Golang Code Style

Joe.Zhong

Version 1.0

Nov 03, 2022

Revision History

Date	Version	Description	Author(s)
	1.0	Draft	

Table of Contents

6. Readability

6.1 避免参数语义不明确
6.2 避免转义
6.3 删除不必要的else
6.4 使用tag
6.5 避免使用init()
6.6 不使用panic，返回错误
6.7 减少嵌套

7. Security

7.1 defer回收资源
7.2 处理类型断言异常，避免产生panic
7.3 通过errors.is匹配错误
7.4 顶层分支使用recover，避免panic
7.5 for range引用方式(值与参考)

1. Variable

1.1 对于未导出的顶层常量和变量使用前缀"_"

在未导出的顶级vars和consts 前面加上**前缀**"_"，以使用它们在被引用时，知道是全局符号
例外：未导出的错误值，应该以err开头，或 表征错误的前缀开头
基本依据：顶层变量和常量具有包范围作用域. 使用通用名称很容易在其他文件中意外引用 未导出符号

Bad	Good

// foo.go
const (
  defaultPort = 8080
  defaultUser = "guess"
)
 	
// bar.go
func Bar() {
  defaultPort := 9000 // 删除此行会导致编译错误
  fmt.Println("Default Port: ", defaultPort)
}
	// 通过前缀约定 可知道 顶级变量作用范围
	// foo.go
	const (
	  _defaultPort = 8080
	  _defaultUser = "guess"
	)

