，在处理时间时始终使用 `"time"` 包，因为它有助于以更安全、更准确的方式处理这些不正确的假设。

使用 `time.Time` 表达瞬时时间

在处理时间的瞬间时使用 `time.Time`，在比较、添加或减去时间时使用 `time.Time` 中的方法。

Bad **Good**

```
func isActive(now, start, stop int) bool {
  return start <= now && now < stop
}

func isActive(now, start, stop time.Time) bool {
  return (start.Before(now) || start.Equal(now)) && now.Before(stop)
}
```

使用 `time.Duration` 表达时间段

在处理时间段时使用 `time.Duration`.

Bad **Good**

```
func poll(delay int) {
  for {
    // ...
    time.Sleep(time.Duration(delay) * time.Millisecond)
  }
}
poll(10) // 是几秒钟还是几毫秒?

func poll(delay time.Duration) {
  for {
    // ...
    time.Sleep(delay)
  }
```

```
}
poll(10*time.Second)
```

回到第一个例子，在一个时间瞬间加上 24 小时，我们用于添加时间的方法取决于意图。如果我们想要下一个日历日 (当前天的下一天) 的同一个时间点，我们应该使用 `Time.AddDate` 。但是，如果我们想保证某一时刻比前一时刻晚 24 小时，我们应该使用 `Time.Add` 。

```
newDay := t.AddDate(0 /* years */, 0 /* months */, 1 /* days */)
maybeNewDay := t.Add(24 * time.Hour)
```

对外部系统使用 `time.Time` 和 `time.Duration`

尽可能在与外部系统的交互中使用 `time.Duration` 和 `time.Time` 例如：

- Command-line 标志： `flag` 通过 `time.ParseDuration` 支持 `time.Duration`
- JSON: `encoding/json` 通过其 `UnmarshalJSON` method 方法支持将 `time.Time` 编码为 RFC 3339 字符串
- SQL: `database/sql` 支持将 `DATETIME` 或 `TIMESTAMP` 列转换为 `time.Time` ，如果底层驱动程序支持则返回
- YAML: `gopkg.in/yaml.v2` 支持将 `time.Time` 作为 RFC 3339 字符串，并通过 `time.ParseDuration` 支持 `time.Duration` 。

当不能在这些交互中使用 `time.Duration` 时，请使用 `int` 或 `float64` ，并在字段名称中包含单位。

例如，由于 `encoding/json` 不支持 `time.Duration` ，因此该单位包含在字段的名称中。

Bad	Good

```
// {"interval": 2}
type Config struct {
  Interval int `json:"interval"`
}

// {"intervalMillis": 2000}
type Config struct {
```

```
    IntervalMillis int `json:"intervalMillis"`
}
```

当在这些交互中不能使用 `time.Time` 时，除非达成一致，否则使用 `string` 和 RFC 3339 中定义的格式时间戳。默认情况下， `Time.UnmarshalText` 使用此格式，并可通过 `time.RFC3339` 在 `Time.Format` 和 `time.Parse` 中使用。

尽管这在实践中并不成问题，但请记住， `"time"` 包不支持解析闰秒时间戳（8728），也不在计算中考虑闰秒（15190）。如果您比较两个时间瞬间，则差异将不包括这两个瞬间之间可能发生的闰秒。

Errors

错误类型

声明错误的选项很少。
在选择最适合您的用例的选项之前，请考虑以下事项。

- 调用者是否需要匹配错误以便他们可以处理它?
  如果是，我们必须通过声明顶级错误变量或自定义类型来支持 `errors.Is` 或 `errors.As` 函数。
- 错误消息是否为静态字符串，还是需要上下文信息的动态字符串?
  如果是静态字符串，我们可以使用 `errors.New` ，但对于后者，我们必须使用 `fmt.Errorf` 或自定义错误类型。
- 我们是否正在传递由下游函数返回的新错误?
  如果是这样，请参阅错误包装部分。

错误匹配?	错误消息	指导
No	static	`errors.New`
No	dynamic	`fmt.Errorf`
Yes	static	top-level `var` with `errors.New`
Yes	dynamic	custom `error` type

例如，
使用 `errors.New` 表示带有静态字符串的错误。
如果调用者需要匹配并处理此错误，则将此错误导出为变量以支持将其与 `errors.Is` 匹配。

无错误匹配 **错误匹配**

```
// package foo

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

// package bar

if err := foo.Open(); err != nil {
  // Can't handle the error.
  panic("unknown error")
}
```

```
// package foo

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

func Open() error {
  return ErrCouldNotOpen
}

// package bar

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

对于动态字符串的错误，
如果调用者不需要匹配它，则使用 fmt.Errorf，
如果调用者确实需要匹配它，则自定义 `error` 。

无错误匹配 **错误匹配**

```
// package foo

func Open(file string) error {
```

```
  return fmt.Errorf("file %q not found", file)
}

// package bar

if err := foo.Open("testfile.txt"); err != nil {
  // Can't handle the error.
  panic("unknown error")
}


