go安全编码规范

整理知识，传播智慧

好好学习，天天向上

gopher：石鸟路遇

email：1245863260@qq.com

Simple, Poetic, Pithy

1、目标

提高代码质量，提高代码可读性、可维护性、安全性。

原则：可读性、可维护、安全、可靠、可测试、高效、可移植。

2、代码风格

涉及标识符、格式、注释风格。一致的编程习惯，能使代码容易阅读，便于理解和维护。

2. 1、命名

使用统一的风格命名：大驼峰和小驼峰

类别	导出使用	包内使用
结构体、接口、自定义别名和类型	大驼峰	小驼峰
函数、方法	大驼峰	小驼峰
结构体成员变量	大驼峰	小驼峰
常量	大驼峰	小驼峰
全局变量	大驼峰	小驼峰
方法接受者、函数参数、返回值、局部变量		小驼峰
break、continue、goto的标签		小驼峰

> 包内使用小驼峰、外部引用大驼峰

常量

const定义修饰，其值在程序生命周期内固定不变。

2.1.1、方法接受者

接受者名称应当简洁（1-2个字母即可），不要使用me、self、this这种泛指的名称。

```
// 正例
func (c *Controller) gofun(c <-chan struct{})
```

2.1.2、避免使用go内置标识符

```
// 反例
var string string = "stone bird"
```

2.1.3、导出错误的变量采用ErrName的格式

```
type ReadError struct {

}

var ErrReading = errors.New("read err")
```

> 错误类型: 使用*NameError*
>
> 错误变量：*ErrName*

2.1.4、尽量避免使用包名作为前缀

```
// stream.go
package stream

// 反例
type StreamBuffer struct{ //不符合，var buf = stream.StreamBuffer过于累赘

}

// 正例
type Buffer struct{ // 符合 var buf = stream.Buffer 层次清晰

}
```

2.1.5、文件名

文件名全部采用小写单词、允许包含数字和'_'组合方式。

sha256.go

http_cli.go

stone.pb.go // 特殊例子protobuf 格式

2.1.6、目录名

目录名采用全小写的方式、允许包含数据、'-'（头尾不能是中横线）。

hurl-cli

md5

2.1.7、包名

包名采用全小写单词，允许包含数字，无下划线（测试包除外）

```
// 反例
package stone_bird

// 正例
package sha256
package sha256_test
```

2.1.8、包名要和所在目录名一致

如果不一致，会给阅读带来困难，难以理解。

2.2 注释

`//` 单行注释　`/* */` 多行注释

注释和代码一样重要、应该按需注释。

2.2.1、文件头注释包含包含版权说明

文件头部首先应该包含版权说明。如果需要添加其他内容，可以在版权说明下面补充。

如文件功能说明、作者、日期、注意事项等。

```
/*
Copyright (c) 2021 hu. All rights reserved.
@Author: stonebirdjx
@Email: 1245863260@qq.com, g1245863260@gmail.com
@File: hurl.go
@Date: 2021/11/14 20:42
@Desc: hurl tool v2.0 main function
*/
```

2.2.2 、包注释

紧贴package语句之上的注释称为包注释，包注释内容可在`godoc`上显示

```
/*
Package zip
*/
package zip
```

2.2.3、代码注释

所有外部引用的标识符都应该有注释。

```
// FindByName find some string by enter the name
func FindByName(name string) string{

}
```

2.2.4、注释符合注释内容之间要保留一个空格

```
/* 单行注释 */
// 单行注释

/*
第一行注释
第二行注释
第三行注释
*/
```

2.2.5、代码注释应该置于代码的上方和右边

```
// StoneBird 函数注释
func Stonebird(){
    name := "json" // name变量注释
}
```

2.2.6、不写空有格式的函数和方法头注释

尽量通过函数名注释其功能，按需写函数头注释.

> 避免写无用、信息冗余的函数头注释。

```
// 反例

// 函数名:WriteData  --重复
// 功能: 写入字符串
// 参数:             --空有格式没内容
// 返回值:           --空有格式没内容
func WriteData(buf []byte,length int)(int error){}
```

2.2.7、正式代码（给客户）不应该包含TODO/TBD/FIXME注释

```
// 反例

// TODO: 补充xxx
// FIXME: 修复相关bug
```

2.3、格式

2.3.1、使用工具对代码格式化

使用gofmt对代码进行格式化，会将GO源码按标准风格进行缩进、对齐、保留注释等。

使用goimports，与gofmt拥有相同格式化功能，还能自动删除和引用包。

可以用goland的工具格式化代码。

2.3.2、行宽不超过120个字符。

代码行宽不宜过长，否则不利于阅读。建议不超过120个字符长度，除非能显著增加可读性。

2.3.4、相对独立的程序块或者语句之间用空行分割。

减少空行使用，一般一个空行即可，最多两个空行。

```
// 反例
err := do()

if err != nil { // 不是相对独立的代码，不应该用空行隔开

}

// 正例
func first(){}

func second(){}
```

3、编程实践

3.1、声明和初始化

3.1.1、保证作用域尽量的小

作用域值变量、函数、类型等在程序内可见可引用的范围。范围越大，引起错误的可能性就越高。

进需要公开对外部的接口时导出符合：不能导出只在保内使用的常量、变量、函数、或闭包。

最小化作用域的方法：

> *1、尽量推迟变量定义，在变量使用时才能被初始化*
>
> *2、把相关语句放在一起或单独提取到子函数，是函数尽量小而集中，功能单一*

```
// 反例
i := 0 // 不符合，循环之后不需要访问i。而且i容易被篡改
for ; i < 10; i++ {
}

// 正例
for i := 0; i < 10; i++ {
}
```