1.2 顶层变量声明

在顶层声明变量时，请使用标准**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( )
	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()
```

1.3 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，即： = .

Bad	Good
var s = "foo"	s := "foo"

但是，在某些情况下，使用var关键字 设置默认值会更清晰. 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)
	  }
	  }
	}

1.4 尽量缩少变量作用域

如果可能，尽量缩小变量作用范围. 除非它与减少嵌套规则冲突.

Bad	Good

err := ioutil.WriteFile(name, data, 0644)
if err != nil {
  return err
}
	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，则不应该尝试缩小范围.

Bad	Good

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
}
	data, err := ioutil.ReadFile(name)
	if err != nil {
	  return err
	}
	if err = cfg.Decode(data); err != nil {
	  return err
	}
	fmt.Println(cfg)
	return nil

2. Struct

2.1 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.

Bad	Good

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

2.2 初始化时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.

Bad	Good

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

2.3 通过var声明零值结构

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

2.4 初始化引用

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
sptr := new(T) sptr.Name = "bar"	sptr := &T{ Name: "bar" }

2.5 嵌套约定

嵌入类型应该放置在字段列表顶部，且必须有一个空行，以区别于常规的嵌套字段.

Bad	Good

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

3. Slice

3.1 声明空slice

声明零切片，len与cap都为零时，通过var关键字声明，无需调用make()创建

Bad	Good

num := [ ] 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)
	}

3.2 检查切片是否为空

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
	}

3.3 返回空slice

nil是一个有效长度为0的slice，返回长度为零的slice时，应该返回nil

Bad	Good

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

3.4 尽可能指定容量

尽可能指定slice的容量大小，这将减小追加元素时造成重新分配内存的损耗.

Bad	Good

data := make([ ] int, 0)
for i := 0; i < size; i++ {
  data append(data, i)
}
	data := make([ ] int, 0, size)
	for i := 0; i < size; i++ {
	  data = append(data, i)
	}

3.5 访问slice时，必须先检查长度

在访问slice时，必须先检查长度是否合法，以防止程序panic.

Bad	Good

func check(data [ ] byte) bool {
  if data[0] == 'a' && data[1] == 'b' {
  return true
  }
  return false
}
	func check(data [ ] byte) bool {
	  if len(data) > 2 {
	  if data[0] == 'a' && data[1] == 'b' {
	  return true
	  }
	  }
	  return false
	}

3.6 尽量不作为入参

slice是引用类型，作为入参时采用地址传递，函数中对slice的更改会影响原始数据.

Bad	Good

func modify(array [ ] int) {
  if len(array) > 1 {
  array[0] = 1
  }
}
 	
func main() {
  arr := [ ] int { 0, 2, 3, 4, 5 }
  modify(arr)
}
	func modify1(array [ ] int) { // 先copy，再传参
	  if len(array) > 1 {
	  array[0] = 1
	  }
	}
	
	func modify2(array [5]int) { // 无需copy，直接传参
	  array[0] = 1
	}
	
	func main() {
	  var arr [5] int
	  array := [ ] int { 0, 2, 3, 4, 5 }
	  copy(arr[:], array)
	  modify(arr)
	}

3.7 避免重复执行len

len是go内置函数，每次调用都会计算slice的长度，要避免重复计算slice长度.

Bad	Good

list := [ ] int {1, 2, 3, 4, 5, 6, 7}
for i := 0; i < len(list); i++ {
  // ....
}
	list := i, l := 0, len(list); i < l; i++ {
	  // ...
	}

3.8 避免重复将字符串转换至slice

[ ]byte(str)会以 字符串 为内容基础 并创建字节slice，要避免反复从固定串创建字节slice.

Bad	Good

for i := 0; i < n; i++ {
  w.Write([ ] byte(str))
}
	for i, data := 0, []byte(str); i < n; i++ {
	  w.Write(data)
	}

3.9 通过内置copy方式复制slice

slice持有指向数据的指针，属于引用访问，因此 在复制时不能简单地使用赋值方式.

Bad	Good

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = trips
}
 	
trips := ...
d1.SetTrips(trips)
 	
// Did you mean to modify d1.trips?
trips[0] = ...
	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 d.trips.
	trips[0] = ...

4. Map

4.1 使用make初始化

声明map后，必须通过make对其进行初始化，访问未初始化的map会导致panic.

Bad	Good

var (
  m1 = map[T1]T2{ } // 读写安全
  m2 map[T1]T2 // nil map，写入时会panic
)
	var (
	  m1 = make(map[T1]T2) // 读写安全
	  m2 map[T1]t2 // nil map，写入时会panic
	)

可以通过初始化列表的方式 来 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,
	}

4.2 初始化时尽量指定容量

尽可能指定map的大小，这将减少将元素加入map时造成重新分配内存而造成的损耗.

Bad	Good

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

4.3 禁止并发写map

由于并发写map会造成panic，所以在并发场景下，必须通过锁机制来保证写操作的原子性.

Bad	Good

m := make(map[int]int)
go func() {
  for {
  _ = m[1]
  }
}()
go func() {
  for {
  m[2] = 1
  }
}()
select { }
	m, lock := make(map[int]int), &sync.Mutex{}
	go func() {
	  for {
	  lock.Lock()
	  _ = m[1]
	  lock.Unlock()
	  }
	}()
	go func() {
	  for {
	  lock.Lock()
	  m[2] = 1
	  lock.Unlock()
	  }
	}()
	select { }

.

4.4 暴露内部map时，只能返回数据副本

因为暴露内部map时，会导致map不受锁保护，所以只能返回数据副本.