// package foo

type NotFoundError struct {
  File string
}

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

func Open(file string) error {
  return &NotFoundError{File: file}
}



// package bar

if err := foo.Open("testfile.txt"); err != nil {
  var notFound *NotFoundError
  if errors.As(err, &notFound) {
    // handle the error
  } else {
    panic("unknown error")
  }
}
```

请注意，如果您从包中导出错误变量或类型，
它们将成为包的公共 API 的一部分。

错误包装

如果调用其他方法时出现错误，通常有三种处理方式可以选择：

- 将原始错误原样返回

- 使用 `fmt.Errorf` 搭配 `%w` 将错误添加进上下文后返回
- 使用 `fmt.Errorf` 搭配 `%v` 将错误添加进上下文后返回

如果没有要添加的其他上下文，则按原样返回原始错误。
这将保留原始错误类型和消息。
这非常适合底层错误消息有足够的信息来追踪它来自哪里的错误。

否则，尽可能在错误消息中添加上下文
这样就不会出现诸如“连接被拒绝”之类的模糊错误，
您会收到更多有用的错误，例如“调用服务 foo：连接被拒绝”。

使用 `fmt.Errorf` 为你的错误添加上下文，
根据调用者是否应该能够匹配和提取根本原因，在 `%w` 或 `%v` 动词之间进行选择。

- 如果调用者应该可以访问底层错误，请使用 `%w` 。
  对于大多数包装错误，这是一个很好的默认值，
  但请注意，调用者可能会开始依赖此行为。因此，对于包装错误是已知 `var` 或类型的情况，请将其作为函数契约的一部分进行记录和测试。
- 使用 `%v` 来混淆底层错误。
  调用者将无法匹配它，但如果需要，您可以在将来切换到 `%w` 。

在为返回的错误添加上下文时，通过避免使用"failed to"之类的短语来保持上下文简洁，
当错误通过堆栈向上渗透时，它会一层一层被堆积起来：

Bad	Good

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

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

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

```
x: y: new store: the error
```

然而，一旦错误被发送到另一个系统，应该清楚消息是一个错误（例如 `err` 标签或日志中的"Failed"前缀）。

另见 [不要只检查错误，优雅地处理它们]。

错误命名

对于存储为全局变量的错误值，
根据是否导出，使用前缀 `Err` 或 `err` 。
请看指南 [对于未导出的顶层常量和变量，使用_作为前缀]。

```
var (
  // 导出以下两个错误，以便此包的用户可以将它们与 errors.Is 进行匹配。

  ErrBrokenLink = errors.New("link is broken")
  ErrCouldNotOpen = errors.New("could not open")

  // 这个错误没有被导出，因为我们不想让它成为我们公共 API 的一部分。我们可能仍然在带有错

  errNotFound = errors.New("not found")
)
```

对于自定义错误类型，请改用后缀 `Error` 。

```
// 同样，这个错误被导出，以便这个包的用户可以将它与 errors.As 匹配。

type NotFoundError struct {
  File string
}

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

// 并且这个错误没有被导出，因为我们不想让它成为公共 API 的一部分。我们仍然可以在带有 err
type resolveError struct {
  Path string
}

func (e *resolveError) Error() string {
  return fmt.Sprintf("resolve %q", e.Path)
}
```

一次处理错误

当调用方从被调用方接收到错误时，它可以根据对错误的了解，以各种不同的方式进行处理。

其中包括但不限于：

- 如果被调用者约定定义了特定的错误，则将错误与 `errors.Is` 或 `errors.As` 匹配，并以不同的方式处理分支
- 如果错误是可恢复的，则记录错误并正常降级
- 如果该错误表示特定于域的故障条件，则返回定义明确的错误
- 返回错误，无论是 wrapped 还是逐字逐句

无论调用方如何处理错误，它通常都应该只处理每个错误一次。例如，调用方不应该记录错误然后返回，因为*its*调用方也可能处理错误。

例如，考虑以下情况：

Description	Code

Bad: 记录错误并将其返回

堆栈中的调用程序可能会对该错误采取类似的操作。这样做会在应用程序日志中造成大量噪音，但收效甚微。

```
u, err := getUser(id)
if err != nil {
  // BAD: See description
  log.Printf("Could not get user %q: %v", id, err)
  return err
}
```

Good: 将错误换行并返回

堆栈中更靠上的调用程序将处理该错误。使用 `%w` 可确保它们可以将错误与 `errors.Is` 或 `errors.As` 相匹配（如果相关）。

```
u, err := getUser(id)
if err != nil {
  return fmt.Errorf("get user %q: %w", id, err)
}
```

Good: 记录错误并正常降级

如果操作不是绝对必要的，我们可以通过从中恢复来提供降级但不间断的体验。

```
if err := emitMetrics(); err != nil {
  // Failure to write metrics should not
  // break the application.
  log.Printf("Could not emit metrics: %v", err)
}
```

Good: 匹配错误并适当降级

如果被调用者在其约定中定义了一个特定的错误，并且失败是可恢复的，则匹配该错误案例并正常降级。对于所有其他案例，请包装错误并返回。

堆栈中更靠上的调用程序将处理其他错误。

```
tz, err := getUserTimeZone(id)
if err != nil {
  if errors.Is(err, ErrUserNotFound) {
    // User doesn't exist. Use UTC.
    tz = time.UTC
  } else {
    return fmt.Errorf("get user %q: %w", id, err)
  }
}
```

处理断言失败

类型断言 将会在检测到不正确的类型时，以单一返回值形式返回 panic。因此，请始终使用“逗号 ok”习语。

Bad	Good

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

```
t, ok := i.(string)
if !ok {
  // 优雅地处理错误
}
```

不要使用 panic

在生产环境中运行的代码必须避免出现 panic。panic 是 级联失败 的主要根源。如果发生错误，该函数必须返回错误，并允许调用方决定如何处理它。

Bad	Good

```
func run(args []string) {
  if len(args) == 0 {
    panic("an argument is required")
  }
  // ...
}

func main() {
  run(os.Args[1:])
}
```

```
func run(args []string) error {
  if len(args) == 0 {
    return errors.New("an argument is required")
  }
  // ...
  return nil
}

func main() {
  if err := run(os.Args[1:]); err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
  }
}
```

panic/recover 不是错误处理策略。仅当发生不可恢复的事情（例如：nil 引用）时，程序才必须 panic。程序初始化是一个例外：程序启动时应使程序中止的不良情况可能会引起 panic。

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

即使在测试代码中，也优先使用 `t.Fatal` 或者 `t.FailNow` 而不是 panic 来确保失败被标记。

Bad **Good**

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

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

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

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

使用 go.uber.org/atomic

使用 sync/atomic 包的原子操作对原始类型（`int32`，`int64` 等）进行操作，因为很容易忘记使用原子操作来读取或修改变量。

go.uber.org/atomic 通过隐藏基础类型为这些操作增加了类型安全性。此外，它包括一个方便的 `atomic.Bool` 类型。

Bad **Good**

