golang 编码规范

一、 默认规范

1. 注释

推荐使用“//”注释。注释必须是简明扼要，表述准确。

1.1. 包注释

每个包都应该有一个包注释，包如果有多个go文件，就只需要在入口文件写包注释。 一般以Package开头。

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is Governed by a BSD-style
// license that can be found in the LICENSE file.

// Package strings implements simple functions to manipulate strings.
package strings
```

1.2. 函数和结构体注释

第一行写概况，并且使用被声明的名字作为开头。

```
// Compile parses a regular expression and returns, if successful, a Regexp
func Compile(str string) (regexp *Regexp, err error) {

// Request represents a request to run a command.
type Request struct { ...
```

1.3 详细注释

处理复杂逻辑时需要详细注释，可在package内创建doc.go详细描述。 如src/strconv/doc.go

```
// Copyright 2015 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// Package strconv implements conversions to and from string representations
// of basic data types.
//
// Numeric Conversions
//
// The most common numeric conversions are Atoi (string to int) and Itoa (int to
string).
//
//  i, err := strconv.Atoi("-42")
//  s := strconv.Itoa(-42)
```

```
//
// These assume decimal and the Go int type.
//
// ParseBool, ParseFloat, ParseInt, and ParseUint convert strings to values:
//
//  b, err := strconv.ParseBool("true")
//  f, err := strconv.ParseFloat("3.1415", 64)
//  i, err := strconv.ParseInt("-42", 10, 64)
//  u, err := strconv.ParseUint("42", 10, 64)
//
// The parse functions return the widest type (float64, int64, and uint64),
// but if the size argument specifies a narrower width the result can be
// converted to that narrower type without data loss:
//
//  s := "2147483647" // biggest int32
//  i64, err := strconv.ParseInt(s, 10, 32)
//  ...
//  i := int32(i64)
//
// FormatBool, FormatFloat, FormatInt, and FormatUint convert values to strings:
//
//  s := strconv.FormatBool(true)
//  s := strconv.FormatFloat(3.1415, 'E', -1, 64)
//  s := strconv.FormatInt(-42, 16)
//  s := strconv.FormatUint(42, 16)
//
// AppendBool, AppendFloat, AppendInt, and AppendUint are similar but
// append the formatted value to a destination slice.
//
// String Conversions
//
// Quote and QuoteToASCII convert strings to quoted Go string literals.
// The latter guarantees that the result is an ASCII string, by escaping
// any non-ASCII Unicode with \u:
//
//  q := strconv.Quote("Hello, 世界")
//  q := strconv.QuoteToASCII("Hello, 世界")
//
// QuoteRune and QuoteRuneToASCII are similar but accept runes and
// return quoted Go rune literals.
//
// Unquote and UnquoteChar unquote Go string and rune literals.
//
package strconv
```

2、命名

使用短命名，因为长名字并不会使得事物更易读，文档注释会比格外长的名字更有用。 需要导出的任何类型必须以大写字母开头。

2.1. 包名

包名应该为小写单词，而且是全word(非缩写)，不推荐下划线或者混合大小写。

2.2. 变量名

全局变量：驼峰式，可导出的使用大写字母开头,应该是名词或名词短语 参数传递：驼峰式，小写字母开头,应该是名词或名词短语 局部变量：下划线风格命名

2.3. 常量

提供给外部使用的常量均使用全部大写字母组成，并使用下划线分词：

```
const APP_VER = "1.0"
```

如果是枚举类型的常量，需要先创建相应类型：

```
type Proto string

const (
    HTTP  Proto = "http"
    HTTPS Proto = "https"
)

type TimeDay int

const (
  Sunday       TimeDay = iota //  0
  Monday                      //  1
  Tuesday                     //  2
  Wednesday                   //   3
  Thursday                    //   4
  Friday                      //   5
  Partyday                    //   6
  numberOfDays                // this constant is not exported
)
```

2.4 结构体名

结构体名字：驼峰式，可导出的使用大写字母开头,应该是名词或名词短语 但是在结构体方法中，结构体参数命名使用小写单词首字母，尽量不要使用self，this等

2.5. 接口名

单函数的接口用 函数+“er” 命名，如：Reader，Writer

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

2个函数的接口，组合命名

```
type WriteFlusher interface {
    Write([]byte) (int, error)
    Flush() error
}
```

3个以上函数的接口名，类似于结构体名

```
type Car interface {
    Start([]byte)
    Stop() error
    Recover()
}
```

2.6 单元测试文件命名

单元测试文件名命名规范为：source_test.go

3. import package格式

对标准包，程序内部包，第三方包进行分组。

```
import (
    "encoding/json"          //标准包
    "strings"

    "xxx.com/project/models"       //内部包
    "xxx.com/project/utils"

    "github.com/go-sql-driver/mysql"     //第三方包
)
```

引用包时不要使用相对路径。

```
import “../net”
```

4.流程控制

4.1. if

if接受初始化语句，约定如下方式建立局部变量。

```
if err := file.Chmod(0664); err != nil {
    return err
}
```

4.2. for

采用短声明建立局部变量。

```
for i := 0; i < 10; i++ {
}
```

4.3. range

```
for _, value := range array {
  //注意对于slise结构value的地址是相同，要使用copy值
}
```

4.4. return

尽早return