3.1.2、不要使用难以理解的字面量

难以理解的字面量是指通过代码上下文难以明确业务含义的字面量。

```
// 反例

// 下面的数字字面量，不容易理解其具体含义
switch curType {
    case 1:
    case 2:
    case 3:
    default:
}

// 正例
type ServType int
const (
    ServSet ServType = 1
    ServQuery ServType = 4
)
var curType = getType()
// 通过名称判断
switch curType {
case ServSet:
case ServQuery:
default:
}
```

3.1.3、避免使用全局变量

禁止将全局变量导出，使用全局变量会导致业务代码和全局变量间产生耦合，并且难以追踪数据的变化。应尽量避免使用全局变量。

包内使用全局变量，应该对其进行封装，并注意避免因并发而出现的数据竞争。

```
// 反例
var SumValue = 0 // 不符合，导出了全局变量。

// 正例
var total = 0 // 有当前模块确保数据不存在竞争

func GetTotal() int {
    return total
}
```

3.1.4、变量使用时才声明并初始化

遵循变量作用域最小原则与就近声明原则，在变量使用时才声明并初始化，使得代码更容易阅读。

```
// 反例
var name string
...
name = "stone bird"

// 正例
var name = "stone bird"
```

3.1.5、相关申明放在一个组内

具有相同用途的常量、或变量应该放在一个常量组、或变量组。相关性高的放置在同一个组里面，以提升代码的可读性。

```
type Duration int

const (
    Nanosecond Duration = 1
    MicroSecond  = 1000 * Nanosecond
    ...
)
```

3.1.6、初始化结构体变量时、尽可能采用复合字面量的方式

初始化结构体变量时，优先采用复合字面量的方式，并指定字段名称，以提升代码可读性。应该尽量用每一行一个`字段名：初始化值`的形式。

> 数组和*slice*的成员是结构体的情况除外。

```
type stoneBird struct {
    name string
    age int
    language string
}

// 反例
sb := new(stoneBird)
sb.name = "stone"
sb.age = 18
sb.language = "go"

// 正例
func New() stoneBird{
    return stoneBird{
        name:     "stone",
        age:      18,
        language: "go",
    }
}

// 例外
s := []stoneBird{
    {name: "stone",age: 18,language: "go"},
    {name: "stone",age: 19,language: "go"},
    {name: "stone",age: 20,language: "go"},
}
```

```
s1 :=  []stoneBird{
    {"stone",18,"go"},
    {"stone",19,"go"},
    {"stone",20,"go"},
}
```

3.2 整数

3.2.1、确保无符合整数运算时不会回绕

所谓回绕是指无法用无符号整数表示运算结果。这个结果会根据该类型可以表示的最大值加1执行求模操作。

+、-、*、++、--、+=、-=、<<=、<<

将运算结果用以一下用途、应防止回绕

> 1、作为数组、slice的索引
>
> 2、作为待申请数组变量的大小。
>
> 3、作为数组、slice的边界

```
// 反例
func add(a, b uint64) {
    c := a + b // 如果 a + b > math.MaxUnit64 会存在无符号整数回绕（从0开始计算）
}

// 正例
// 加法判断是否大于 math.MaxUint64，其他类型类推
func add(a, b uint64) (uint64, error) {
    if a > math.MaxUint64-b {
        return 0,errors.New("lager num")
    }
    return a + b, nil
}

// 减法判断 前者是否小于后者
func sub(a, b uint64) (uint64, error) {
    if a < b {
        return 0, errors.New("回绕")
    }
    return a - b, nil
}
```

3.2.2、确保有符合整数运算时不会出现溢出

“整数溢出”是一种未定义的行为，意味着编译器处理器有符合整数溢出时具有很多选择，有符合整数溢出会发生在如下操作中

+、-、*、/、++、--、+=、-=、/=、%=、<<=、<<

后续代码如以此作为分配内存的大小，将导致申请的内存比实际所需的过小、或者过大，从而导致内存不足的问题。

将运算结果用以一下用途、应防止溢出

> 1、作为数组、slice的索引
>
> 2、作为待申请数组变量的大小。
>
> 3、作为数组、slice的边界

```
// 反例
func add(a, b int32) {
    c := a + b // a + b > math.MaxInt32 时两者相加会产生有符合整数溢出
}

// 正例,需要先计算会不会溢出（从负值开始计算）
func doint(a,b int32) {
    var c int32
    if (b > 0 && a > (math.MaxInt32-b)) ||
        (b < 0 && a < (math.MaxInt32-b)){
        // 错误处理
    }
    c = a + b
}
```

3.2.3、确保参与移位的操作数的位数足够

移位的位数应该是无符号整数，go对无类型的常数依照参与表达式运算数据类型进行推导

```
// 反例
func foo(num uint16,bit uint8) bool {
    if num > (1 << bit) { // 不符合：这里（1 << bit）被推导成了uint16，可被移位操作回绕
        return true
    }
    return false
}

// 正例
func foo(num uint16,bit uint8) bool {
    if uint32(num) > (uint32(1) << bit) { // 强制类型转换后，应满足函数设计要求
        return true
    }
    return false
}
```