```
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!
}

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()
}
```

避免可变全局变量

使用选择依赖注入方式避免改变全局变量。
既适用于函数指针又适用于其他值类型

Bad	Good

```
// sign.go
var _timeNow = time.Now
func sign(msg string) string {
  now := _timeNow()
  return signWithTime(msg, now)
}

// sign.go
type signer struct {
  now func() time.Time
}
func newSigner() *signer {
  return &signer{
    now: time.Now,
  }
}
```

```
func (s *signer) Sign(msg string) string {
  now := s.now()
  return signWithTime(msg, now)
}

// sign_test.go
func TestSign(t *testing.T) {
  oldTimeNow := _timeNow
  _timeNow = func() time.Time {
    return someFixedTime
  }
  defer func() { _timeNow = oldTimeNow }()
  assert.Equal(t, want, sign(give))
}

// sign_test.go
func TestSigner(t *testing.T) {
  s := newSigner()
  s.now = func() time.Time {
    return someFixedTime
  }
  assert.Equal(t, want, s.Sign(give))
}
```

避免在公共结构中嵌入类型

这些嵌入的类型泄漏实现细节、禁止类型演化和模糊的文档。

假设您使用共享的 `AbstractList` 实现了多种列表类型，请避免在具体的列表实现中嵌入 `AbstractList` 。
相反，只需手动将方法写入具体的列表，该列表将委托给抽象列表。

```
type AbstractList struct {}
// 添加将实体添加到列表中。
func (l *AbstractList) Add(e Entity) {
  // ...
}
// 移除从列表中移除实体。
func (l *AbstractList) Remove(e Entity) {
  // ...
}
```

Bad **Good**

```
// ConcreteList 是一个实体列表。
type ConcreteList struct {
  *AbstractList
}
```

```
// ConcreteList 是一个实体列表。
type ConcreteList struct {
  list *AbstractList
}
// 添加将实体添加到列表中。
func (l *ConcreteList) Add(e Entity) {
  l.list.Add(e)
}
// 移除从列表中移除实体。
func (l *ConcreteList) Remove(e Entity) {
  l.list.Remove(e)
}
```

Go 允许 类型嵌入 作为继承和组合之间的折衷。外部类型获取嵌入类型的方法的隐式副本。默认情况下，这些方法委托给嵌入实例的同一方法。

结构还获得与类型同名的字段。
所以，如果嵌入的类型是 public，那么字段是 public。为了保持向后兼容性，外部类型的每个未来版本都必须保留嵌入类型。

很少需要嵌入类型。
这是一种方便，可以帮助您避免编写冗长的委托方法。

即使嵌入兼容的抽象列表 *interface*，而不是结构体，这将为开发人员提供更大的灵活性来改变未来，但仍然泄露了具体列表使用抽象实现的细节。

Bad **Good**

```
// AbstractList 是各种实体列表的通用实现。
type AbstractList interface {
  Add(Entity)
  Remove(Entity)
}
// ConcreteList 是一个实体列表。
```

```
type ConcreteList struct {
  AbstractList
}

// ConcreteList 是一个实体列表。
type ConcreteList struct {
  list AbstractList
}
// 添加将实体添加到列表中。
func (l *ConcreteList) Add(e Entity) {
  l.list.Add(e)
}
// 移除从列表中移除实体。
func (l *ConcreteList) Remove(e Entity) {
  l.list.Remove(e)
}
```

无论是使用嵌入结构还是嵌入接口，都会限制类型的演化。

- 向嵌入接口添加方法是一个破坏性的改变。
- 从嵌入结构体删除方法是一个破坏性改变。
- 删除嵌入类型是一个破坏性的改变。
- 即使使用满足相同接口的类型替换嵌入类型，也是一个破坏性的改变。

尽管编写这些委托方法是乏味的，但是额外的工作隐藏了实现细节，留下了更多的更改机会，还消除了在文档中发现完整列表接口的间接性操作。

避免使用内置名称

Go 语言规范 概述了几个内置的，
不应在 Go 项目中使用的 预先声明的标识符。

根据上下文的不同，将这些标识符作为名称重复使用，
将在当前作用域（或任何嵌套作用域）中隐藏原始标识符，或者混淆代码。
在最好的情况下，编译器会报错；在最坏的情况下，这样的代码可能会引入潜在的、难以恢复的错误。

Bad	Good

```
var error string
// `error` 作用域隐式覆盖
```

```
// or

func handleErrorMessage(error string) {
    // `error` 作用域隐式覆盖
}

var errorMessage string
// `error` 指向内置的非覆盖

// or

func handleErrorMessage(msg string) {
    // `error` 指向内置的非覆盖
}

type Foo struct {
    // 虽然这些字段在技术上不构成阴影，但`error`或`string`字符串的重映射现在是不明确的
    error  error
    string string
}

func (f Foo) Error() error {
    // `error` 和 `f.error` 在视觉上是相似的
    return f.error
}

func (f Foo) String() string {
    // `string` and `f.string` 在视觉上是相似的
    return f.string
}


type Foo struct {
    // `error` and `string` 现在是明确的。
    err error
    str string
}

func (f Foo) Error() error {
    return f.err
}

func (f Foo) String() string {
```

```
    return f.str
}
```

注意，编译器在使用预先分隔的标识符时不会生成错误，
但是诸如 `go vet` 之类的工具会正确地指出这些和其他情况下的隐式问题。

避免使用 `init()`

尽可能避免使用 `init()` 。当 `init()` 是不可避免或可取的，代码应先尝试：

1. 无论程序环境或调用如何，都要完全确定。
2. 避免依赖于其他 `init()` 函数的顺序或副作用。虽然 `init()` 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和容易出错。
3. 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。
4. 避免 `I/O` ，包括文件系统、网络和系统调用。

不能满足这些要求的代码可能属于要作为 `main()` 调用的一部分（或程序生命周期中的其他地方），
或者作为 `main()` 本身的一部分写入。特别是，打算由其他程序使用的库应该特别注意完全确定性，
而不是执行“init magic”

Bad	Good