```
f, err := os.Open(name)
if err != nil {
    return err
}

defer f.Close()
d, err := f.Stat()
if err != nil {
    return err
}

codeUsing(f, d)
```

5.函数

函数一般都有error或者更多返回值，使用带有返回值的函数声明更利于理解。如：

```
func nextInt(b []byte, pos int) (value, nextPos int, err error) {
// code
}
```

6.错误处理

不要在逻辑代码中使用panic error作为函数的值返回,必须对error进行处理。一般会根据业务情况对error进行错误包装 (Error Wrapping)，很多场景需要包含更多信息如：code，cause，metadata等。

```
// Bad

if err != nil {
    // error handling
} else {
    // normal code
}

// Good
if err != nil {
    // error handling
    return // or continue, etc.
}
// normal code
x, err := f()
if err != nil {
    // error handling
    return
}
// use x
```

7.使用defer释放资源

使用 defer 释放资源，诸如文件和锁。

```
// Bad
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

// Good
```

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

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

p.count++
return p.count
```

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.`

8. 主函数退出

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 don't 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。

不推荐

```
// Bad

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 := ioutil.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
  return string(b)
}

// Good
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 := ioutil.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

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 := ioutil.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
  // ...
```

```
}

// Good

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 := ioutil.ReadAll(f)
  if err != nil {
    return err
  }
  // ...
}
```

9.尽量少使用init()

尽可能避免使用`init()`。当`init()`是不可避免使用时，应考虑以下：

1. 无论程序环境或调用如何，都要完全清楚。
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。

二、 实践总结

2.1 格式和规范

2.1.1 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
import "a"
import "b"

// Good
import (  "a"  "b" )
```

这同样适用于常量、变量和类型声明：

```
// Bad
const a = 1
const b = 2
var a = 1
var b = 2
type Area float64
type Volume float64

// Good
const (  a = 1  b = 2 )
var (  a = 1  b = 2 )
type (  Area float64  Volume float64 )
```

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

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

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

2.1.2 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
import (
          "fmt"
          "os"
          "go.uber.org/atomic"
          "golang.org/x/sync/errgroup"
)

// Good
import (
          "fmt"
          "os"

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

2.1.3 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 Package Names 和 Go 包样式指南.

2.1.4 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`.

2.1.5 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
import (
   "fmt"
   "os"

    nettrace "golang.net/x/trace"
)

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

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

2.1.6 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
func (s *something) Cost()
{
  return calcCost(s.weights)
}

type something struct{ ... }

func calcCost(n []int) int {...}

func (s *something) Stop() {...}

func newSomething() *something {
  return &something{}
}

// Good

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 {...}
```

2.1.7 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

for _, v := range data {
if v.F1 == 1 {
  v = process(v)
  if err := v.Call(); err == nil {
    v.Send()
  } else {
    return err
  }
} else {
  log.Printf("Invalid v: %v", v)
}
}

// Good
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()
}
```

2.1.8 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 }`

2.1.9 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

var _s string = F()
func F() string { return "A" }

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

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

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。

```
// Bad

// foo.go

const (
  defaultPort = 8080
  defaultUser = "user"
)

// bar.go

func Bar() {
  defaultPort := 9090
```

```
  ...
  fmt.Println("Default port", defaultPort)

  // We will not see a compile error if the first line of
  // Bar() is deleted.
}


//Good
// foo.go

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

2.1.10 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。 它应该在对用户不利影响的情况下完成这项工作（另请参见：`避免在公共结构中嵌入类型` Avoid Embedding Types in Public Structs）。

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。

```go
// Bad
type A struct {
    // Bad: A.Lock() and A.Unlock() are
    //      now available, provide no
    //      functional benefit, and allow
    //      users to control details about
    //      the internals of A.
    sync.Mutex
}

//Good
type countingWriteCloser struct {
    // Good: Write() is provided at this
    //       outer layer for a specific
    //       purpose, and delegates work
    //       to the inner type's Write().
    io.WriteCloser

    count int
}

func (w *countingWriteCloser) Write(bs []byte) (int, error) {
    w.count += len(bs)
    return w.WriteCloser.Write(bs)
}

// Bad
type Book struct {
    // Bad: pointer changes zero value usefulness
    io.ReadWriter

    // other fields
}

// later

var b Book
b.Read(...)  // panic: nil pointer
b.String()   // panic: nil pointer
b.Write(...) // panic: nil pointer

// Good
type Book struct {
    // Good: has useful zero value
    bytes.Buffer

    // other fields
```

```
}

// later

var b Book
b.Read(...)  // ok
b.String()   // ok
b.Write(...) // ok


// Bad
type Client struct {
    sync.Mutex
    sync.WaitGroup
    bytes.Buffer
    url.URL
}

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

2.1.11 使用字段名初始化结构体

初始化结构体时，应该指定字段名称。现在由 `go vet` 强制执行。

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

2.1.12 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

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

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

2.1.13 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。但是在与前端restful接口返回nil时，前端不一定能很好的处理nil，这点需要注意。

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

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

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

// Good
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。

2.1.14 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	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，则不应尝试缩小范围。

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

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

