目录

介绍

样式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。本指南基于`Uber Golang开发规范`而制定，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Golang。多年来，该指南已根据其他人的反馈进行了修改，其中许多是Golang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. The Go common mistakes guide

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IDE设置为

- 保存时运行`goimports`
- 运行`golint`和`go vet`检查错误
- 提交代码时勾选：`Go fmt`、`Rearrange code`、`Optimize imports`以及`Check TODO (Shwo All)`
- 如果不习惯使用`Go fmt`可以使用IDE自带的代码格式化来完成
- 不同的语言在代码格式化上处理不同，比如Java则不建议使用代码格式化而是**坚持自己写出来的代码就比格式化出来的代码要好**

指导原则

写出好代码的基本原则

好的代码就是一个艺术品，好的代码不管是借鉴开源代码还是在做Code Review时，都应该是赏心悦目的而不是读代码的时候时刻都在骂娘，如何写出好的代码，下面给出一些参考

- **规范不仅是文档，而是必须遵守的原则**，任何一个人熟悉规范是一回事（因为规范没有任何技术含量），但是能不能写规范的代码又是另外一回事
- 如果你写代码的时候发现自己在给一个方法或者变量命名时，时刻在注意变量名或者方法是不是简单而有效，而不是一股脑的只顾写代码，什么都不想
- 在考虑减少不必要的语句
- 写代码如同写英文作文，讲究段落感，该留出段落的地方一定要空行；意义相近的声明一定要分组
- **遵守规范并时刻强迫自己遵守规范**，如果没有相应的规范，就遵守现有的事实标准。如Http的

Header命名，你看浏览器默认发的Header命名即可（正确的应该是Xxxx-Yyyy而不是Xxxx_Yyyy）
- 培养自己手写代码的能力（在纸上写，而不是用IDE写），你和高级程序员之间差的就是手写代码的能力

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

Interface合理性验证

在编译时验证接口的符合性。这包括

- 将实现特定接口所需的导出类型作为其API的一部分
- 导出或未导出的类型是实现同一接口的类型集合的一部分
- 其他违反接口的情况会破坏用户

反例

```
type Handler struct {
  // ...
}
func (h *Handler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  // ...
}
```

正例

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

如果`*Handler`永远不会与`http.Handler`接口匹配,那么语句`var _ http.Handler = (*Handler)(nil)`将无法编译。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

```
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"}}

// 你只能通过值调用 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")
```

同样，即使该方法具有值接收器，也可以通过指针来满足接口。

```
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

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

反例

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

正例

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

如果你使用结构体指针，mutex可以非指针形式作为结构体的组成字段，或者更好的方式是直接嵌入到结构体中，如果是私有结构体类型或是要实现Mutex接口的类型，我们可以使用嵌入mutex的方法

```
// 为私有类型或需要实现互斥接口的类型嵌入
type smap struct {
  sync.Mutex // only for unexported types（仅适用于非导出类型）
```

```
  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]
}
```

```
// 对于导出的类型，请使用专用字段
type SMap struct {
  mu sync.Mutex // 对于导出类型，请使用私有锁

  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]
}
```

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

反例

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

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

// 你是要修改 d1.trips 吗?
trips[0] = ...
```

正例

```
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

反例

```
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()
```

正例

```
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

反例

```
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
```

正例

```
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)

反例

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

正例

```
// 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

反例

```
type Operation int

const (
  Add Operation = iota
  Subtract
  Multiply
)

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

正例

```
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

反例

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

正例

```
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`]

反例

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

正例