Bad	Good

type model struct {
  sync.Mutex
 	
  data map[string]string
}
 	
func (p *model) Snapshot() map[string]string {
  p.Lock()
  defer p.Unlock()
 	
  return p.data
}
 	
// snapshot 不再受锁机制保护
snaphot := m.Snapshot()
	type model struct {
	  sync.Mutex
	
	  data map[string]string
	}
	
	func (p *model) Snapshot() map[string]string {
	  p.Lock()
	  defer p.Unlock()
	
	  c := make(map[string]string, len(p.data))
	  for k, v := range p.data {
	  c[k] = v
	  }
	  return c
	}
	
	snaphot := m.Snapshot()

5. Channel

5.1 禁止使用未初始化channel

读写未初始化的channel会造成deadlock，关闭未初始化的channel，会造成panic.

Bad	Good
var ch chan int ch <- 0 // 死锁 ... <-ch // 死锁 .... close(ch) // panic	ch := make(chan int, 2) ch <- 0 <- ch close(ch)

5.2 无缓冲channel，必须先读后写

对于无缓冲channel，只有写入没有读取、只有读取没有写入 或者 在单线程场景先写后读，以上情况都会造成死锁

Bad	Good
ch := make(chan int) ch <- 0 go func() { <-ch }() time.Sleep(time.Second)	ch := make(chan int) go func() { <-ch }() ch <- 0 time.Sleep(time.Second)

5.3 严格审查缓冲区大小

缓冲区的大小与使用场景有密切关系，大小的制定必须经过严格的审查.

Bad	Good

ch := make(chan int, 24) // 不可能满足所有场景	// size1 最大并发量;
	// size2 容易极限
	// size3 最大有效数量;
	ch := make(chan int, sizeN)

5.4 插入元素时，尽量增加超时检测

增加超时检测可以帮助发现程序运行时的业务处理瓶颈，记录日志 并根据日志反馈的情况调整尺寸.

Bad	Good

ch <- any // 可能会引起阻塞	select {
	  case ch <- any:
	  // 正常情况
	  case <- time.After(pushTimeout):
	  // 超时情况
	}

5.5 禁止重复释放

重复释放channel会造成panic，它一般存在于异常流程处理分支中，不容易被发现.

Bad	Good

func foo(c chan int) {
  defer close(c)
 	
  err := business()
  if err != nil {
  c <- 0
  close(c) // 重复释放
  return
  }
  c <- 1
}
	func foo(c chan int) {
	  defer close(c)
	
	  err := business()
	  if err != nil {
	  c <- 0
	  return
	  }
	
	  c <- 1
	}

6. readability

6.1 避免参数语义不明确

当函数中的参数的含义不明显时，应添加注释加以说明.

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

6.2 避免转义

含有转义符的字符串不容易被读懂，串值也不直观；通过"`"声明字符串，可以避免转义.

Bad	Good
wantError := "unknown name: \"test\""	wantError := `unknown error: "test"`

6.3 删除不必要的else

如果在if的两个分支都对变量赋值，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
	}

6.4 使用tag

通过tag注释可以让字段在不同系统之间互相兼容，也可以明确不同场景下的约定.

Bad	Good

type Config struct {
  Id string
}
	type Config struct {
	  Id string `toml:"id" json:"id" gorm:"..."`
	}

6.5 避免使用init()

多个init时，执行顺序不直观，而且init之间的依赖关系很难维护.

Bad	Good

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

6.6 不使用panic，返回错误

panic会导致线上业务异常或者中断，所有异常分支都必须返回错误，不能panic.
panic/recover不能作为错误处理策略；但程序初始化是例外的，初始化失败可以panic.

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)
	  }
	}

6.7 减少嵌套

代码要优先处理错误/特殊情况，尽早返回，继续循环来减少嵌套.

Bad	Good

func _, 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()
	}

7. Security

7.1 defer回收资源

使用defer来回收和撤销资源，如 文件、内存对象 和 锁.

Bad	Good

p.Lock()
if p.count < 10 {
  p.Unlock()
  return p.count
}
p.count++
newCount := p.count
p.Unlock()
return newCount
// 很容易忘记解锁，代码可读性也较差
	p.Lock()
	defer p.Unlock()
	if p.count < 10 {
	  return p.count
	}
	p.count++
	return p.count
	// 解锁的代码位置固定，也不容易重复解锁，可读性较好；代码块体积越大，效果越明显

7.2 处理类型断言异常，避免产生panic

类型断言时，单个返回值可能会引起异常，保险的方式是捕捉类型断言异常并正常处理它.

Bad	Good

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

7.3 通过errors.is匹配错误

通过 errors.is 匹配已知的错误.

No error matching	Error matching

// 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")
	  }
	}

7.4 顶层分支使用recover，避免panic

敏捷场景中，业务是并行开发，更新迭代较为频繁，recover可以保护已稳定的业务逻辑.

Bad	Good

func ProcessMsg(packet *Packet) {
  switch packet.Header.CmdType {
  case ClientCmd:
  handleMsg(packet)
  }
}
	func ProcessMsg(packet *Packet) {
	  defer func() {
	  if p := recover(); p != nil {
	  err = fmt.ErrorF("")
	  if e, ok := p.(error); ok {
	  err = e
	  }
	  log.Fatalf(err)
	  }
	  }()
	
	  switch packet.Header.CmdType {
	  case ClientCmd:
	  handleMsg(packet)
	  }
	}

7.5 for range 引用方式(值与参考)

for range只会评估(evaluate)一次，并复制(copy)值，再赋予变量.

Bad	Good

type someStruct struct {
  Content string
}
func modifyFail(any []someStruct) {
  for _, v := range any {
  ptr := &v
  ptr.Content = "new"
  v.Content = "new"
  }
}
 	
func main() {
  any := []someStruct {
  {
  Content: "old",
  },
  }
  modifyFail(any)
  fmt.Println(any)
}
	func modify1(any []someStruct) {
	  if nb := len(any); nb > 0 {
	  for i := 0; i < nb; i++ {
	  cp := &any[0]
	  // do something
	  cp.Content = "new"
	  }
	  }
	}
	
	func modify2(any []someStruct) {
	  for k, _ := range any[:] {
	  // do something
	  any[k].Content = "new"
	  }
	}