```
  return err
}

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

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

fmt.Println(cfg)
return nil
```

2.1.15 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)
```

2.1.16 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"`

2.1.17 一些初始化例子

- 结构体初始化

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，写test时字段名可以省略。

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

- 省略结构体重的默认值

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。 也就是直接使用默认值。

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},
  // ...
}
```

- 对默认值结构体声明使用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， 并与我们更喜欢的 declare empty slices方式相匹配。

- 初始化结构体引用

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 = map[T1]T2{} m2 map[T1]T2 )`	`var ( m1 = make(map[T1]T2) m2 = make(map[T1]T2))`
m1 读写安全; m2 在写入时会 panic, 声明和初始化看起来非常相似的。	m1,m2 读写安全; 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)`

- interface的初始化

在编译时验证接口的符合性。这包括：

1. 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
2. 实现同一接口的(导出和非导出)类型属于实现类型的集合
3. 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
type Handler struct {
  // ...
}



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

// Good
type Handler struct {
  // ...
}

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.

2.1.18 slice和map的拷贝

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

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

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

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

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

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

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

- 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
```

```go
type Stats struct {
  mu sync.Mutex
  counters map[string]int
}

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

  return s.counters
}

// snapshot is no longer protected by the mutex, so any
// access to the snapshot is subject to data races.
snapshot := stats.Snapshot()

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

2.1.19 避免使用全局变量

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

```go
// Bad

// sign.go

var _timeNow = time.Now
```

```go
func sign(msg string) string {
  now := _timeNow()
  return signWithTime(msg, now)
}

// Good

// 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)
}

// Bad
// 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))
}

// Good
// sign_test.go

func TestSigner(t *testing.T) {
  s := newSigner()
  s.now = func() time.Time {
    return someFixedTime
  }

  assert.Equal(t, want, s.Sign(give))
}
```

不要使用package内部全局变量

```
// Bad

type Config struct {
    // ...
}

var _config Config

func init() {
    // Bad: based on current directory
    cwd, _ := os.Getwd()

    // Bad: I/O
    raw, _ := ioutil.ReadFile(
        path.Join(cwd, "config", "config.yaml"),
    )

    yaml.Unmarshal(raw, &_config)
}

// Good
type Config struct {
    // ...
}

func loadConfig() Config {
    cwd, err := os.Getwd()
    // handle err

    raw, err := ioutil.ReadFile(
        path.Join(cwd, "config", "config.yaml"),
    )
    // handle err

    var config Config
    yaml.Unmarshal(raw, &config)

    return config
}
```

2.1.20 使用option

Option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。

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

// package db

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


// Good

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

```
// Bad
// 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)

// Good
// 只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：

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

2.2 性能相关

2.2.1 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`

2.2.2 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`

2.2.3 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

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 capacity提示并不保证完全的抢占式分配，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
// m是在没有大小提示的情况下创建的；  在运行时会有资源占用
m := make(map[string]os.FileInfo)

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

// Good
// m是有大小提示创建的；在运行时占用更少的资源。实际开发中不要忽视err。
files, _ := ioutil.ReadDir("./files")

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

- 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
// BenchmarkBad-4    100000000    2.48s
for n := 0; n < b.N; n++ {
  data := make([]int, 0)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}

// Good
// BenchmarkGood-4   100000000    0.21s
for n := 0; n < b.N; n++ {
  data := make([]int, 0, size)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
```

2.2.4 使用sync.Pool

Go 语言从 1.3 版本开始提供了对象重用的机制，即 sync.Pool。sync.Pool 是可伸缩的，同时也是并发安全的，其大小仅受限于内存的大小。sync.Pool 用于存储那些被分配了但是没有被使用，而未来可能会使用的值。这样就可以不用再次经过内存分配，可直接复用已有对象，减轻 GC 的压力，从而提升系统的性能。

sync.Pool 的大小是可伸缩的，高负载时会动态扩容，存放在池中的对象如果不活跃了会被自动清理。

sync.Pool 的使用方式非常简单：

- 结构体对象

只需要实现 New 函数即可。对象池中没有对象时，将会调用 New 函数创建。

```
var studentPool = sync.Pool{
    New: func() interface{} {
        return new(T)
    },
}
```

```
s := studentPool.Get().(*T)
json.Unmarshal(buf, s)
studentPool.Put(s)
```

其中：`Get()` 用于从对象池中获取对象，因为返回值是 `interface{}`，因此需要类型转换。

`Put()` 则是在对象使用完毕后，返回对象池。

- bytes.Buffer

```
var bufferPool = sync.Pool{
  New: func() interface{} {
    return &bytes.Buffer{}
  },
}

var data = make([]byte, 10000)

func BenchmarkBufferWithPool(b *testing.B) {
  for n := 0; n < b.N; n++ {
    buf := bufferPool.Get().(*bytes.Buffer)
    buf.Write(data)
    buf.Reset()
    bufferPool.Put(buf)
  }
}

func BenchmarkBuffer(b *testing.B) {
  for n := 0; n < b.N; n++ {
    var buf bytes.Buffer
    buf.Write(data)
  }
}
```

测试结果