```
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`

- 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

反例

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

正例

```
// {"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

错误类型

Go 中有多种声明错误（Error) 的选项

- [`errors.New`] 对于简单静态字符串的错误
- [`fmt.Errorf`] 用于格式化的错误字符串
- 实现 `Error()` 方法的自定义类型
- 用[`"pkg/errors".Wrap`]的Wrapped errors

返回错误时，请考虑以下因素以确定最佳选择

- 这是一个不需要额外信息的简单错误吗？如果是这样，[`errors.New`] 足够了
- 客户需要检测并处理此错误吗？如果是这样，则应使用自定义类型并实现该 `Error()` 方法

- 您是否正在传播下游函数返回的错误？如果是这样，请查看本文后面有关错误包装 section on error wrapping 部分的内容
- 否则 [`fmt.Errorf`] 就可以了

如果客户端需要检测错误，并且您已使用创建了一个简单的错误[`errors.New`]，请使用一个错误变量。

反例

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

正例

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

如果您有可能需要客户端检测的错误，并且想向其中添加更多信息（例如，它不是静态字符串），则应使用自定义类型

反例

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

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

正例

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

直接导出自定义错误类型时要小心，因为它们已成为程序包公共API的一部分。最好公开匹配器功能以检查错误

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

错误包装

一个（函数/方法）调用失败时，有三种主要的错误传播方式

- 如果没有要添加的其他上下文，并且您想要维护原始错误类型，则返回原始错误
- 添加上下文，使用[`"pkg/errors".Wrap`]以便错误消息提供更多上下文，[`"pkg/errors".Cause`]可用于提取原始错误
- 如果调用者不需要检测或处理的特定错误情况，使用[`fmt.Errorf`]

建议在可能的地方添加上下文，以使您获得诸如“调用服务 foo：连接被拒绝”之类的更有用的错误，而不是诸如“连接被拒绝”之类的模糊错误，在将上下文添加到返回的错误时，请避免使用“failed to”之类的短语来保持上下文简洁，这些短语会陈述明显的内容，并随着错误在堆栈中的渗透而逐渐堆积

反例

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

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

正例

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

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

但是，一旦将错误发送到另一个系统，就应该明确消息是错误消息（例如使用`err`标记，或在日志中以"Failed"为前缀），另请参见 [Don't just check errors, handle them gracefully]，不要只是检查错误，要优雅地处理错误

处理类型断言失败

[类型断言]的单个返回值形式针对不正确的类型将产生 panic。因此，请始终使用"comma ok"的惯用法

反例

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

正例

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

不要panic

在生产环境中运行的代码必须避免出现panic。panic是[cascading failures]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

反例

```
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])
}
```

正例

```
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

反例

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

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

正例

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

f, err := ioutil.TempFile("", "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

反例

```
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!
}
```

正例

```
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

反例

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

正例

```
// 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 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) {
  // ...
}
```

反例

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

正例

```
// 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) {
  return l.list.Add(e)
}
// 移除从列表中移除实体。
func (l *ConcreteList) Remove(e Entity) {
  return 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

反例

```
// 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
}
```

正例

```
// 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) {
  return l.list.Add(e)
}
// 移除从列表中移除实体。
func (l *ConcreteList) Remove(e Entity) {
  return l.list.Remove(e)
}
```

无论是使用嵌入式结构还是使用嵌入式接口，嵌入式类型都会限制类型的演化

- 向嵌入式接口添加方法是一个破坏性的改变。
- 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替代方法替换嵌入类型，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

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

反例

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

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

正例

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

// 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

反例

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

// BenchmarkBad-4   50000000   22.2 ns/op
```

正例

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

// BenchmarkGood-4  500000000   3.25 ns/op
```

尽量初始化时指定Map容量

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)
```

为`make()`提供容量信息（hint）尝试在初始化时调整map大小，这减少了在将元素添加到map时增长和分配的开销（**想想这是为什么，不懂去看看map的resize算法你就明白了**）。注意，map不能保证分配hint个容量。因此，即使提供了容量，添加元素仍然可以进行分配

反例

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

files, _ := ioutil.ReadDir("./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
```

