特别放送-给你一份清晰、可直接套用的Go编码规范

你好，我是孔令飞。

我们在上一讲学习了“写出优雅Go项目的方法论”，那一讲内容很丰富，是我多年Go项目开发的经验沉淀，需要你多花一些时间好好消化吸收。吃完大餐之后，咱们今天来一期特别放送，就是上一讲我提到过的编码规范。这一讲里，为了帮你节省时间和精力，我会给你一份清晰、可直接套用的 Go 编码规范，帮助你编写一个高质量的 Go 应用。

这份规范，是我参考了Go官方提供的编码规范，以及Go社区沉淀的一些比较合理的规范之后，加入自己的理解总结出的，它比很多公司内部的规范更全面，你掌握了，以后在面试大厂的时候，或者在大厂里写代码的时候，都会让人高看你一眼，觉得你code很专业。

这份编码规范中包含代码风格、命名规范、注释规范、类型、控制结构、函数、GOPATH 设置规范、依赖管理和最佳实践九类规范。如果你觉得这些规范内容太多了，看完一遍也记不住，这完全没关系。你可以多看几遍，也可以在用到时把它翻出来，在实际应用中掌握。这篇特别放送的内容，更多是作为写代码时候的一个参考手册。

1. 代码风格

1.1 代码格式

- 代码都必须用 `gofmt` 进行格式化。
- 运算符和操作数之间要留空格。
- 建议一行代码不超过120个字符，超过部分，请采用合适的换行方式换行。但也有些例外场景，例如import行、工具自动生成的代码、带tag的struct字段。
- 文件长度不能超过800行。
- 函数长度不能超过80行。
- import规范
  - 代码都必须用 goimports进行格式化（建议将代码Go代码编辑器设置为：保存时运行 goimports）。
  - 不要使用相对路径引入包，例如 import ../util/net 。
  - 包名称与导入路径的最后一个目录名不匹配时，或者多个相同包名冲突时，则必须使用导入别名。

```
// bad
 "github.com/dgrijalva/jwt-go/v4"

 //good
 jwt "github.com/dgrijalva/jwt-go/v4"
```

-
  - 导入的包建议进行分组，匿名包的引用使用一个新的分组，并对匿名包引用进行说明。

```
import (
 // go 标准包
 "fmt"
```

```
  // 第三方包
    "github.com/jinzhu/gorm"
    "github.com/spf13/cobra"
    "github.com/spf13/viper"

  // 匿名包单独分组，并对匿名包引用进行说明
    // import mysql driver
    _ "github.com/jinzhu/gorm/dialects/mysql"

  // 内部包
    v1 "github.com/marmotedu/api/apiserver/v1"
    metav1 "github.com/marmotedu/apimachinery/pkg/meta/v1"
    "github.com/marmotedu/iam/pkg/cli/genericclioptions"
 )
```

1.2 声明、初始化和定义

- 当函数中需要使用到多个变量时，可以在函数开始处使用var声明。在函数外部声明必须使用 `var` ，不要采用 `:=` ，容易踩到变量的作用域的问题。

```
var (
 Width  int
 Height int
)
```

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

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

// good
sval := T{Name: "foo"}

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

- struct 声明和初始化格式采用多行，定义如下。

```
type User struct{
    Username  string
    Email     string
}

user := User{
 Username: "colin",
```

```
 Email: "colin404@foxmail.com",
}
```

- 相似的声明放在一组，同样适用于常量、变量和类型声明。

```
// bad
import "a"
import "b"

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

- 尽可能指定容器容量，以便为容器预先分配内存，例如：

```
v := make(map[int]string, 4)
v := make([]string, 0, 4)
```

- 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()
// 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" }
```

- 对于未导出的顶层常量和变量，使用_作为前缀。

```
// bad
const (
  defaultHost = "127.0.0.1"
  defaultPort = 8080
)

// good
const (
  _defaultHost = "127.0.0.1"
```

```
  _defaultPort = 8080
)
```

- 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
type Client struct {
  version int
  http.Client
}

// good
type Client struct {
  http.Client

  version int
}
```

1.3 错误处理

- error作为函数的值返回，必须对error进行处理，或将返回值赋值给明确忽略。对于defer xx.Close()可以不用显式处理。

```
func load() error {
 // normal code
}

// bad
load()

// good
 _ := load()
```

- error作为函数的值返回且有多个返回值的时候，error必须是最后一个参数。

```
// bad
func load() (error, int) {
 // normal code
}

// good
func load() (int, error) {
 // normal code
}
```

- 尽早进行错误处理，并尽早返回，减少嵌套。

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

// good
if err != nil {
 // error handling
 return err
}
// normal code
```