```
type Foo struct {
    // ...
}
var _defaultFoo Foo
func init() {
    _defaultFoo = Foo{
        // ...
    }
}

var _defaultFoo = Foo{
    // ...
}
// or，为了更好的可测试性：
```

```
var _defaultFoo = defaultFoo()
func defaultFoo() Foo {
    return Foo{
        // ...
    }
}

type Config struct {
    // ...
}
var _config Config
func init() {
    // Bad: 基于当前目录
    cwd, _ := os.Getwd()
    // Bad: I/O
    raw, _ := os.ReadFile(
        path.Join(cwd, "config", "config.yaml"),
    )
    yaml.Unmarshal(raw, &_config)
}

type Config struct {
    // ...
}
func loadConfig() Config {
    cwd, err := os.Getwd()
    // handle err
    raw, err := os.ReadFile(
        path.Join(cwd, "config", "config.yaml"),
    )
    // handle err
    var config Config
    yaml.Unmarshal(raw, &config)
    return config
}
```

考虑到上述情况，在某些情况下， `init()` 可能更可取或是必要的，可能包括：

- 不能表示为单个赋值的复杂表达式。
- 可插入的钩子，如 `database/sql` 、编码类型注册表等。
- 对 Google Cloud Functions 和其他形式的确定性预计算的优化。

追加时优先指定切片容量

追加时优先指定切片容量

在尽可能的情况下，在初始化要追加的切片时为 `make()` 提供一个容量值。

Bad **Good**

```
for n := 0; n < b.N; n++ {
  data := make([]int, 0)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
```

```
for n := 0; n < b.N; n++ {
  data := make([]int, 0, size)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
```

```
BenchmarkBad-4    100000000    2.48s
```

```
BenchmarkGood-4   100000000    0.21s
```

主函数退出方式 (Exit)

Go 程序使用 `os.Exit` 或者 `log.Fatal*` 立即退出 (使用 `panic` 不是退出程序的好方法，请 不要使用 panic。)

仅在 `main()` 中调用其中一个 `os.Exit` 或者 `log.Fatal*` 。所有其他函数应将错误返回到信号失败中。

Bad **Good**

```
func main() {
  body := readFile(path)
  fmt.Println(body)
```

```
}
func readFile(path string) string {
  f, err := os.Open(path)
  if err != nil {
    log.Fatal(err)
  }
  b, err := os.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
  return string(b)
}

func main() {
  body, err := readFile(path)
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(body)
}
func readFile(path string) (string, error) {
  f, err := os.Open(path)
  if err != nil {
    return "", err
  }
  b, err := os.ReadAll(f)
  if err != nil {
    return "", err
  }
  return string(b), nil
}
```

原则上：退出的具有多种功能的程序存在一些问题：

- 不明显的控制流：任何函数都可以退出程序，因此很难对控制流进行推理。
- 难以测试：退出程序的函数也将退出调用它的测试。这使得函数很难测试，并引入了跳过 `go test` 尚未运行的其他测试的风险。
- 跳过清理：当函数退出程序时，会跳过已经进入 `defer` 队列里的函数调用。这增加了跳过重要清理任务的风险。

一次性退出

如果可能的话，你的 `main()` 函数中 **最多一次** 调用 `os.Exit` 或者 `log.Fatal` 。如果有多个错误场景停止程序执行，请将该逻辑放在单独的函数下并从中返回错误。

这会缩短 `main()` 函数，并将所有关键业务逻辑放入一个单独的、可测试的函数中。

Bad | **Good**

```
package main
func main() {
  args := os.Args[1:]
  if len(args) != 1 {
    log.Fatal("missing file")
  }
  name := args[0]
  f, err := os.Open(name)
  if err != nil {
    log.Fatal(err)
  }
  defer f.Close()
  // 如果我们调用 log.Fatal 在这条线之后
  // f.Close 将会被执行。
  b, err := os.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
  // ...
}
```

```
package main
func main() {
  if err := run(); err != nil {
    log.Fatal(err)
  }
}
func run() error {
  args := os.Args[1:]
  if len(args) != 1 {
    return errors.New("missing file")
  }
  name := args[0]
  f, err := os.Open(name)
  if err != nil {
    return err
  }
  defer f.Close()
  b, err := os.ReadAll(f)
  if err != nil {
    return err
```

```
  }
  // ...
}
```

上面的示例使用 `log.Fatal`，但该指南也适用于 `os.Exit` 或任何调用 `os.Exit` 的库代码。

```
func main() {
  if err := run(); err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
  }
}
```

您可以根据需要更改 `run()` 的签名。例如，如果您的程序必须使用特定的失败退出代码退出， `run()` 可能会返回退出代码而不是错误。这也允许单元测试直接验证此行为。

```
func main() {
  os.Exit(run(args))
}

func run() (exitCode int) {
  // ...
}
```

请注意，这些示例中使用的 `run()` 函数并不是强制性的。
`run()` 函数的名称、签名和设置具有灵活性。除其他外，您可以：

- 接受未分析的命令行参数 (e.g., `run(os.Args[1:])`)
- 解析 `main()` 中的命令行参数并将其传递到 `run`
- 使用自定义错误类型将退出代码传回 `main ()`
- 将业务逻辑置于不同的抽象层 `package main`

本指南只要求在 `main()` 中有一个位置负责实际的退出流程。

在序列化结构中使用字段标记

任何序列化到 JSON、YAML、,
或其他支持基于标记的字段命名的格式应使用相关标记进行注释。

Bad	Good

```
type Stock struct {
  Price int
  Name  string
}
bytes, err := json.Marshal(Stock{
  Price: 137,
  Name:  "UBER",
})
```