```
BenchmarkBufferWithPool-8     8778160     133 ns/op        0 B/op    0 allocs/op
BenchmarkBuffer-8              906572    1299 ns/op    10240 B/op    1 allocs/op
```

2.2.5 使用atomic代替锁

开发中经常要使用锁来保证一致性，锁的性能不高，在某些场景下使用atomic代替。

```
package main

import (
  "sync/atomic"
```

```
  "time"
)

func loadConfig() map[string]string {
  return make(map[string]string)
}

func requests() chan int {
  return make(chan int)
}

func main() {
  var config atomic.Value // holds current server configuration
  // Create initial config value and store into config.
  config.Store(loadConfig())
  go func() {
    // Reload config every 10 seconds
    // and update config value with the new version.
    for {
      time.Sleep(10 * time.Second)
      config.Store(loadConfig())
    }
  }()
  // Create worker goroutines that handle incoming requests
  // using the latest config value.
  for i := 0; i < 10; i++ {
    go func() {
      for r := range requests() {
        c := config.Load()
        // Handle request r using config c.
        _, _ = r, c
      }
    }()
  }
}
```

2.2.6 使用worker-pool

很多场景下，golang可以支持千万级的协程数量，但某些场景下，我们需要控制协程数据又要保证高效处理业务，这个时候一般使用worker-pool。worker-pool需要做好任务分发机制和多协程间的同步。

可参考：https://github.com/bytedance/gopkg/tree/develop/util/gopool

三、安全编程

1. 内存管理

1.1【必须】切片长度校验

- 在对slice进行操作时，必须判断长度是否合法，防止程序panic

```
// bad: 未判断data的长度，可导致 index out of range
func decode(data []byte) bool {
  if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z' && data[4] ==
'E' && data[5] == 'R' {
    fmt.Println("Bad")
    return true
  }
  return false
}

// bad: slice bounds out of range
func foo() {
  var slice = []int{0, 1, 2, 3, 4, 5, 6}
  fmt.Println(slice[:10])
}

// good: 使用data前应判断长度是否合法
func decode(data []byte) bool {
  if len(data) == 6 {
    if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z' && data[4]
== 'E' && data[5] == 'R' {
      fmt.Println("Good")
      return true
    }
  }
  return false
}
```

1.2【必须】nil指针判断

- 进行指针操作时，必须判断该指针是否为nil，防止程序panic，尤其在进行结构体Unmarshal时

```
type Packet struct {
  PackeyType    uint8
  PackeyVersion uint8
  Data          *Data
}

type Data struct {
  Stat uint8
  Len  uint8
```

```
  Buf  [8]byte
}

func (p *Packet) UnmarshalBinary(b []byte) error {
  if len(b) < 2 {
    return io.EOF
  }

  p.PackeyType = b[0]
  p.PackeyVersion = b[1]

  // 若长度等于2，那么不会new Data
  if len(b) > 2 {
    p.Data = new(Data)
  }
  return nil
}

// bad: 未判断指针是否为nil
func main() {
  packet := new(Packet)
  data := make([]byte, 2)
  if err := packet.UnmarshalBinary(data); err != nil {
    fmt.Println("Failed to unmarshal packet")
    return
  }

  fmt.Printf("Stat: %v\n", packet.Data.Stat)
}

// good: 判断Data指针是否为nil
func main() {
  packet := new(Packet)
  data := make([]byte, 2)

  if err := packet.UnmarshalBinary(data); err != nil {
    fmt.Println("Failed to unmarshal packet")
    return
  }

  if packet.Data == nil {
    return
  }

  fmt.Printf("Stat: %v\n", packet.Data.Stat)
}
```

1.3【必须】整数安全

- 在进行数字运算操作时，需要做好长度限制，防止外部输入运算导致异常：
  - 确保无符号整数运算时不会反转
  - 确保有符号整数运算时不会出现溢出
  - 确保整型转换时不会出现截断错误
  - 确保整型转换时不会出现符号错误
- 以下场景必须严格进行长度限制：
  - 作为数组索引
  - 作为对象的长度或者大小
  - 作为数组的边界（如作为循环计数器）

```
// bad：未限制长度，导致整数溢出
func overflow(numControlByUser int32) {
  var numInt int32 = 0
  numInt = numControlByUser + 1
  // 对长度限制不当，导致整数溢出
  fmt.Printf("%d\n", numInt)
  // 使用numInt，可能导致其他错误
}

func main() {
  overflow(2147483647)
}

// good
func overflow(numControlByUser int32) {
  var numInt int32 = 0
  numInt = numControlByUser + 1
  if numInt < 0 {
    fmt.Println("integer overflow")
    return
  }
  fmt.Println("integer ok")
}

func main() {
  overflow(2147483647)
}
```

1.4【必须】make分配长度验证

- 在进行make分配内存时，需要对外部可控的长度进行校验，防止程序panic。

```
// bad
func parse(lenControlByUser int, data []byte) {
  size := lenControlByUser
  // 对外部传入的size，进行长度判断以免导致panic
```