3.2.4、确保除法运算和模运算中的除数不为0

```
if a == 0 {
    return false
}
avg = total / a
```

3.2.5、确保整数转换不会造成数据截断或者符号错误

当一个较大类型转换为较小类型时，并且该数原值超出较小整型的表示范围，就会发生截断错误。

有符号整型向无符号整型转换时，最高位会丧失作为符号位的功能

> 1、带符号整数值为非负时，向无符号整型转换后，值不变

> 2、带符号整数值为负数时，向无符号整型转换后，结果通常是一个非常大的整数

```
// 反例
var a = math.MaxInt32
b := int16(a) // 不符合，不同类型强制转化数据会发生数据截断

// 正例
if (a < math.MinInt16) || (a > math.MaxInt16) {
    // 错误处理
}

// 反例
var a = math.MinInt32
b := uint32(a) // 不符合，产生符号丢失

// 正例
if a < 0 {
     // 错误处理
}
b := uint32(a)
```

3.3 、字符串

3.3.1、需要字符串转义时优先使用raw string

raw string 是反引号的字符序列，`foo`

```
// 反例
const prefix = "name=\"stonebird\""

// 正例
const prefix = `name="stonebird"`
```

3.4、数组和slice

3.4.1、避免使用短声明定义一个空slice

使用var 关键字时定义空slice更清晰，并且不容易出错。

```
// 反例
filter = []int{} // 不符合：避免短声明定义一个空的slice

// 正例
var filter []int
```

3.4.2、始终使用len(s)==0检查slice是否为空

要检查slice是否为空（没有元素），必须始终检查其长度是否为0，长度为0的slice不一定是nil 如`filter := []int{}`,当如果slice ==nil

长度一定为0.

```go
// 反例
func isEmpty(s []string) bool {
    return  s == nil // 不符合：申请了空间的slice长度为0，不是nil 如[]int{}
}

// 正例
func isEmpty(s []string) bool {
    return  len(s) == 0 // 不符合：申请了空间的slice长度为0，不是nil 如[]int{}
}
```

3.5、map

3.5.1、确保key读取到map的元素有效

key如果不存在，会返回零值。使用if ok ，判断key是否存在。

```go
if v, ok := mp["key"]; ok {

}
```

3.6、表达式

3.6.1、表达式的比较，应该遵循左侧变化、右侧不变的原则

当变量或方法调用与常量比较时，如果常量在左边，不符合阅读习惯，难以理解

```go
// 反例
if nil != p && Max > p.v {

}

// 正例
if p != nil && p.v > Max {

}

// 例外
if v > min && v < max{

}

if min < v && v < max{

}
```

3.7 控制语句

3.7.1、循环必须有显示的退出条件

程序无法正常结束是一种缺陷，特别是它无法响应外部对他的控制。如果循环没有退出条件，循环在任何或预计时间内将完不成执行，导致死循环。

应尽量避免使用空for，如果无法避免，需要在循环过程中设置能退出的条件，并使用break退出。

```
// 反例
for {
    // 不符合，没有显示的退出条件
}

// 正例
for {
    if condition {
        break
    }
}
```

3.7.2、避免在循环体中修改循环控制变量

如果在循环体修改循环控制参数的数值，可能导致死循环，或者循环次数达不到预期

```
// 反例
for i := 0 ; i < 10 ; i++ {
    i -= 1
}

// 正例
for i := 0 ; i < 10 ; i++ {

}
```

3.7.3、慎用goto语句

goto语句会破坏程序的结构性，除非必须不适用goto，使用goto时，`也只允许跳转到本函数内的goto语句之后的标签`（向下跳转）。

3.7.4 、goto语句只能向下跳转

使用向前跳的goto语句会增加代码复杂性，使得程序结构难以理解

```
// 反例
unsafe:
if done {
    goto unsafe // 不符合：goto往上跳转标签
}

// 正例
if done{
    goto label
}
label:
```

3.7.5、Switch要有default分支

每个switch，都应该包含一个default分支，即使default分支，没有业务逻辑代码。

> 枚举类型除外

```
// 正例
switch str {
case "a":
default:
}

// 例外
switch c {
case red:
case green:
case blue:
}
```

3.7.6、禁止使用浮点数作为循环计数器

存储浮点数的精度有限，精度问题可能导致条件判断不准确，导致循环达不到预期。

```
// 反例
for i := float32(2000000001);i < float32(2000000001);i++ {
    // 2e+9 不会进入循环
}
```

3.7.7、不要再迭代集合数据结构的过程中条件或者删除元素。

使用for-range或for循环可以便捷的访问map和slice中的元素。

对slice是修改不会影响循环次数，但会影响结果。

对map的key的增加或删除会影响循环次数。

```
// 反例
func main() {
    a := []int{0, 2, 1, 3, 5}
    for _, val := range a { // 不符合，在迭代原有的slice时，同事删除了索引为2的元素。
        fmt.Println(val)
        if len(a) > 4{
            a = append(a[0:2],a[3:]...)
        }
    }
}
// output 0 2 3 5 5

for k , v := range mp {
    mp["answer"] = 10 // 不符合：增加了key会影响对mp的变量
}
```

3.8、函数和方法

3.8.1、函数和方法的设计

3.8.1.1、合理选择方法的接受者

合理选择方法接受者的类型，对代码的可读性，和维护性都有不同的影响。

原则如下：

> *1*、如果方法不会改变接受者的内容，使用值类型，表明方法的*const*定义。
>
> *2*、如果接收者是大型结构体或者数组，可考虑指针类型，提高效率。
>
> *3*、如果接受者包含*sync.Mutex*或类似同步字段的结构体，则必须使用指针，以免被复制。
>
> *4*、如果接收者是*map*、*func*、或者*chan*，不要使用指针类型。
>
> *5*、如果方法接收者是*slice*，且方法中不会修改接受者容量，或者重新分配内存，不要使用指针类型。