```
type Stock struct {
  Price int    `json:"price"`
  Name  string `json:"name"`
  // Safe to rename Name to Symbol.
}
bytes, err := json.Marshal(Stock{
  Price: 137,
  Name:  "UBER",
})
```

理论上：
结构的序列化形式是不同系统之间的契约。
对序列化表单结构（包括字段名）的更改会破坏此约定。在标记中指定字段名使约定明确，
它还可以通过重构或重命名字段来防止意外违反约定。

不要一劳永逸地使用 goroutine

Goroutines 是轻量级的，但它们不是免费的：
至少，它们会为堆栈和 CPU 的调度消耗内存。
虽然这些成本对于 Goroutines 的使用来说很小，但当它们在没有受控生命周期的情况下大量生成时会导致严重的性能问题。
具有非托管生命周期的 Goroutines 也可能导致其他问题，例如防止未使用的对象被垃圾回收并保留不再使用的资源。

因此，不要在代码中泄漏 goroutine。
使用 go.uber.org/goleak
来测试可能产生 goroutine 的包内的 goroutine 泄漏。

一般来说，每个 goroutine:

- 必须有一个可预测的停止运行时间；或者
- 必须有一种方法可以向 goroutine 发出信号它应该停止

在这两种情况下，都必须有一种方式代码来阻塞并等待 goroutine 完成。

For example:

Bad	Good

```
go func() {
  for {
    flush()
    time.Sleep(delay)
  }
}()
```

```
var (
  stop = make(chan struct{}) // 告诉 goroutine 停止
  done = make(chan struct{}) // 告诉我们 goroutine 退出了
)
go func() {
  defer close(done)
  ticker := time.NewTicker(delay)
  defer ticker.Stop()
  for {
    select {
    case <-tick.C:
      flush()
    case <-stop:
      return
    }
  }
}()
// 其它...
close(stop)  // 指示 goroutine 停止
<-done       // and wait for it to exit
```

没有办法阻止这个 goroutine。这将一直运行到应用程序退出。

这个 goroutine 可以用 `close(stop)`，
我们可以等待它退出 `<-done`.

等待 goroutines 退出

给定一个由系统生成的 goroutine，
必须有一种方案能等待 goroutine 的退出。
有两种常用的方法可以做到这一点：

- 使用 `sync.WaitGroup`。
  如果您要等待多个 goroutine，请执行此操作

```
var wg sync.WaitGroup
for i := 0; i < N; i++ {
  wg.Add(1)
  go func() {
    defer wg.Done()
    // ...
  }()
}

// To wait for all to finish:
wg.Wait()
```

- 添加另一个 `chan struct{}`，goroutine 完成后会关闭它。
  如果只有一个 goroutine，请执行此操作。

```
done := make(chan struct{})
go func() {
  defer close(done)
  // ...
}()

// To wait for the goroutine to finish:
<-done
```

不要在 `init()` 使用 goroutines

`init()` 函数不应该产生 goroutines。
另请参阅 避免使用 init()。

如果一个包需要一个后台 goroutine，
它必须公开一个负责管理 goroutine 生命周期的对象。
该对象必须提供一个方法（`Close`、`Stop`、`Shutdown` 等）来指示后台 goroutine 停止并等待它的退出。

Bad	Good

```
func init() {
  go doWork()
}
func doWork() {
  for {
    // ...
  }
}
```

```
type Worker struct{ /* ... */ }
func NewWorker(...) *Worker {
  w := &Worker{
    stop: make(chan struct{}),
    done: make(chan struct{}),
    // ...
  }
  go w.doWork()
}
func (w *Worker) doWork() {
  defer close(w.done)
  for {
    // ...
    case <-w.stop:
      return
  }
}
// Shutdown 告诉 worker 停止
// 并等待它完成。
func (w *Worker) Shutdown() {
  close(w.stop)
  <-w.done
}
```

当用户导出这个包时，无条件地生成一个后台 goroutine。
用户无法控制 goroutine 或停止它的方法。

仅当用户请求时才生成工作人员。
提供一种关闭工作器的方法，以便用户可以释放工作器使用的资源。

请注意，如果工作人员管理多个 goroutine，则应使用 `WaitGroup` 。
请参阅 等待 goroutines 退出。

性能

性能方面的特定准则只适用于高频场景。

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 **Good**

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

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

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

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

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。相反，请执行一次转换并捕获结果。

Bad **Good**

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

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

```
BenchmarkBad-4   50000000   22.2 ns/op

BenchmarkGood-4  500000000   3.25 ns/op
```

指定容器容量

尽可能指定容器容量，以便为容器预先分配内存。这将在添加元素时最小化后续分配（通过复制和调整容器大小）。

指定 Map 容量提示

在尽可能的情况下，在使用 `make()` 初始化的时候提供容量信息

```
make(map[T1]T2, hint)
```

向 `make()` 提供容量提示会在初始化时尝试调整 map 的大小，这将减少在将元素添加到 map 时为 map 重新分配内存。

注意，与 slices 不同。map 容量提示并不保证完全的、预先的分配，而是用于估计所需的 hashmap bucket 的数量。
因此，在将元素添加到 map 时，甚至在指定 map 容量时，仍可能发生分配。

Bad | **Good**

```
m := make(map[string]os.FileInfo)

files, _ := os.ReadDir("./files")
for _, f := range files {
    m[f.Name()] = f
}
```

```
files, _ := os.ReadDir("./files")

m := make(map[string]os.FileInfo, len(files))
for _, f := range files {
    m[f.Name()] = f
}
```

`m` 是在没有大小提示的情况下创建的；在运行时可能会有更多分配。

m 是有大小提示创建的；在运行时可能会有更少的分配。

指定切片容量

在尽可能的情况下，在使用 make() 初始化切片时提供容量信息，特别是在追加切片时。

```
make([]T, length, capacity)
```

与 maps 不同，slice capacity 不是一个提示：编译器将为提供给 make() 的 slice 的容量分配足够的内存，
这意味着后续的 append() 操作将导致零分配（直到 slice 的长度与容量匹配，在此之后，任何 append 都可能调整大小以容纳其他元素）。

Bad	Good