```
  buffer := make([]byte, size)
  copy(buffer, data)
}

// good
func parse(lenControlByUser int, data []byte) ([]byte, error) {
  size := lenControlByUser
  // 限制外部可控的长度大小范围
  if size > 64*1024*1024 {
    return nil, errors.New("value too large")
  }
  buffer := make([]byte, size)
  copy(buffer, data)
  return buffer, nil
}
```

1.5【必须】禁止SetFinalizer和指针循环引用同时使用

- 当一个对象从被GC选中到移除内存之前，runtime.SetFinalizer()都不会执行，即使程序正常结束或者发生错误。由指针构成的"循环引用"虽然能被GC正确处理，但由于无法确定Finalizer依赖顺序，从而无法调用runtime.SetFinalizer()，导致目标对象无法变成可达状态，从而造成内存无法被回收。

```
// bad
func foo() {
  var a, b Data
  a.o = &b
  b.o = &a

  // 指针循环引用，SetFinalizer()无法正常调用
  runtime.SetFinalizer(&a, func(d *Data) {
    fmt.Printf("a %p final.\n", d)
  })
  runtime.SetFinalizer(&b, func(d *Data) {
    fmt.Printf("b %p final.\n", d)
  })
}

func main() {
  for {
    foo()
    time.Sleep(time.Millisecond)
  }
}
```

1.6【必须】禁止重复释放channel

- 重复释放一般存在于异常流程判断中，如果恶意攻击者构造出异常条件使程序重复释放channel，则会触发运行时panic，从而造成DoS攻击。

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

// good
func foo(c chan int) {
  defer close(c) // 使用defer延迟关闭channel
  err := processBusiness()
  if err != nil {
    c <- 0
    return
  }
  c <- 1
}
```

1.7【必须】确保每个协程都能退出

- 启动一个协程就会做一个入栈操作，在系统不退出的情况下，协程也没有设置退出条件，则相当于协程失去了控制，它占用的资源无法回收，可能会导致内存泄露。

```
// bad: 协程没有设置退出条件
func doWaiter(name string, second int) {
  for {
    time.Sleep(time.Duration(second) * time.Second)
    fmt.Println(name, " is ready!")
  }
}
```

1.8【推荐】不使用unsafe包

- 由于unsafe包绕过了 Golang 的内存安全原则，一般来说使用该库是不安全的，可导致内存破坏，尽量避免使用该包。若必须要使用unsafe操作指针，必须做好安全校验。

```
// bad: 通过unsafe操作原始指针
func unsafePointer() {
  b := make([]byte, 1)
  foo := (*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(0xfffffffe)))
  fmt.Print(*foo + 1)
}

// [signal SIGSEGV: segmentation violation code=0x1 addr=0xc100068f55 pc=0x49142b]
```

1.9【推荐】不使用slice作为函数入参

- slice是引用类型，在作为函数入参时采用的是地址传递，对slice的修改也会影响原始数据

```
// bad: slice作为函数入参时是地址传递
func modify(array []int) {
  array[0] = 10 // 对入参slice的元素修改会影响原始数据
}

func main() {
  array := []int{1, 2, 3, 4, 5}

  modify(array)
  fmt.Println(array) // output: [10 2 3 4 5]
}

// good: 函数使用数组作为入参，而不是slice
func modify(array [5]int) {
  array[0] = 10
}

func main() {
  // 传入数组，注意数组与slice的区别
  array := [5]int{1, 2, 3, 4, 5}

  modify(array)
  fmt.Println(array)
}
```

2 文件操作

2.1【必须】 路径穿越检查

- 在进行文件操作时，如果对外部传入的文件名未做限制，可能导致任意文件读取或者任意文件写入，严重可能导致代码执行。

```
// bad: 任意文件读取
func handler(w http.ResponseWriter, r *http.Request) {
```

```
  path := r.URL.Query()["path"][0]

  // 未过滤文件路径，可能导致任意文件读取
  data, _ := ioutil.ReadFile(path)
  w.Write(data)

  // 对外部传入的文件名变量，还需要验证是否存在../等路径穿越的文件名
  data, _ = ioutil.ReadFile(filepath.Join("/home/user/", path))
  w.Write(data)
}

// bad: 任意文件写入
func unzip(f string) {
  r, _ := zip.OpenReader(f)
  for _, f := range r.File {
    p, _ := filepath.Abs(f.Name)
    // 未验证压缩文件名，可能导致../等路径穿越，任意文件路径写入
    ioutil.WriteFile(p, []byte("present"), 0640)
  }
}

// good: 检查压缩的文件名是否包含..路径穿越特征字符，防止任意写入
func unzipGood(f string) bool {
  r, err := zip.OpenReader(f)
  if err != nil {
    fmt.Println("read zip file fail")
    return false
  }
  for _, f := range r.File {
    if !strings.Contains(f.Name, "..") {
      p, _ := filepath.Abs(f.Name)
      ioutil.WriteFile(p, []byte("present"), 0640)
    } else {
      return false
    }
  }
  return true
}
```

2.2【必须】 文件访问权限

- 根据创建文件的敏感性设置不同级别的访问权限，以防止敏感数据被任意权限用户读取。例如，设置文件权限为：`-rw-r-----`

```
ioutil.WriteFile(p, []byte("present"), 0640)
```