3.8.1.2 函数功能要单一

函数功能要单一，过长的函数往往意味着函数功能不单一。可以进一步拆分或分层。而且过于复杂的函数往往不利于阅读，难以维护。

原则如下：

> *1*、函数行数，建议不超过*50*行（非空非注释）
>
> *2*、函数的参数个数，建议不超过*5*个
>
> *3*、最大代码块嵌套深度，建议不超过*4*层。
>
> *4*、函数的返回值不超过*3*个。

函数参数

函数的参数过多，会使得函数易于受到外部代码变化的影响，影响维护工作。优化方式如下

- 对方法进行抽象或重构
- 将相关函数合在一起，定义成结构体。

函数返回值

函数或方法返回值过多，会降低可维护性，也给使用者带来不必要的负担。

如果返回值太多，可以将关系密切的返回值参数封装成一个结构体。

函数嵌套深度

嵌套深度是指函数代码块之间相互包含的深度。嵌套过深，不利于理解。

```
// 反例
if command {
    err := getSomething()
    if err != nil {
        return false
    }
    return true
}
return false

// 正例
if !command {
```

```
    return false
}

err := getSomething()
if err != nil {
   return false
}

return true
```

3.8.1.3、设计函数时优先使用返回值而不是输出参数

优先使用返回值，而且错误值和返回值不要复用一个返回值。

```
// 反例
func newInt(b []byte,i int, x *int) int {  // 不符合，通过传入的指针来修改外部的实参值
    for ; i < len(b) && isDigit(b[i]);i++ {

    }
    *x = 0
}

// 正例
func newInt(b []byte,i int) (int,int){

}
```

3.8.1.4、函数的返回值应避免使用具名返回值

```
// 反例
func (c *conn) Bad(node *Node ,err error){ // 不符合：命名的返回值与其类型存在冗余信息。不方便阅读，return需要推断。

}

// 正例
func (c *conn) Bad(*Node ,error){

}

// 例外
func getName()(firstName, lastName, nickName string){
    firstName = "stone"
    lastName = "bird"
    nickName = "h"
    return
}
```

3.8.1.5、返回已被修改大小的slice参数对象

slice 的append 操作有可能修改slice指向的地址，如果函数被设计用于更新传入的slice参数，则他的实现需要满足：

如函数包含的slice的参数（不是slice指针），并对其append操作，必须返回修改后的slice。

```
// 反例
func appendSlice(s []int){ // 不符合：修改了slice大小，因此必须返回slice
    s[0] = 10 // 符合：只是修改内容没有修改大小
    s = append(s,11) // 不符合：修改了大小，如函数不返回s则不符合规则。
}

// 正例
func appendSlice(s []int) []int {
    s[0] = 10
    s2 = append(s,11)
    return s2
}
```

3.8.1.6、避免栈调用层次太深

调用层次太深，在栈上分配大量内存，容易导致出现栈溢出错误。GO语音有一些容易让函数调用陷入无限递归的情况，需要避免。

- 指定嵌入类型成员调用嵌入类型的方法。
- 确保String()方法不会被无限递归。

```
// 反例
func (j *job) Printf(str string) {
    j.Printf(str string) // 不符合：直接调用j.Printf会导致无限递归调用
}
// 正例
func (j *job) Printf(str string) {
    j.Logger.Printf(str string)
}
```

3.8.2、init

3.8.2.1、一个文件只定义一个init函数

一个文件只定义一个init函数，同时把init函数尽量放在源文件靠前的位置，这样有助于代码阅读。

3.8.2.2、一个包如果有多个init函数，不要存在任何依赖关系

不要再多个init函数之间，依赖变量、或资源的初始化顺序。

3.8.2.3、注意init函数内进行初始化的场景限制

由于init函数没有返回值，如果init函数初始化失败，只能通过异常反馈错误。因此应该慎用init函数进行复杂的初始化操作。

注意如下：

- 避免在init函数中进行资源申请、打开文件、连接数据库等可能失败的操作。
- 避免在init函数中调用可能抛出异常的函数。

3.8.3、闭包

3.8.3.1、禁止在闭包中直接使用循环控制变量

```
// 反例
for i := 0 ;i < 5 ; i++ {
    wg.Add(1)
    go func (){ // 不符合：避免在闭包中直接使用i
        defer wg.Done()
        fmt.Println(i)
    }()
}
// output  55555

// 正例
for i := 0 ;i < 5 ; i++ {
    wg.Add(1)
    go func (j int) {
        defer wg.Done()
        fmt.Println(j)
    }(i)
}
```

3.9、结构体和接口类型

3.9.1、合理使用匿名嵌套

匿名嵌套的主要目的是为了将匿名嵌入的类型所具有的的功能扩展到当前定义的类型中。

```
// 反例
type MyStack struct {
    Mylist // 不符合：此匿名嵌入暴露了内部的实现
}

// 正例
type MyStack struct {
    list Mylist // 符合：不使用匿名嵌套，使得嵌入实现的方法不会被暴露出去
}
```

3.9.2、接口

3.9.2.1、从使用者角度设计接口

应该从使用者而不是实现者的角度设计接口，实现者应返回具体类型。

```
type Producer interface {DoSomething()} // interface 不能在实现的地方（同一个文件里）定义，应该单独的定义在某个位置

// 正例
package impl

type MyProducer struct{...}

func (mp MyProducer)DoSomething(){...}
func NewProducer() MyProducer {
    return MyProducer{}
}
```

3.9.2.2、避免接口过大

go语音推荐使用组合的方式来写程序。go开发者一般会嵌入其他已有接口类型的方式来构建新的接口。

小结构体的主要优势：

- 接口越小，抽象程度越高，越彻底，被人接受的程度越好，其实接口的实现和现实中情况是一样的，最小的接口就是`interface{}`
- 小接口有较少的方法，一般仅仅一个方法。要实现这个接口，开发者仅仅实现一个或少数几个方法就可以了。单元测试尤为重要，可以付出较少的成本来验证你的程序是否有bug。
- 职责单一，容易组合。