正例

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

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

// `m`是有大小提示创建的；在运行时可能会有更少的分配
```

规范

项目命名规范

参考各个开源项目（`spring`、`apache`以及`google`）的命名规范

- 项目名称使用使用小写字母
- 单词之间使用中划线`-`连接

源码文件规范

对于源码文件，有如下参考

- 全部使用小写字符
- 使用下划线`_`来连接各个单词
- **每个源文件末尾留出一空白行**（原因是在Linux/Unix等文本环境下可以很方便的跳到文件的末尾）

合理使用空格

合理的使用空格，能使代码更具有阅读性

- 关键字后加空格
- ,号之后加空格
-)和{之间加空格
- 注释//后加空格
- =号两边加空格

注释

注释能增强代码的可读性

- **禁止使用蹩脚英文注释**
- 不要为了注释而注释

反例

```
// call 这是方法调用
func call() {
  fmt.Println("call")
}
```

正例

```
// Upload 文件上传
// id 文件编号
// uploadType 上传类型
// ...
func Upload(
  id int64,
  uploadType UploadType,
  key string,
  filename string,
  outputFilename string,
  name string,
) (err error) {
  log.WithFields(log.Fields{
    "id":             id,
    "filename":       filename,
    "outputFilename": outputFilename,
    "uploadType":     uploadType,
    "key":            key,
    "name":           name,
  }).Info("开始上传文件")
}
```

- 注意空格

反例

```
//GET方法
const MethodGET = "GET"
```

正例

```
// GET方法
const MethodGET = "GET"
```

- Public（公开）的方法必须添加注释

变量名

给常量变量命名时，遵循以下原则

- 使用驼峰命名
- 力求准确表达出变量的意思，**不能使用无行业经验的单个字母命名的变量**
- 对于约定俗成的常量或者变量名，可以全部大写，比如GET、PUT、DELETE等
- 不能使用特殊符号如$、_等

反例

```
const a = "a"
var   b = "b"

const a_b = "a_b"
var   c_d = "c_d"
```

正例

```
const MethodGET   = "GET"
var   StudentName = "PUT"
```

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`以及`lib`，这些是不好的，信息量不足的名称

另请参阅[Package Names]和[Go 包样式指南]

方法

我们遵循Go社区关于使用[MixedCaps作为方法名]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`

除此之外，方法还有如下限制

- 方法体不能超过80行，如果超过需重构
- 和常量变量命名一致，不要图简单使用完全不相关的方法名(最极端的例子是方法名如`a` `b` `c`等)

反例

```
func f() {
  // 靠，f是干嘛的，鬼才知道
  //如果你的方法体过长，比如某公司一个方法6000多行代码，是的，你没看错，一个方法6000多行，而且还是知名公司
}
```

正例

```
func uploadFile() {
  // 一看就知道是上传文件的方法
}
```

换行

代码每行不超过140个字符（参看IDE里那根竖线，代码字符超过就要换行）

反例

```
// 方法签名超过了IDE的竖线，阅读代码时需使用滚动条
func upload(id int64, uploadType UploadType, key string, filename string,
outputFilename string, name string) (err error) {
  log.WithFields(log.Fields{
    "id":             id,
    "filename":       filename,
    "outputFilename": outputFilename,
    "uploadType":     uploadType,
    "key":            key,
    "name":           name,
  }).Info("开始上传文件")
}
```

正例

```
// 尽量保持一行一个参数，原因是可以很方便的对参数进行详细解释（如果有必要的话）
func upload(
  id int64,
  uploadType UploadType,
  key string,
  filename string,
  outputFilename string,
  name string,
) (err error) {
```

```
  // 方法调用也适用于换行
  log.WithFields(log.Fields{
    "id":             id,
    "filename":       filename,
    "outputFilename": outputFilename,
    "uploadType":     uploadType,
    "key":            key,
    "name":           name,
  }).Info("开始上传文件")
}
```

注意：如果意义相近，可以将实参或者形参放在一起成对出现（参看写出好代码的基本原则里面的代码要有段落感）