```
for n := 0; n < b.N; n++ {
  data := make([]int, 0)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
```

```
for n := 0; n < b.N; n++ {
  data := make([]int, 0, size)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
```

```
BenchmarkBad-4    100000000    2.48s
```

```
BenchmarkGood-4   100000000    0.21s
```

规范

避免过长的行

避免使用需要读者水平滚动或过度转动头部的代码行。

我们建议将行长度限制为 **99 characters** (99 个字符).
作者应该在达到这个限制之前换行，
但这不是硬性限制。
允许代码超过此限制。

一致性

本文中概述的一些标准都是客观性的评估，是根据场景、上下文、或者主观性的判断；

但是最重要的是，**保持一致**.

一致性的代码更容易维护、是更合理的、需要更少的学习成本、并且随着新的约定出现或者出现错误后更容易迁移、更新、修复 bug

相反，在一个代码库中包含多个完全不同或冲突的代码风格会导致维护成本开销、不确定性和认知偏差。所有这些都会直接导致速度降低、代码审查痛苦、而且增加 bug 数量。

将这些标准应用于代码库时，建议在 package（或更大）级别进行更改，子包级别的应用程序通过将多个样式引入到同一代码中，违反了上述关注点。

相似的声明放在一组

Go 语言支持将相似的声明放在一个组内。

Bad	Good

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

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

这同样适用于常量、变量和类型声明：

Bad	Good

```
const a = 1
const b = 2

var a = 1
var b = 2

type Area float64
type Volume float64

const (
  a = 1
  b = 2
)

var (
  a = 1
  b = 2
)

type (
  Area float64
  Volume float64
)
```

仅将相关的声明放在一组。不要将不相关的声明放在一组。

Bad	Good

```
type Operation int

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

type Operation int

const (
  Add Operation = iota + 1
  Subtract
```

```
  Multiply
)

const EnvVar = "MY_ENV"
```

分组使用的位置没有限制，例如：你可以在函数内部使用它们：

Bad	Good

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

  ...
}

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

  ...
}
```

例外：如果变量声明与其他变量相邻，则应将变量声明（尤其是函数内部的声明）分组在一起。对一起声明的变量执行此操作，即使它们不相关。

Bad	Good

```
func (c *client) request() {
  caller := c.name
  format := "json"
  timeout := 5*time.Second
  var err error
  // ...
}
```

```
func (c *client) request() {
  var (
    caller  = c.name
    format  = "json"
    timeout = 5*time.Second
    err error
  )
  // ...
}
```

import 分组

导入应该分为两组：

- 标准库
- 其他库

默认情况下，这是 goimports 应用的分组。

Bad	Good

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

```
import (
  "fmt"
  "os"

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

包名

当命名包时，请按下面规则选择一个名称：

- 全部小写。没有大写或下划线。
- 大多数使用命名导入的情况下，不需要重命名。
- 简短而简洁。请记住，在每个使用的地方都完整标识了该名称。
- 不用复数。例如 `net/url`，而不是 `net/urls`。
- 不要用“common”，“util”，“shared”或“lib”。这些是不好的，信息量不足的名称。

另请参阅 Go 包命名规则 和 Go 包样式指南.

函数名

我们遵循 Go 社区关于使用 MixedCaps 作为函数名 的约定。有一个例外，为了对相关的测试用例进行分组，函数名可能包含下划线，如：
`TestMyFunction_WhatIsBeingTested`.

导入别名

如果程序包名称与导入路径的最后一个元素不匹配，则必须使用导入别名。

```
import (
  "net/http"

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

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。

Bad	Good

```
import (
  "fmt"
  "os"

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

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

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

函数分组与顺序

- 函数应按粗略的调用顺序排序。
- 同一文件中的函数应按接收者分组。

因此，导出的函数应先出现在文件中，放在 `struct`，`const`，`var` 定义的后面。

在定义类型之后，但在接收者的其余方法之前，可能会出现一个 `newXYZ()` / `NewXYZ()`

由于函数是按接收者分组的，因此普通工具函数应在文件末尾出现。

Bad	Good

```
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{}
}
```

```
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 {...}
```

减少嵌套

代码应通过尽可能先处理错误情况/特殊情况并尽早返回或继续循环来减少嵌套。减少嵌套多个级别的代码的代码量。

Bad	Good

```
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)
  }
}
```

```
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()
}
```

不必要的 else

如果在 if 的两个分支中都设置了变量，则可以将其替换为单个 if。

Bad	Good

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

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

顶层变量声明

在顶层，使用标准 var 关键字。请勿指定类型，除非它与表达式的类型不同。

Bad	Good

```
var _s string = F()

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

```
var _s = F()
// 由于 F 已经明确了返回一个字符串类型，因此我们没有必要显式指定_s 的类型
// 还是那种类型

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

如果表达式的类型与所需的类型不完全匹配，请指定类型。

```
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
```

对于未导出的顶层常量和变量，使用_作为前缀

在未导出的顶级 vars 和 consts ， 前面加上前缀_，以使它们在使用时明确表示它们是全局符号。

基本依据：顶级变量和常量具有包范围作用域。使用通用名称可能很容易在其他文件中意外使用错误的值。

Bad	Good