3 系统接口

3.1【必须】命令执行检查

- 使用`exec.Command`、`exec.CommandContext`、`syscall.StartProcess`、`os.StartProcess`等函数时，第一个参数（path）直接取外部输入值时，应使用白名单限定可执行的命令范围，不允许传入`bash`、`cmd`、`sh`等命令；
- 使用`exec.Command`、`exec.CommandContext`等函数时，通过`bash`、`cmd`、`sh`等创建shell，-c后的参数（arg）拼接外部输入，应过滤\n $ & ; | ' " ( ) ` 等潜在恶意字符；

```
// bad
func foo() {
  userInputedVal := "&& echo 'hello'" // 假设外部传入该变量值
  cmdName := "ping " + userInputedVal

  // 未判断外部输入是否存在命令注入字符，结合sh可造成命令注入
  cmd := exec.Command("sh", "-c", cmdName)
  output, _ := cmd.CombinedOutput()
  fmt.Println(string(output))

  cmdName := "ls"
  // 未判断外部输入是否是预期命令
  cmd := exec.Command(cmdName)
  output, _ := cmd.CombinedOutput()
  fmt.Println(string(output))
}

// good
func checkIllegal(cmdName string) bool {
  if strings.Contains(cmdName, "&") || strings.Contains(cmdName, "|") ||
strings.Contains(cmdName, ";") ||
    strings.Contains(cmdName, "$") || strings.Contains(cmdName, "'") ||
strings.Contains(cmdName, "`") ||
    strings.Contains(cmdName, "(") || strings.Contains(cmdName, ")") ||
strings.Contains(cmdName, "\"") {
    return true
  }
  return false
}

func main() {
  userInputedVal := "&& echo 'hello'"
  cmdName := "ping " + userInputedVal

  if checkIllegal(cmdName) { // 检查传给sh的命令是否有特殊字符
    return // 存在特殊字符直接return
  }

  cmd := exec.Command("sh", "-c", cmdName)
```

```
  output, _ := cmd.CombinedOutput()
  fmt.Println(string(output))
}
```

4 通信安全

4.1【必须】网络通信采用TLS方式

- 明文传输的通信协议目前已被验证存在较大安全风险，被中间人劫持后可能导致许多安全风险，因此必须采用至少TLS的安全通信方式保证通信安全，例如gRPC/Websocket都使用TLS1.3。

```
// good
func main() {
  http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) {
    w.Header().Add("Strict-Transport-Security", "max-age=63072000; includeSubDomains")
    w.Write([]byte("This is an example server.\n"))
  })

  // 服务器配置证书与私钥
  log.Fatal(http.ListenAndServeTLS(":443", "yourCert.pem", "yourKey.pem", nil))
}
```

4.2【推荐】TLS启用证书验证

- TLS证书应当是有效的、未过期的，且配置正确的域名，生产环境的服务端应启用证书验证。

```
// bad
import (
  "crypto/tls"
  "net/http"
)

func doAuthReq(authReq *http.Request) *http.Response {
  tr := &http.Transport{
    TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
  }
  client := &http.Client{Transport: tr}
  res, _ := client.Do(authReq)
  return res
}

// good
import (
  "crypto/tls"
  "net/http"
)
```

```
func doAuthReq(authReq *http.Request) *http.Response {
  tr := &http.Transport{
    TLSClientConfig: &tls.Config{InsecureSkipVerify: false},
  }
  client := &http.Client{Transport: tr}
  res, _ := client.Do(authReq)
  return res
}
```

5 敏感数据保护

5.1【必须】敏感信息访问

- 禁止将敏感信息硬编码在程序中，既可能会将敏感信息暴露给攻击者，也会增加代码管理和维护的难度
- 使用配置中心系统统一托管密钥等敏感信息

5.2【必须】敏感数据输出

- 只输出必要的最小数据集，避免多余字段暴露引起敏感信息泄露
- 不能在日志保存密码（包括明文密码和密文密码）、密钥和其它敏感信息
- 对于必须输出的敏感信息，必须进行合理脱敏展示

```
// bad
func serve() {
  http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    user := r.Form.Get("user")
    pw := r.Form.Get("password")

    log.Printf("Registering new user %s with password %s.\n", user, pw)
  })
  http.ListenAndServe(":80", nil)
}

// good
func serve1() {
  http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    user := r.Form.Get("user")
    pw := r.Form.Get("password")

    log.Printf("Registering new user %s.\n", user)

    // ...
    use(pw)
  })
  http.ListenAndServe(":80", nil)
}
```

- 避免通过GET方法、代码注释、自动填充、缓存等方式泄露敏感信息

5.3【必须】敏感数据存储

- 敏感数据应使用SHA2、RSA等算法进行加密存储
- 敏感数据应使用独立的存储层，并在访问层开启访问控制
- 包含敏感信息的临时文件或缓存一旦不再需要应立刻删除

5.4【必须】异常处理和日志记录

- 应合理使用panic、recover、defer处理系统异常，避免出错信息输出到前端

```
defer func () {
  if r := recover(); r != nil {
    fmt.Println("Recovered in start()")
  }
}()
```

- 对外环境禁止开启debug模式，或将程序运行日志输出到前端

```
// bad
dlv --listen=:2345 --headless=true --api-version=2 debug test.go
// good
dlv debug test.go
```