- 如果需要在 if 之外使用函数调用的结果，则应采用下面的方式。

```
// bad
if v, err := foo(); err != nil {
 // error handling
}

// good
v, err := foo()
if err != nil {
 // error handling
}
```

- 错误要单独判断，不与其他逻辑组合判断。

```
// bad
v, err := foo()
if err != nil || v  == nil {
 // error handling
 return err
}

// good
v, err := foo()
if err != nil {
 // error handling
 return err
}

if v == nil {
 // error handling
 return errors.New("invalid value v")
}
```

- 如果返回值需要初始化，则采用下面的方式。

```
v, err := f()
if err != nil {
    // error handling
    return // or continue.
}
// use v
```

- 错误描述建议
  - 告诉用户他们可以做什么，而不是告诉他们不能做什么。
  - 当声明一个需求时，用must 而不是should。例如，must be greater than 0、must match regex '[a-z]+'。
  - 当声明一个格式不对时，用must not。例如，must not contain。
  - 当声明一个动作时用may not。例如，may not be specified when otherField is empty、only name may be specified。
  - 引用文字字符串值时，请在单引号中指示文字。例如，ust not contain '..'。
  - 当引用另一个字段名称时，请在反引号中指定该名称。例如，must be greater than `request`。
  - 指定不等时，请使用单词而不是符号。例如，must be less than 256、must be greater than or equal to 0 (不要用 larger than、bigger than、more than、higher than)。
  - 指定数字范围时，请尽可能使用包含范围。
  - 建议 Go 1.13 以上，error 生成方式为 `fmt.Errorf("module xxx: %w", err)`。
  - 错误描述用小写字母开头，结尾不要加标点符号，例如：

```
// bad
errors.New("Redis connection failed")
errors.New("redis connection failed.")

// good
errors.New("redis connection failed")
```

1.4 panic处理

- 在业务逻辑处理中禁止使用panic。
- 在main包中，只有当程序完全不可运行时使用panic，例如无法打开文件、无法连接数据库导致程序无法正常运行。
- 在main包中，使用 `log.Fatal` 来记录错误，这样就可以由log来结束程序，或者将panic抛出的异常记录到日志文件中，方便排查问题。
- 可导出的接口一定不能有panic。
- 包内建议采用error而不是panic来传递错误。

1.5 单元测试

- 单元测试文件名命名规范为 example_test.go。
- 每个重要的可导出函数都要编写测试用例。
- 因为单元测试文件内的函数都是不对外的，所以可导出的结构体、函数等可以不带注释。
- 如果存在 func （b *Bar） Foo，单测函数可以为 func TestBar_Foo。

1.6 类型断言失败处理

type assertion 的单个返回值针对不正确的类型将产生 panic。请始终使用 “comma ok”的惯用法。

```
// bad
t := n.(int)

// good
t, ok := n.(int)
if !ok {
 // error handling
}
// normal code
```

2. 命名规范

命名规范是代码规范中非常重要的一部分，一个统一的、短小的、精确的命名规范可以大大提高代码的可读性，也可以借此规避一些不必要的Bug。

2.1 包命名

- 包名必须和目录名一致，尽量采取有意义、简短的包名，不要和标准库冲突。
- 包名全部小写，没有大写或下划线，使用多级目录来划分层级。
- 项目名可以通过中划线来连接多个单词。
- 包名以及包所在的目录名，不要使用复数，例如，是net/utl，而不是net/urls。
- 不要用 common、util、shared 或者 lib 这类宽泛的、无意义的包名。
- 包名要简单明了，例如 net、time、log。

2.2 函数命名

- 函数名采用驼峰式，首字母根据访问控制决定使用大写或小写，例如：MixedCaps或者mixedCaps。
- 代码生成工具自动生成的代码(如xxxx.pb.go)和为了对相关测试用例进行分组，而采用的下划线(如TestMyFunction_WhatIsBeingTested)排除此规则。

2.3 文件命名

- 文件名要简短有意义。
- 文件名应小写，并使用下划线分割单词。

2.4 结构体命名

- 采用驼峰命名方式，首字母根据访问控制决定使用大写或小写，例如MixedCaps或者mixedCaps。
- 结构体名不应该是动词，应该是名词，比如 Node、NodeSpec。
- 避免使用Data、Info这类无意义的结构体名。
- 结构体的声明和初始化应采用多行，例如：

```
// User 多行声明
type User struct {
    Name  string
    Email string
}

// 多行初始化
u := User{
    UserName: "colin",
    Email:    "colin404@foxmail.com",
}
```