可以参考io标准库的设计。

3.9.2.3、避免类型可能不为空的接口变量和nil直接进行比较

在go语言底层，interface定义的变量有两个成员变量，一个类型和一个值，只有当两个都为nil的情况下的这个接口变量才是nil。

如果某个具体的对象赋值给接口变量，无论赋值的对象是不是nil，这个接口变量都不可能是nil，因此不存在在nil值赋值给接口变量。

- 确保赋值给接口变量的对象的指针有效，否则应赋予无类型的nil值
- 必须使用error作为函数的错误返回值类型，对非错误必须返回无类型的nil。

```
// 正例
if err := do() ;err != nil {
    return err
}
return nil
```

3.9.3、类型断言

3.9.3.1、必须处理类型断言的失败

类型断言可以判断接口变量封装的是否是期望类型；也可以判断接口变量封装的类型是否实现了目标接口。在对接口变量进行类型断言时，必须判断断言是否成功，如果失败仍执行会panic。

类型断言使用场景有两种：if ok 和 switch，switch分支使用default分支来处理其他情况

```
if v ,ok := a.(string) ; !ok {
    // 错误处理
}

switch t := t.(type) {
case int:
case bool:
default: // 必须使用默认分支来处理失败的情况
}
```

4.0、包

4.0.1、使用internal目录避免内部公开的Api对外暴露

在设计项目或模块业务代码时，应该合理的使用internal目录，把内部公开的代码放在internal中，以提供对外合理的api

比如 example/a/internal只能被example/a开始的包导入，不能被example/e、 example/f等其他包导入。

4.0.2、禁止使用.来简化导入包

这种写法不利于阅读

```
// 反例
import . "example/a"  // 不符合：使用了 . 来简化导入包

// 例外
在测试文件_test.go中，为了避免循环依赖时，才能使用
```

4.0.3、禁止使用相对路径导包

```
// 反例
import "../net"
```

4.0.4、导入包的顺序按稳定度排序

建议导入的顺序是： 标准库、第三方库、本项目的库

4.0.5、导入包时名称未冲突的情况下应避免使用别名

```
// 反例
import (
    rtace "runtime/trace" // 不符合：未冲突，尽量避免使用别名
)

// 正例
import (
    rtace "runtime/trace"
    xtrace "golang.net/x/trace"
)
```

4.1、错误和异常处理

4.1.1、使用恰当的错误处理机制

错误处理机制有:

- `in-band` 错误提示：通过返回的内容，区别是正常值还是错误值
- 使用异常panic机制
- 使用error接口的方式返回错误值 （推荐使用）

4.1.2、错误处理

4.1.2.1、确保正确处理函数的错误返回值。

任何时候都不要忽略返回的error类型

```
// 反例
b, _ : = get() // v ,err
b.bar() // 未正常处理错误，b可能为nil

// 正例
if b,err := get() ;err != nil {
    b.bar()
}
```

4.1.2.2、错误信息不应该大写，或者以标点结尾。

错误字符串要求：除专有名词或者缩写词外，避免字符串以大写字面开头，或以标点结尾

```
// 反例
fmt.Errorf("Something bad") // 不符合，上下文拼接后，字符串中间会有大写字面开头的场景

// 正例
fmt.Errorf("something good")
```

4.1.3、异常处理

4.1.3.1、包外可见的函数禁止向外抛出panic

包外可见的函数禁止向外抛出panic，如果导出函数本身或者调用其他函数可能抛出panic，需要内部recover并转换成error返回给外包调用者。

4.1.3.2、确保发生异常时程序尝试恢复到合理的状态并记录日志

当函数/方法发生异常时，一般对象需要恢复到原来状态，安全关键的对象必须恢复到原来的状态。保持对象一致性常用手段包括：

> - 输入校验，如校验方法的参数
> - 调整逻辑顺序，使可能发生异常的代码在对象修改之前执行。
> - 对业务操作失败时回滚
> - 对临时副本对象操作，直到操作完成，才把更新提交到原始对象上。
> - 编码修改对象状态

4.2、并发

4.2.1、避免数据竞争

map 、slice不是并发安全的，所以在并发下共享数据访问不存在数据竞争（不单指map和slice）。并发时优先推荐消息传递数据，而不是共享内存。

go1.9 之后可以用sync.Map满足并发安全的map需求.

可以使用go 的race工具分析代码是否存在数据竞争。

```
go test -race pkgname
go run -race src.go
go build -race cmds
go install -race pkgname
```

4.2.2、避免goroutine被永久阻塞

避免goroutine永久阻塞导致内存泄漏；goroutine永久阻塞一般是由于锁或者chan未正确使用造成的。

确保select的不会被长时间或者永久阻塞，可添加default分支：

需要遵循以下要求：

> - 确保读取chan时，chan存在数据，否则会被阻塞。
> - 不向值为nil（var ch chan int）的chan发送或接收数据，否则将导致goroutine永久阻塞
> - 避免死锁导致goroutine阻塞。

4.2.3、chan类型作函数参数时限定类型

chan类型作为函数参数传递时限定类型，根据函数创建最新权限得channel，以增加代码的可读性。

```
// 正例
func readOnly (in <-chan int) {

}

func writeOnly(out chan<- int){

}

func readAndWrite(ch chan int){

}
```