6 加密解密

6.1【必须】不得硬编码密码/密钥

- 在进行用户登陆，加解密算法等操作时，不得在代码里硬编码密钥或密码，可通过变换算法或者配置等方式设置密码或者密钥。

```
// bad
const (
  user     = "dbuser"
  password = "s3cretp4ssword"
)

func connect() *sql.DB {
  connStr := fmt.Sprintf("postgres://%s:%s@localhost/pqgotest", user, password)
  db, err := sql.Open("postgres", connStr)
  if err != nil {
    return nil
  }
  return db
}

// bad
```

```
var (
  commonkey = []byte("0123456789abcdef")
)

func AesEncrypt(plaintext string) (string, error) {
  block, err := aes.NewCipher(commonkey)
  if err != nil {
    return "", err
  }
}
```

6.2【必须】密钥存储安全

- 在使用对称密码算法时，需要保护好加密密钥。当算法涉及敏感、业务数据时，可通过非对称算法协商加密密钥。其他较为不敏感的数据加密，可以通过变换算法等方式保护密钥。

6.3【推荐】不使用弱密码算法

- 在使用加密算法时，不建议使用加密强度较弱的算法。

```
// bad
crypto/des，crypto/md5，crypto/sha1，crypto/rc4等。

// good
crypto/rsa，crypto/aes等。
```

7 正则表达式

7.1【推荐】使用regexp进行正则表达式匹配

- 正则表达式编写不恰当可被用于DoS攻击，造成服务不可用，推荐使用regexp包进行正则表达式匹配。regexp保证了线性时间性能和优雅的失败：对解析器、编译器和执行引擎都进行了内存限制。但regexp不支持以下正则表达式特性，如业务依赖这些特性，则regexp不适合使用。
  - 回溯引用Backreferences
  - 查看Lookaround

```
// good
matched, err := regexp.MatchString(`a.b`, "aaxbb")
fmt.Println(matched) // true
fmt.Println(err)     // nil
```

8 校验

8.1【必须】按类型进行外部输入数据校验

- 所有外部输入的参数，应使用`validator`进行白名单校验，校验内容包括但不限于数据长度、数据范围、数据类型与格式，校验不通过的应当拒绝

```
// good
import (
  "fmt"
  "github.com/go-playground/validator/v10"
)

var validate *validator.Validate

func validateVariable() {
  myEmail := "abc@tencent.com"
  errs := validate.Var(myEmail, "required,email")
  if errs != nil {
    fmt.Println(errs)
    return
    //停止执行
  }
  // 验证通过，继续执行
  ...
}

func main() {
  validate = validator.New()
  validateVariable()
}
```

- 无法通过白名单校验的应使用`html.EscapeString`、`text/template`或`bluemonday`对`<, >, &, ',"`等字符进行过滤或编码

```
import (
  "text/template"
)

// TestHTMLEscapeString HTML特殊字符转义
func main(inputValue string) string {
  escapedResult := template.HTMLEscapeString(inputValue)
  return escapedResult
}
```

8.2【必须】模板渲染过滤验证

- 使用`text/template`或者`html/template`渲染模板时禁止将外部输入参数引入模板，或仅允许引入白名单内字符。

```
// bad
func handler(w http.ResponseWriter, r *http.Request) {
  r.ParseForm()
  x := r.Form.Get("name")

  var tmpl = `<!DOCTYPE html><html><body>
    <form action="/" method="post">
        First name:<br>
    <input type="text" name="name" value="">
    <input type="submit" value="Submit">
    </form><p>` + x + ` </p></body></html>`

  t := template.New("main")
  t, _ = t.Parse(tmpl)
  t.Execute(w, "Hello")
}

// good
import (
  "fmt"
  "github.com/go-playground/validator/v10"
)

var validate *validator.Validate
validate = validator.New()

func validateVariable(val) {
  errs := validate.Var(val, "gte=1,lte=100") // 限制必须是1-100的正整数
  if errs != nil {
    fmt.Println(errs)
    return false
  }
  return true
}

func handler(w http.ResponseWriter, r *http.Request) {
  r.ParseForm()
  x := r.Form.Get("name")

  if validateVariable(x) {
    var tmpl = `<!DOCTYPE html><html><body>
            <form action="/" method="post">
            First name:<br>
            <input type="text" name="name" value="">
```

```
            <input type="submit" value="Submit">
            </form><p>` + x + ` </p></body></html>`
    t := template.New("main")
    t, _ = t.Parse(tmpl)
    t.Execute(w, "Hello")
  } else {
    // ...
  }
}
```

9 SQL操作

9.1【必须】SQL语句默认使用预编译并绑定变量

- 使用`database/sql`的prepare、Query或使用GORM等ORM执行SQL操作

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/sqlite"
)

type Product struct {
  gorm.Model
  Code  string
  Price uint
}

...
var product Product
...
db.First(&product, 1)
```

- 使用参数化查询，禁止拼接SQL语句，另外对于传入参数用于order by或表名的需要通过校验

```
// bad
import (
  "database/sql"
  "fmt"
  "net/http"
)

func handler(db *sql.DB, req *http.Request) {
  q := fmt.Sprintf("SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='%s' ORDER BY
PRICE",
    req.URL.Query()["category"])
  db.Query(q)
}
```