2.5 接口命名

- 接口命名的规则，基本和结构体命名规则保持一致：
  - 单个函数的接口名以 “er"” 作为后缀（例如Reader，Writer），有时候可能导致蹩脚的英文，但是没关系。
  - 两个函数的接口名以两个函数名命名，例如ReadWriter。
  - 三个以上函数的接口名，类似于结构体名。

例如：

```
// Seeking to an offset before the start of the file is an error.
// Seeking to any positive offset is legal, but the behavior of subsequent
// I/O operations on the underlying object is implementation-dependent.
type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}

// ReadWriter is the interface that groups the basic Read and Write methods.
type ReadWriter interface {
    Reader
    Writer
}
```

2.6 变量命名

- 变量名必须遵循**驼峰式**，首字母根据访问控制决定使用大写或小写。
- 在相对简单（对象数量少、针对性强）的环境中，可以将一些名称由完整单词简写为单个字母，例如：
  - user 可以简写为 u；

  - userID 可以简写 uid。

- 特有名词时，需要遵循以下规则：
  - 如果变量为私有，且特有名词为首个单词，则使用小写，如 apiClient。
  - 其他情况都应当使用该名词原有的写法，如 APIClient、repoID、UserID。

下面列举了一些常见的特有名词。

```
// A GonicMapper that contains a list of common initialisms taken from golang/lint
var LintGonicMapper = GonicMapper{
    "API":   true,
    "ASCII": true,
    "CPU":   true,
    "CSS":   true,
    "DNS":   true,
    "EOF":   true,
    "GUID":  true,
    "HTML":  true,
    "HTTP":  true,
    "HTTPS": true,
    "ID":    true,
    "IP":    true,
    "JSON":  true,
    "LHS":   true,
    "QPS":   true,
    "RAM":   true,
    "RHS":   true,
    "RPC":   true,
    "SLA":   true,
    "SMTP":  true,
    "SSH":   true,
    "TLS":   true,
    "TTL":   true,
    "UI":    true,
    "UID":   true,
    "UUID":  true,
    "URI":   true,
    "URL":   true,
    "UTF8":  true,
    "VM":    true,
    "XML":   true,
    "XSRF":  true,
    "XSS":   true,
}
```

- 若变量类型为bool类型，则名称应以Has，Is，Can或Allow开头，例如：

```
var hasConflict bool
var isExist bool
var canManage bool
var allowGitHook bool
```

- 局部变量应当尽可能短小，比如使用buf指代buffer，使用i指代index。
- 代码生成工具自动生成的代码可排除此规则(如xxx.pb.go里面的Id)

2.7 常量命名

- 常量名必须遵循驼峰式，首字母根据访问控制决定使用大写或小写。
- 如果是枚举类型的常量，需要先创建相应类型：

```
// Code defines an error code type.
type Code int

// Internal errors.
const (
    // ErrUnknown - 0: An unknown error occurred.
    ErrUnknown Code = iota
    // ErrFatal - 1: An fatal error occurred.
    ErrFatal
)
```

2.8 Error的命名

- Error类型应该写成FooError的形式。

```
type ExitError struct {
 // ....
}
```

- Error变量写成ErrFoo的形式。

```
var ErrFormat = errors.New("unknown format")
```

3. 注释规范

- 每个可导出的名字都要有注释，该注释对导出的变量、函数、结构体、接口等进行简要介绍。
- 全部使用单行注释，禁止使用多行注释。
- 和代码的规范一样，单行注释不要过长，禁止超过 120 字符，超过的请使用换行展示，尽量保持格式优雅。
- 注释必须是完整的句子，以需要注释的内容作为开头，句点作为结尾，格式为 `// 名称 描述.`。例如：

```
// bad
// logs the flags in the flagset.
```

```
func PrintFlags(flags *pflag.FlagSet) {
 // normal code
}

// good
// PrintFlags logs the flags in the flagset.
func PrintFlags(flags *pflag.FlagSet) {
 // normal code
}
```

- 所有注释掉的代码在提交code review前都应该被删除，否则应该说明为什么不删除，并给出后续处理建议。
- 在多段注释之间可以使用空行分隔加以区分，如下所示：

```
// Package superman implements methods for saving the world.
//
// Experience has shown that a small number of procedures can prove
// helpful when attempting to save the world.
package superman
```

3.1 包注释

- 每个包都有且仅有一个包级别的注释。
- 包注释统一用 `//` 进行注释，格式为 `// Package 包名 包描述` ，例如：

```
// Package genericclioptions contains flags which can be added to you command, bound, completed, and produc
// useful helper functions.
package genericclioptions
```