4.2.4、使用chan时确保对chan的操作有效

确保chan使用时已被初始化（使用make申请过空间），禁止对nil和closed的chan进行操作，否则可能导致以下问题

> - 关闭*closed*的*channel*会产生*panic*
> - 关闭 *nil* 的 *chan* 会产生*panic*
> - 向*closed*的*chan*发送数据会*panic*
> - 当*chan*没数据时，取数据会返回零值。必须通过*if ok*判断是否正常取出数据。

```
// 正例
select {
case _,ok := <- ch:
    if !ok {
        return
    }
}
```

4.2.5、禁止拷贝锁和同步的对象

使用锁机制时，应避免死锁。复制锁或同步的对象本身没有同步保护，因此复制结果可能不完整（不是有原值得快照），即使锁是完整的它也会和已有锁存在冲突。

应确保：

> - 正确的使用锁，程序中不存在死锁；注意使用相同的顺序请求和释放锁来避免死锁。
> - 确保加锁的代码范围合理；禁止在循环语句中使用*defer*语句释放锁。禁止多次释放同一个锁
> - 禁止拷贝包含锁或同步的数据结构（一般在*sync*包里定义了锁或同步的类型结构），只能通过指针或接口等引用类型来传递此类数据结构的引用。

```
// 反例
for i := 0 ; i < len(s) ; i++ {
    m.Lock()
    defer m.Unlock() // 不符合：不能在循环中使用defer来释放锁
}

// 正例
m.Lock()
defer m.Unlock()
for i := 0 ; i < len(s) ; i++ {

}

type Counter struct{
   sync.Mutex
   n int64
}
// 反例
func (c Counter) Value() int64 { // 不符合：方法接受者为值类型，当它被调用时，Couner被复制。简介导致Counter中的Mutex被复制

}

// 正例
func (c *Counter) Value() int64 { // 有锁的对象，使用指针传递。

}
```

4.3、输入输出

4.3.1、临时文件使用完必须及时删除

程序员经常会创建临时文件，如在/var/tmp。这样的目录可能会被定期清理。但未被清理前还是可访问的。每个程序都有责任确保删除已使用完毕的临时文件。

4.3.2、创建文件时必须显示指定合适的文件访问权限

多用户的文件系统是通过权限和许可模型来保护文件可访问的。多用户系统中的文件通常归属于某个特定的用户。攻击者可能通过其他用户攻击。因此创建文件时就为其指定访问权限，以防止未授权的文件访问。

最好指定0600，或者0644的权限

```
// 反例
f ,err := os.Creat("/tmp/test.txt") // 不符合：没有显示指定文件权限

// 正例
f ,err := os.OpenFile("/tmp/test.txt",os.O_CREATE|OS_WRONLY|OS_TRUNC,0600)
```

4.3.3、确保检验文件路径前对其标准化

绝对路径名或者相对路径名中可能包含文件链接（软链接、硬链接、快捷方式、影子文件、别名等）。或者包含特殊字符（如.或..），者使得验证文件路径变得困难。如不检验，工具者可以通过路径，达到攻击的目的。

在文件操作之前，必须对文件路径进行验证，对文件路径标准化，会使得验证文件路径更简单安全。

```
// 反例
func validate(path string) bool {
    pattern := "^/opt/pwm"  // 不符合：未把文件路径标准化，当path为
/opt/pwd/../../etc/passwd时,攻击者可以绕过验证。
    reg := regexp.MustComplie(pattern)
    return reg.MatchString(path)
}

// 正例
func validate(path string) bool {
    realPath,err := filepath.Abs(path)
    if err != nil {
        return false
    }
    pattern := "^/opt/pwm"
    reg := regexp.MustComplie(pattern)
    return reg.MatchString(realPSath)
}
```

4.3.4、确保安全地从压缩包提取文件

解压文件时有两个问题需要避免

- 提取文件的标准路径是否落在解压目录之外
- 提取的文件是否消耗过多的系统资源

需要增加文件数量、文件大小、路径标准化等检验操作

4.3.5、禁止直接使用外部数据构造格式化字符串

当攻击者可以直接控制格式化字符串时，可导致信息泄漏、拒绝服务、系统功能异常等风险。

```
// 反例
func checked(user string) {
    msg := fmt.Sprintf("%s ddd",user)
    fmt.Printf(msg)  // 不符合：msg存在为验证的外部数据，存在格式化漏洞，当指定外部格式
为%p，%d会导致格式化出现非预期的结果
}
```

4.3.6、禁止在日志中保存敏感数据

在日中禁止输出口令、秘钥和其他保护信息。口令包括明文口令和密文口令，对于敏感信息采用以下方法：

- 不在日志中打印敏感信息
- 不将敏感信息输出到控制台和串口
- 如因特殊原因必须打印日志，使用固定长度的*代替敏感信息
- 序列化或者输出敏感信息应该先加密后，输出到外部设备或者文件中
- 禁止在错误或异常处理中泄漏敏感信息

4.4、外部数据校验

4.4.1、对所有外部数据进行合法性校验

编程人员处理外部数据过程中必须时刻保持这种思维意识，不要做任何外部数据符合预期的假设。外部数据必须经过严格判断后才能使用

外部数据来源包括但不限于：网络、用户输入、命令行、文件（包括程序配置文件）、环境变量、用户态数据（对于内核程序）、进程间通信（包括管道、消息、共享内存、socket、RPC等）、API参数、全局变量等

典型场景如下：

作为数组索引

将不信任的数据作为数组索引，可能导致超出数组上线，从而造成内存非法访问。

作为内存偏移地址

作为内存分配的尺寸参数

作为循环条件

作为除数

作为命令行参数

作为数据库查询语句参数

作为输入、输出格式化字符串参数

作为文件路径

输入校验包括但不限于：

- API接口参数合法性
- 检验数据长度
- 检验数据范围
- 校验数据类型和格式
- 检验输入只包含可接收的字符（'白名单形式'），尤其是需要注意一些特殊字符