```
// 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.
}
```

```
// foo.go

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

例外：未导出的错误值可以使用不带下划线的前缀 err 。参见错误命名。

结构体中的嵌入

嵌入式类型（例如 mutex）应位于结构体内的字段列表的顶部，并且必须有一个空行将嵌入式字段与常规字段分隔开。

Bad **Good**

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

type Client struct {
  http.Client

  version int
}
```

内嵌应该提供切实的好处，比如以语义上合适的方式添加或增强功能。
它应该在对用户没有任何不利影响的情况下使用。(另请参见：避免在公共结构中嵌入类型)。

例外：即使在未导出类型中，Mutex 也不应该作为内嵌字段。另请参见：零值 Mutex 是有效的。

嵌入 **不应该**:

- 纯粹是为了美观或方便。
- 使外部类型更难构造或使用。
- 影响外部类型的零值。如果外部类型有一个有用的零值，则在嵌入内部类型之后应该仍然有一个有用的零值。
- 作为嵌入内部类型的副作用，从外部类型公开不相关的函数或字段。
- 公开未导出的类型。
- 影响外部类型的复制形式。
- 更改外部类型的 API 或类型语义。
- 嵌入内部类型的非规范形式。
- 公开外部类型的实现详细信息。
- 允许用户观察或控制类型内部。
- 通过包装的方式改变内部函数的一般行为，这种包装方式会给用户带来一些意料之外情况。

简单地说，有意识地和有目的地嵌入。一种很好的测试体验是，
“是否所有这些导出的内部方法/字段都将直接添加到外部类型”
如果答案是 some 或 no ， 不要嵌入内部类型 - 而是使用字段。

Bad **Good**

```
type A struct {
    // Bad: A.Lock() and A.Unlock() 现在可用
    // 不提供任何功能性好处，并允许用户控制有关 A 的内部细节。
    sync.Mutex
}

type countingWriteCloser struct {
    // Good: Write() 在外层提供用于特定目的，
    // 并且委托工作到内部类型的 Write() 中。
    io.WriteCloser
    count int
}
func (w *countingWriteCloser) Write(bs []byte) (int, error) {
    w.count += len(bs)
    return w.WriteCloser.Write(bs)
}

type Book struct {
    // Bad: 指针更改零值的有用性
    io.ReadWriter
    // other fields
}
// later
var b Book
b.Read(...)  // panic: nil pointer
b.String()   // panic: nil pointer
b.Write(...) // panic: nil pointer

type Book struct {
    // Good: 有用的零值
    bytes.Buffer
    // other fields
}
// later
var b Book
b.Read(...)  // ok
b.String()   // ok
b.Write(...) // ok

type Client struct {
    sync.Mutex
```

```
    sync.WaitGroup
    bytes.Buffer
    url.URL
}

type Client struct {
    mtx sync.Mutex
    wg  sync.WaitGroup
    buf bytes.Buffer
    url url.URL
}
```

本地变量声明

如果将变量明确设置为某个值，则应使用短变量声明形式（:=）。

Bad	Good

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

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

但是，在某些情况下， var 使用关键字时默认值会更清晰。例如，声明空切片。

Bad	Good

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

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

```
        }
    }
}
```

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，

- 您不应明确返回长度为零的切片。应该返回 `nil` 来代替。

Bad	Good

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

- 要检查切片是否为空，请始终使用 `len(s) == 0` 。而非 `nil` 。

Bad	Good

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

- 零值切片（用 `var` 声明的切片）可立即使用，无需调用 `make()` 创建。

Bad	Good

nums := []int{}
// or, nums := make([]int)
 	
if add1 {
  nums = append(nums, 1)
}
 	
if add2 {
  nums = append(nums, 2)
}
	var nums []int
	
	if add1 {
	  nums = append(nums, 1)
	}
	
	if add2 {
	  nums = append(nums, 2)
	}

记住，虽然 nil 切片是有效的切片，但它不等于长度为 0 的切片（一个为 nil，另一个不是），并且在不同的情况下（例如序列化），这两个切片的处理方式可能不同。

缩小变量作用域

如果有可能，尽量缩小变量作用范围。除非它与 减少嵌套的规则冲突。

Bad

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

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

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。

Bad

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

```
    return err
  }

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

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

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

fmt.Println(cfg)
return nil
```

避免参数语义不明确 (Avoid Naked Parameters)

函数调用中的 意义不明确的参数 可能会损害可读性。当参数名称的含义不明显时，请为参数添加 C 样式注释 (`/* ... */`)

Bad	Good

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

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

```
// 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。将来，该参数可以支持不仅仅局限于两个状态（true/false）。

```
type Region int
```

```
const (
  UnknownRegion Region = iota
  Local
)

type Status int

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

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

使用原始字符串字面值，避免转义

Go 支持使用 原始字符串字面值，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。

可以跨越多行并包含引号。使用这些字符串可以避免更难阅读的手工转义的字符串。

Bad	Good

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

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

初始化结构体

使用字段名初始化结构

初始化结构时，几乎应该始终指定字段名。目前由 `go vet` 强制执行。

Bad	Good

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

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

例外：当有 3 个或更少的字段时，测试表中的字段名可以省略。

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

省略结构中的零值字段

初始化具有字段名的结构时，除非提供有意义的上下文，否则忽略值为零的字段。
也就是，让我们自动将这些设置为零值

Bad	Good

```
user := User{
  FirstName: "John",
  LastName: "Doe",
  MiddleName: "",
  Admin: false,
}
```

```
user := User{
  FirstName: "John",
  LastName: "Doe",
}
```

这有助于通过省略该上下文中的默认值来减少阅读的障碍。只指定有意义的值。

在字段名提供有意义上下文的地方包含零值。例如，表驱动测试 中的测试用例可以受益于字段的名称，即使它们是零值的。

```
tests := []struct{
  give string
  want int
}{
  {give: "0", want: 0},
  // ...
}
```

对零值结构使用 `var`

如果在声明中省略了结构的所有字段，请使用 `var` 声明结构。

Bad	Good

```
user := User{}
```

```
var user User
```

这将零值结构与那些具有类似于为 初始化 Maps 创建的，区别于非零值字段的结构区分开来，
我们倾向于声明一个空切片

初始化 Struct 引用

在初始化结构引用时，请使用 `&T{}` 代替 `new(T)` ，以使其与结构体初始化一致。

Bad	Good

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

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

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

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

初始化 Maps

对于空 map 请使用 `make(..)` 初始化，并且 map 是通过编程方式填充的。
这使得 map 初始化在表现上不同于声明，并且它还可以方便地在 make 后添加大小提示。

Bad **Good**

```
var (
  // m1 读写安全;
  // m2 在写入时会 panic
  m1 = map[T1]T2{}
  m2 map[T1]T2
)
```