3.2 变量/常量注释

- 每个可导出的变量/常量都必须有注释说明，格式为`// 变量名 变量描述`，例如：

```
// ErrSigningMethod defines invalid signing method error.
var ErrSigningMethod = errors.New("Invalid signing method")
```

- 出现大块常量或变量定义时，可在前面注释一个总的说明，然后在每一行常量的前一行或末尾详细注释该常量的定义，例如：

```
// Code must start with 1xxxxx.
const (
    // ErrSuccess - 200: OK.
```

```
    ErrSuccess int = iota + 100001

    // ErrUnknown - 500: Internal server error.
    ErrUnknown

    // ErrBind - 400: Error occurred while binding the request body to the struct.
    ErrBind

    // ErrValidation - 400: Validation failed.
    ErrValidation
)
```

3.3 结构体注释

- 每个需要导出的结构体或者接口都必须有注释说明，格式为 `// 结构体名 结构体描述.`。
- 结构体内的可导出成员变量名，如果意义不明确，必须要给出注释，放在成员变量的前一行或同一行的末尾。例如：

```
// User represents a user restful resource. It is also used as gorm model.
type User struct {
    // Standard object's metadata.
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Nickname string `json:"nickname" gorm:"column:nickname"`
    Password string `json:"password" gorm:"column:password"`
    Email    string `json:"email" gorm:"column:email"`
    Phone    string `json:"phone" gorm:"column:phone"`
    IsAdmin  int    `json:"isAdmin,omitempty" gorm:"column:isAdmin"`
}
```

3.4 方法注释

- 每个需要导出的函数或者方法都必须有注释，格式为`// 函数名 函数描述.`，例如：

```
// BeforeUpdate run before update database record.
func (p *Policy) BeforeUpdate() (err error) {
 // normal code
 return nil
}
```

3.5 类型注释

- 每个需要导出的类型定义和类型别名都必须有注释说明，格式为 `// 类型名 类型描述.`，例如：

```
// Code defines an error code type.
type Code int
```

4. 类型

4.1 字符串

- 空字符串判断。

```
// bad
if s == "" {
    // normal code
}

// good
if len(s) == 0 {
    // normal code
}
```

- []byte/string相等比较。

```
// bad
var s1 []byte
var s2 []byte
...
bytes.Equal(s1, s2) == 0
bytes.Equal(s1, s2) != 0

// good
var s1 []byte
var s2 []byte
...
bytes.Compare(s1, s2) == 0
bytes.Compare(s1, s2) != 0
```

- 字符串是否包含子串或字符。

```
// bad
strings.Contains(s, subStr)
strings.ContainsAny(s, char)
strings.ContainRune(s, r)

// good
strings.Index(s, subStr) > -1
strings.IndexAny(s, char) > -1
strings.IndexRune(s, r) > -1
```

- 去除前后子串。

```
// bad
```

```
var s1 = "a string value"
var s2 = "a "
var s3 = strings.TrimPrefix(s1, s2)

// good
var s1 = "a string value"
var s2 = "a "
var s3 string
if strings.HasPrefix(s1, s2) {
    s3 = s1[len(s2):]
}
```

- 复杂字符串使用raw字符串避免字符转义。

```
// bad
regexp.MustCompile("\\.")

// good
regexp.MustCompile(`\.`)
```

4.2 切片

- 空slice判断。

```
// bad
if len(slice) > 0 {
    // normal code
}

// good
if slice != nil && len(slice) > 0 {
    // normal code
}
```

上面判断同样适用于map、channel。

- 声明slice。

```
// bad
s := []string{}
s := make([]string, 0)

// good
var s []string
```

- slice复制。

```
// bad
var b1, b2 []byte
for i, v := range b1 {
   b2[i] = v
}
for i := range b1 {
   b2[i] = b1[i]
}

// good
copy(b2, b1)
```

- slice新增。

```
// bad
var a, b []int
for _, v := range a {
    b = append(b, v)
}

// good
var a, b []int
b = append(b, a...)
```

4.3 结构体

- struct初始化。

struct以多行格式初始化。

```
ype user struct {
 Id   int64
 Name string
}

u1 := user{100, "Colin"}

u2 := user{
    Id:   200,
    Name: "Lex",
}
```

5. 控制结构

5.1 if

- if 接受初始化语句，约定如下方式建立局部变量。

```
if err := loadConfig(); err != nil {
 // error handling
 return err
}
```

- if 对于bool类型的变量，应直接进行真假判断。

```
var isAllow bool
if isAllow {
 // normal code
}
```