原则：信任边界、外部数据校验

4.4.2、禁止直接使用外部数据拼接sql语句

SQL注入是指外部数据构造的SQL语句所代表的数据库操作与预期不符。这样的操作可能导致信息泄漏或者数据被篡改

防护措施如下：

- 使用参数化查询：最有效的防护手段，但对sql语句中的表名、字段名等不适用
- 对外部数据进行白名单校验：适用于拼接SQL中的表名、字段名
- 对外部数据中SQL注入相关的特殊字符进行转义：使用与拼接sql场景

```
// 反例
func queryById(id,pwd string){
    // 不符合：直接使用sql拼接
    var sqlStr = "select name from user where id = '"+id+"' and pwd='"+pwd+"'"
}

// 正例
func queryById(id,pwd string){
    // 不符合：直接使用sql拼接
    var sqlStr = "select name from user where id = ? and pwd= ?"
    db.Query(sqlStr,id,pwd)
}
```

4.4.3、禁止直接使用外部数据拼接命令参数

使用未经检验的不可信输入作为系统命令的参数或命令的一部分，可能会导致命令注入漏洞。

需要注意以下：

- 命令执行的字符串不要取拼接输入的参数。如果必须拼接时，需要对输入参数进行白名单过滤。
- 对传入的参数数据要做类型校验，例如整数数据。
- 保证格式化字符串的正确性。例如int类型的参数拼接要用%d，不要用%s。

4.4.4、确保数组和slice的下标不越界

注意注意以下两点：

- 对外部数据输入或校验的值，如果用于数组或者slice的索引，确保此值在数组和slice的索引范围内
- 待入并验证循环边界的范围，以保证不越界访问数组和切片

```
// 反例
for i := 0 ; i < 100 ; i++ { // 不符合：可能导致数组越界
    buf[i] = buf[i+1]
}

// 正例
for i := 0 ; i < len(buf)-2 ; i++ { // 不符合：可能导致数组越界
    buf[i] = buf[i+1]
}
```

4.5、资源管理

4.5.1、在性能敏感的情况下建议采用池化技术优化资源使用

一般而言，重用一个已经创建的对象比创建一个新对象快，除非确实需要重新创建。

go语音申请的变量内存需要gc。gc回收需要额外的开销。大量的内存回收可能会导致GC期间CPU消耗过多。业务处理能力下降。

其原则是尽可能少申明变量：

- 局部变量尽量利于
- 小变量合并，如果小变量过多，可以把这些变量放在一个结构体内。降低扫描和回收时间
- 高并发任务中通过goroutine池，避免调度和GC带来的冲击；goroutine虽然轻量级，但对于高并发的轻量级任务下，频繁创建groutine来执行，效率也非常低。
- 在合理的情况下使用struct{}空结构体。来避免数据值所占用的空间，如：配合map实现set集合。chan传递空消息等。

4.5.2、避免资源泄漏

在程序每个可能走到的流程中，应确保完整的释放文件句柄、DB连接等资源。特别是在异常发生时，若申请的资源（如文件句柄、数据库连接、内存资源）未得到释放，会引起资源异常占用。

实际操作中能够用defer释放的尽量用defer释放。不能强制使用defer

应确保：

- 主程序结束，所有的goroutine默认都会退出。要有安全的通知goroutine结束机制如（context包）
- 主程序没有结束，有些goroutine已经不在使用，要确保及时退出。

```
// 正例
f,err := os.OpenFile("/test.txt",OS_RDONLY,0644)
if err != nil {
    return
}
defer func(){
    err = f.Close()
    if err != nil {
        // 异常处理
    }
}()
```

4.5.3、合理使用defer

不要再defer中修改返回值（尤其是具名返回的情况下），使得代码不易维护。

避免在循环中使用defer

推荐使用defer的两种场景

- 资源释放
- 如果代码可能导致pannic，使用defer func recover

```
func tryChange() int {
    var result  int
    defer func() {
       result++ // 不符合：试图修改返回值（实际并不能）
    }()
}
```

4.5.4、禁止重复释放资源

重复释放一般处于错误的流程判断中。

要求：

- 禁止重复关闭释放的文件、数据库连接、或网络连接等资源。
- 禁止重复释放chan等系统内置类型的对象资源

```
func foo(c chan int) {
    defer close(c)
    for {
        err := get()
        if err != nil {
            c <- 0
            close(c) // 不符合：重复释放channel
        }
    }
}
```

4.5.5、make申请的slice、map时，根据预估或校验范围大小来申请合适的内存。

为避免扩容带来的性能消耗，在初始化slice和map时，根据预估范围来申请合适的容量。

```
// 正例
var mp = make(map[string]string 1000)
```

4.5.6、避免过多的time.After函数调用导致消耗大量的资源

如果在某个时间段频繁的多次调用，则可能导致很多未过期的Timer值，从而导致大量的内存和计算消耗。

```
// 反例
func longRunning(message <-chan string){
    for {
        select{
            case <-time.After(time.Minute):
                return // 会一直堆积，造成资源消耗
            case msg := <- message:
                do()
        }
    }
}

// 正例
func longRunning(message <-chan string){
```

```
    timer := time.NewTimer(time.Minute)
    for {
        select{
                case <-time.C:
                    return // 会一直堆积，造成资源消耗
                case msg := <- message:
                    do()
                    if !timer.stop(){
                        <-timer.C
                    }
         }
        // 必须重置以复用
        timer.Rest(time.Mintue)
     }
}
```