```
var (
  // m1 读写安全;
  // m2 在写入时会 panic
  m1 = make(map[T1]T2)
  m2 map[T1]T2
)
```

声明和初始化看起来非常相似的。

声明和初始化看起来差别非常大。

在尽可能的情况下，请在初始化时提供 map 容量大小，详细请看 [指定 Map 容量提示]。

另外，如果 map 包含固定的元素列表，则使用 map literals(map 初始化列表) 初始化映射。

Bad **Good**

```
m := make(map[T1]T2, 3)
m[k1] = v1
m[k2] = v2
m[k3] = v3
```

```
m := map[T1]T2{
  k1: v1,
  k2: v2,
  k3: v3,
}
```

基本准则是：在初始化时使用 map 初始化列表 来添加一组固定的元素。否则使用 `make` (如果可以，请尽量指定 map 容量)。

字符串 string format

如果你在函数外声明 `Printf` -style 函数的格式字符串，请将其设置为 `const` 常量。

这有助于 `go vet` 对格式字符串执行静态分析。

Bad	Good

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

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

命名 Printf 样式的函数

声明 `Printf` -style 函数时，请确保 `go vet` 可以检测到它并检查格式字符串。

这意味着您应尽可能使用预定义的 `Printf` -style 函数名称。 `go vet` 将默认检查这些。有关更多信息，请参见 Printf 系列。

如果不能使用预定义的名称，请以 f 结束选择的名称： `Wrapf` ，而不是 `Wrap` 。 `go vet` 可以要求检查特定的 Printf 样式名称，但名称必须以 `f` 结尾。

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

另请参阅 go vet: Printf family check.

编程模式

表驱动测试

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 **Good**

```
// 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)
```

```
// 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。

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 {
  // ...
}
```

并行测试，比如一些专门的循环（例如，生成 goroutine 或捕获引用作为循环体的一部分的那些循环）
必须注意在循环的范围内显式地分配循环变量，以确保它们保持预期的值。

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

```
  // ...
}
for _, tt := range tests {
  tt := tt // for t.Parallel
  t.Run(tt.give, func(t *testing.T) {
    t.Parallel()
    // ...
  })
}
```

在上面的例子中，由于下面使用了 `t.Parallel()`，我们必须声明一个作用域为循环迭代的 `tt` 变量。
如果我们不这样做，大多数或所有测试都会收到一个意外的 `tt` 值，或者一个在运行时发生变化的值。

功能选项

功能选项是一种模式，您可以在其中声明一个不透明 Option 类型，该类型在某些内部结构中记录信息。您接受这些选项的可变编号，并根据内部结构上的选项记录的全部信息采取行动。

将此模式用于您需要扩展的构造函数和其他公共 API 中的可选参数，尤其是在这些功能上已经具有三个或更多参数的情况下。

Bad	Good

```
// package db

func Open(
  addr string,
  cache bool,
  logger *zap.Logger
) (*Connection, error) {
  // ...
}


// package db

type Option interface {
  // ...
}
```

```
func WithCache(c bool) Option {
  // ...
}

func WithLogger(log *zap.Logger) Option {
  // ...
}

// Open creates a connection.
func Open(
  addr string,
  opts ...Option,
) (*Connection, error) {
  // ...
}
```

必须始终提供缓存和记录器参数，即使用户希望使用默认值。

```
db.Open(addr, db.DefaultCache, zap.NewNop())
db.Open(addr, db.DefaultCache, log)
db.Open(addr, false /* cache */, zap.NewNop())
db.Open(addr, false /* cache */, log)
```

只有在需要时才提供选项。

```
db.Open(addr)
db.Open(addr, db.WithLogger(log))
db.Open(addr, db.WithCache(false))
db.Open(
  addr,
  db.WithCache(false),
  db.WithLogger(log),
)
```

我们建议实现此模式的方法是使用一个 `Option` 接口，该接口保存一个未导出的方法，在一个未导出的 `options` 结构上记录选项。

```
type options struct {
  cache  bool
  logger *zap.Logger
}

type Option interface {
```

```
  apply(*options)
}

type cacheOption bool

func (c cacheOption) apply(opts *options) {
  opts.cache = bool(c)
}

func WithCache(c bool) Option {
  return cacheOption(c)
}

type loggerOption struct {
  Log *zap.Logger
}

func (l loggerOption) apply(opts *options) {
  opts.logger = l.Log
}

func WithLogger(log *zap.Logger) Option {
  return loggerOption{Log: log}
}

// Open creates a connection.
func Open(
  addr string,
  opts ...Option,
) (*Connection, error) {
  options := options{
    cache:  defaultCache,
    logger: zap.NewNop(),
  }

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

  // ...
}
```

注意：还有一种使用闭包实现这个模式的方法，但是我们相信上面的模式为作者提供了更多的灵活性，并且更容易对用户进行调试和测试。特别是，我们的这种方式允许在测

试和模拟中比较选项，这在闭包实现中几乎是不可能的。此外，它还允许选项实现其他接口，包括 `fmt.Stringer`，允许用户读取选项的字符串表示形式。

还可以参考下面资料：

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

Linting

比任何 “blessed” linter 集更重要的是，lint 在一个代码库中始终保持一致。

我们建议至少使用以下 linters，因为我认为它们有助于发现最常见的问题，并在不需要规定的情况下为代码质量建立一个高标准：

- errcheck 以确保错误得到处理
- goimports 格式化代码和管理 imports
- golint 指出常见的文体错误
- govet 分析代码中的常见错误
- staticcheck 各种静态分析检查

Lint Runners

我们推荐 golangci-lint 作为 go-to lint 的运行程序，这主要是因为它在较大的代码库中的性能以及能够同时配置和使用许多规范。这个 repo 有一个示例配置文件 .golangci.yml 和推荐的 linter 设置。

golangci-lint 有 various-linters 可供使用。建议将上述 linters 作为基本 set，我们鼓励团队添加对他们的项目有意义的任何附加 linters。

Stargazers over time