```
// good
func handlerGood(db *sql.DB, req *http.Request) {
  // 使用?占位符
  q := "SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='?' ORDER BY PRICE"
  db.Query(q, req.URL.Query()["category"])
}
```

10 会话管理

10.1【必须】安全维护session信息

- 用户登录时应重新生成session，退出登录后应清理session。

```
import (
  "github.com/gorilla/handlers"
  "github.com/gorilla/mux"
  "net/http"
)

// 创建cookie
func setToken(res http.ResponseWriter, req *http.Request) {
  expireToken := time.Now().Add(time.Minute * 30).Unix()
  expireCookie := time.Now().Add(time.Minute * 30)

  //...

  cookie := http.Cookie{
    Name:     "Auth",
    Value:    signedToken,
    Expires:  expireCookie, // 过期失效
    HttpOnly: true,
    Path:     "/",
    Domain:   "127.0.0.1",
    Secure:   true,
  }

  http.SetCookie(res, &cookie)
  http.Redirect(res, req, "/profile", 307)
}

// 删除cookie
func logout(res http.ResponseWriter, req *http.Request) {
  deleteCookie := http.Cookie{
    Name:    "Auth",
    Value:   "none",
    Expires: time.Now(),
  }
```

```
  http.SetCookie(res, &deleteCookie)
  return
}
```

10.2【必须】CSRF防护

- 涉及系统敏感操作或可读取敏感信息的接口应校验`Referer`或添加`csrf_token`。

```
// good
import (
  "github.com/gorilla/csrf"
  "github.com/gorilla/mux"
  "net/http"
)

func main() {
  r := mux.NewRouter()
  r.HandleFunc("/signup", ShowSignupForm)
  r.HandleFunc("/signup/post", SubmitSignupForm)
  // 使用csrf_token验证
  http.ListenAndServe(":8000",
    csrf.Protect([]byte("32-byte-long-auth-key"))(r))
}
```

四、实践中遇到的坑

1. 并发安全

1.1【必须】禁止在闭包中直接调用循环变量

- 在循环中启动协程，当协程中使用到了循环的索引值，由于多个协程同时使用同一个变量会产生数据竞争，造成执行结果异常。

```
// bad
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())
  var group sync.WaitGroup

  for i := 0; i < 5; i++ {
    group.Add(1)
    go func() {
      defer group.Done()
      fmt.Printf("%-2d", i) // 这里打印的i不是所期望的
    }()
  }
  group.Wait()
}
```

```
// good
func main() {
  runtime.GOMAXPROCS(runtime.NumCPU())
  var group sync.WaitGroup

  for i := 0; i < 5; i++ {
    group.Add(1)
    go func(j int) {
      defer func() {
        if r := recover(); r != nil {
          fmt.Println("Recovered in start()")
        }
        group.Done()
      }()
      fmt.Printf("%-2d", j) // 闭包内部使用局部变量
    }(i) // 把循环变量显式地传给协程
  }
  group.Wait()
}
```

1.2【必须】禁止并发写map

- 并发写map容易造成程序崩溃并异常退出，建议加锁保护

```
// bad
func main() {
  m := make(map[int]int)
  // 并发读写
  go func() {
    for {
      _ = m[1]
    }
  }()
  go func() {
    for {
      m[2] = 1
    }
  }()
  select {}
}
```

1.3【必须】确保并发安全

敏感操作如果未作并发安全限制，可导致数据读写异常，造成业务逻辑限制被绕过。可通过同步锁或者原子操作进行防护。

通过同步锁共享内存

```go
// good
var count int

func Count(lock *sync.Mutex) {
  lock.Lock() // 加写锁
  count++
  fmt.Println(count)
  lock.Unlock() // 解写锁，任何一个Lock()或RLock()均需要保证对应有Unlock()或RUnlock()
}

func main() {
  lock := &sync.Mutex{}
  for i := 0; i < 10; i++ {
    go Count(lock) // 传递指针是为了防止函数内的锁和调用锁不一致
  }
  for {
    lock.Lock()
    c := count
    lock.Unlock()
    runtime.Gosched() // 交出时间片给协程
    if c > 10 {
      break
    }
  }
}
```

- 使用`sync/atomic`执行原子操作

```go
// good
import (
  "sync"
  "sync/atomic"
)

func main() {
  type Map map[string]string
  var m atomic.Value
  m.Store(make(Map))
  var mu sync.Mutex // used only by writers
  read := func(key string) (val string) {
    m1 := m.Load().(Map)
    return m1[key]
  }
  insert := func(key, val string) {
    mu.Lock() // 与潜在写入同步
    defer mu.Unlock()
    m1 := m.Load().(Map) // 导入struct当前数据
    m2 := make(Map)      // 创建新值
```

```
    for k, v := range m1 {
      m2[k] = v
    }
    m2[key] = val
    m.Store(m2) // 用新的替代当前对象
  }
  _, _ = read, insert
}
```

五、参考

- Effective Go
- The Go common mistakes guide
- uber_go_guide_cn
- 腾讯代码安全指南
- sync-pool使用