4.6、内存访问

4.6.1、禁止解引用空指针

go中，对值类型，声明即默认初始化。对引用类型，申明后其值是空值（nil）。使用引用类型前，必须对其有效的构建并初始化好。

```
// 反例
func main() {
    var s *student
    if command {
        s = &student{}
    }
    s.Foo() // 不符合：s可能为nil
}

// 正例
func main() {
    var s *student
    if command {
        s = &student{}
    }
    if s == nil {
        return
    }
    s.Foo() // 不符合：s可能为nil
}
```

4.6.2、禁止解引用空的接口和内置引用类型变量

对chan、map内置引用类型变量，访问元素前，必须使用make显式的初始化。

对interface类型变量，在解引用或调用变量之前，应判断接口变量不为空。

```
// 反例
// 不符合：未使用make显式初始化
var mp map[string]*student
var ch chan int
```

4.6.3、使用[]byte而不是string存储敏感数据、敏感数据使用完后应立即清0

使用[]byte存储敏感数据，可以降低敏感数据泄漏风险

```
// 正例
buf := make([]byte,1024)
_,err := f.read(buf)
```

4.6.4、避免使用unsafe包

uintptr是一个可容纳指针的指数，表示对象的地址。

使用时应确保：

> - *uintptr* 值转换*unsafe.Pointer*时地址有效
> - 解引用*unsafe.Pointer*时不存在内存越界访问

4.7、cgo

4.7.1、最小化使用cgo范围

cgo调用开销非常大，需要谨慎使用，如果能用纯go解决的就不要用cgo

4.7.2、使用C虚拟包辅助函数C.CString和C.CBytes返回的类型转换参数使用完后需要手动释放

```
// 正例

// #include "stdio.h"
import "C"
import "unsafe"

func main() {
    cs := C.CString("hello")
    defer C.free(unsafe.Pointer(cs)) //需要手动释放
    C.puts(sc)
}
```

4.7.3、避免将Go内存指针直接传入C函数使用

CGO时go语音和c语音实现互通。在需要将go的字符串传入C语音时，先通过C.CString将go语音的字符串对应的数据复制到C语音的内存空间上。确保安全的使用，但会导致效率下降

```
// 正例
package main
/*
#include "stdio.h"
#include "stdlib.h"

void printString(const char* s){
    printf("%s",s)
}
*/
```

```
import "C"
import "unsafe"

func main() {
    cs := C.CString("hello")
    defer C.free(unsafe.Pointer(cs)) //需要手动释放
    C.printString(sc)
}
```

4.7.4、避免使用C代码接管系统的信号

应尽量避免C代码接管系统的信号。signal应该go的信号处理来接管

4.7.5、避免直接将C代码嵌入在Go文件中

直接经C代码嵌套在go中，可读性差。应该将C代码打包程静态库/动态库的方式调用。

4.7.6、避免在CGO中使用C代码调用Go函数

防止下面情形

- 被C函数调用的go函数中再调用C函数
- 被Go函数调用的C函数中再调用Go函数

4.8、其他

4.8.1、禁止使用math/rand包提供的函数产生安全用途的伪随机数

math/rand 包提供生成的伪随机数，不能保证其产生的随机数序列质量。

crypto/rand （推荐使用）包中提供了密码安全学伪随机数生成器。提供reader变量，在Unix中reader变量读取/dev/urandom生成随机数，在linux系统中使用getrandom生成随机数，在windows系统中使用CryptGenRandom API生成随机数

安全用于包括但不限于以下几种

- 会话标识的SessionID
- 挑战算法中的伪随机数生成
- 验证码中的随机数生成
- 用于密码算法用途（例如：生成IV，salt值、秘钥等）的随机数生成

4.8.2、禁止代码中包含公网地址

对产品发布的软件（包含软件包、补丁包）中的公网地址（包含公网ip、公网url地址、域名、邮箱地址）要求如下

- 禁止包含用户界面不可见、或者产品资料为公开的公网地址
- 已公开的公网地址禁止写在代码和脚本中，可存在在配置文件或者数据库中

> *import* 包中的*url* 是例外

4.8.3、删除无效或者永不执行的代码

注重DT覆盖率，无效或者永不执行的代码通常是编程错误的结果，应该主动识别整理和消除。

> 公共库中未使用的导出函数和变量不违反该准则

4.8.4、不用代码直接删除掉，不要注释掉

被注释掉的代码，无法被正常维护。当企图恢复这段代码时，极有可能被引入被忽略的缺陷

正确的做法是，不需要的代码直接删除掉。若需要时可以通过版本管理工具恢复

> *cgo*中的注释是例外

4.8.5、避免调用os.Exit()和runtime.Goexit()退出函数

程序应该优雅的退出，具备有序的退出机制，并在真正退出前做好善后的工作。

除main函数外，禁止在任何地方调用os.Exit()函数，调用os.Exit()会立即终止，终止前无法执行相应的defer，可能导致某些资源难以有效的清理。

正确的做法是通过错误值传递到上一级调用者。结束前确保资源有序的释放，禁止下级程序直接结束进程。

5、附录

5.1、go语音内置标识符

分类	内置标识符
类型	bool、byte、complex64、complex128、error、float32、float64、int、int8、int16、int32、int64、rune、string、uint、uint8、uint16、uint32、uint64、uintptr
常量	true、false、iota
零值	nil
函数	append、cap、close、complex、copy、delete、imag、len、make、new、panic、print、println、real、recover

5.2、参考资料

Effective Go

GO code review Comment

Uber go style Guide

Gitlab Go guide

go proverbs