```
func callMethod(
  username string, password string,
  year int, month int, day int,
  hour int, min int, seconds int,
) {
  fmt.Println("test")
}

callMethod(
  user.Username, user.Password, // 用户名和密码意义相近，阅读代码的人一看就知道上下文
  year, month, day, // 月、日、年基本上一起出现
  hour, min, seconds, // 时、分、秒一样，基本上一起出现
)
```

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**

反例

```
import (
  "fmt"
  "os"

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

正例

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

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

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

反例

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

正例

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

这同样适用于常量、变量和类型声明

反例

```

const a = 1
const b = 2

var a = 1
var b = 2

type Area float64
type Volume float64
```

正例

```
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**

反例

```
type Operation int

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

正例

```
type Operation int

const (
  Add Operation = iota + 1
  Subtract
  Multiply
)

const ENV_VAR = "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

反例

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

  ...
}
```

正例

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

  ...
}
```

导入分组

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

反例

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

正例

```
import (
  `fmt`
  `os`

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

注意：**import语句使用``代替""**

函数分组与顺序

分组和顺序主要影响代码的逻辑性

- 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
- 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
- **简单而有效的办法是，将这些交给IDE去完成（详见IDE的相关配置）**

反例

```
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{}
}
```

正例

```
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

反例

```
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)
  }
}
```

正例

```
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

反例

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

正例

```
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

反例

```
var _s string = F()

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

正例

```
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。例外：未导出的错误值，应以`err`开头。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

反例

```
// 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.
}
```

正例

```
// foo.go

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

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（**想想基本原则里面的代码要有段落感**）

反例

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

正例

```
type Client struct {
  http.Client

  version int
}
```

使用字段名初始化结构体

初始化结构体时，几乎始终应该指定字段名称。现在由[`go vet`]强制执行

反例

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

正例

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

例外：如果有3个或更少的字段，则可以在测试表中省略字段名称

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

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（`:=`）

反例

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

正例

```
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

反例

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

正例

```
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

  ### 反例

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

  ### 正例

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

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

  ### 反例

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

  ### 正例

  ```
  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

  ### 反例

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

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

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

正例

```
var nums []int

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

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

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

反例

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

正例

```
if err := ioutil.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

反例

```
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
}
```

正例

```
data, err := ioutil.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

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（`/* ... */`）

反例

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

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

正例

```
// 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

反例

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

正例

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

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

反例

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

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

正例

```
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。

反例

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

// 声明和初始化看起来非常相似的
```

正例

```
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 尽量初始化时指定 Map 容量，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

反例

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

正例

```
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)

字符串格式化

如果你在函数外声明`Printf`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

反例

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

正例

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

命名Printf样式的函数

声明`Printf`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`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].

日志记录

好的日志能在代码出Bug时帮助你很快的定位问题

- 使用统一的日志框架记录日记，包括不限于（`logrus`、`zap`或者`zerolog`）
- **记录日志一定要带上`上下文`**

  ### 反例

  ```
  // 假如代码出了Bug，除了上传这几个字外，你还能得到更有用的信息?
  log.Info("开始上传文件")
  ```

正例

```
// 假如代码出了Bug，可以通过id, filename, uploadType, key, name等信息去Root Cause
log.WithFields(log.Fields{
  "id":             id,
  "filename":       filename,
  "outputFilename": outputFilename,
  "uploadType":     uploadType,
  "key":            key,
  "name":           name,
}).Info("开始上传文件")
```

- 别把时间花在研究哪家的日志框架更好上，只要性能不是最差的就可以了。事实上，更好的办法是引入第三方的日志解决方案，程序只记录日志，日志的上传和分析交给第三方日志解决方案去处理吧
- 区分好日志级别(Debug, Info, Warn, Error等级别)，别乱用

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

反例

```
// 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)
```

正例

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

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

反例

```
// package db

func Open(
  addr string,
  cache bool,
  logger *zap.Logger
) (*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)
```

正例

```
// 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) {
  // ...
}


// 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，在不可能进行比较的情况下它允许在测试和模拟中对选项进行比较。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

- [自引用功能和选项设计]
- [友好的API的功能选项]