5.2 for

- 采用短声明建立局部变量。

```
sum := 0
for i := 0; i < 10; i++ {
    sum += 1
}
```

- 不要在 for 循环里面使用 defer，defer只有在函数退出时才会执行。

```
// bad
for file := range files {
 fd, err := os.Open(file)
 if err != nil {
  return err
 }
 defer fd.Close()
 // normal code
}

// good
for file := range files {
 func() {
  fd, err := os.Open(file)
  if err != nil {
   return err
  }
  defer fd.Close()
  // normal code
 }()
}
```

5.3 range

- 如果只需要第一项（key），就丢弃第二个。

```
for key := range keys {
// normal code
}
```

- 如果只需要第二项，则把第一项置为下划线。

```
sum := 0
for _, value := range array {
    sum += value
}
```

5.4 switch

- 必须要有default。

```
switch os := runtime.GOOS; os {
    case "linux":
        fmt.Println("Linux.")
    case "darwin":
        fmt.Println("OS X.")
    default:
        fmt.Printf("%s.\n", os)
}
```

5.5 goto

- 业务代码禁止使用 `goto` 。
- 框架或其他底层源码尽量不用。

6. 函数

- 传入变量和返回变量以小写字母开头。
- 函数参数个数不能超过5个。
- 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。
- 尽量采用值传递，而非指针传递。
- 传入参数是 map、slice、chan、interface ，不要传递指针。

6.1 函数参数

- 如果函数返回相同类型的两个或三个参数，或者如果从上下文中不清楚结果的含义，使用命名返回，其他情况不建议使用命名返回，例如：

```
func coordinate() (x, y float64, err error) {
 // normal code
}
```

- 传入变量和返回变量都以小写字母开头。
- 尽量用值传递，非指针传递。
- 参数数量均不能超过5个。
- 多返回值最多返回三个，超过三个请使用 struct。

6.2 defer

- 当存在资源创建时，应紧跟defer释放资源(可以大胆使用defer，defer在Go1.14版本中，性能大幅提升，defer的性能损耗即使在性能敏感型的业务中，也可以忽略)。
- 先判断是否错误，再defer释放资源，例如：

```
rep, err := http.Get(url)
if err != nil {
    return err
}

defer resp.Body.Close()
```

6.3 方法的接收器

- 推荐以类名第一个英文首字母的小写作为接收器的命名。
- 接收器的命名在函数超过20行的时候不要用单字符。
- 接收器的命名不能采用me、this、self这类易混淆名称。

6.4 嵌套

- 嵌套深度不能超过4层。

6.5 变量命名

- 变量声明尽量放在变量第一次使用的前面，遵循就近原则。
- 如果魔法数字出现超过两次，则禁止使用，改用一个常量代替，例如：

```
// PI ...
const Prise = 3.14

func getAppleCost(n float64) float64 {
 return Prise * n
}

func getOrangeCost(n float64) float64 {
 return Prise * n
}
```

7. GOPATH 设置规范

- Go 1.11 之后，弱化了 GOPATH 规则，已有代码（很多库肯定是在1.11之前建立的）肯定符合这个规则，建议保留 GOPATH 规则，便于维护代码。
- 建议只使用一个 GOPATH，不建议使用多个 GOPATH。如果使用多个GOPATH，编译生效的 bin 目录是在第一个 GOPATH 下。

8. 依赖管理

- Go 1.11 以上必须使用 Go Modules。
- 使用Go Modules作为依赖管理的项目时，不建议提交vendor目录。
- 使用Go Modules作为依赖管理的项目时，必须提交go.sum文件。

9. 最佳实践

- 尽量少用全局变量，而是通过参数传递，使每个函数都是“无状态”的。这样可以减少耦合，也方便分工和单元测试。
- 在编译时验证接口的符合性，例如：

```
type LogHandler struct {
  h   http.Handler
  log *zap.Logger
}
var _ http.Handler = LogHandler{}
```

- 服务器处理请求时，应该创建一个context，保存该请求的相关信息（如requestID），并在函数调用链中传递。

9.1 性能

- string 表示的是不可变的字符串变量，对 string 的修改是比较重的操作，基本上都需要重新申请内存。所以，如果没有特殊需要，需要修改时多使用 []byte。
- 优先使用 strconv 而不是 fmt。

9.2 注意事项

- append 要小心自动分配内存，append 返回的可能是新分配的地址。
- 如果要直接修改 map 的 value 值，则 value 只能是指针，否则要覆盖原来的值。
- map 在并发中需要加锁。
- 编译过程无法检查 interface{} 的转换，只能在运行时检查，小心引起 panic。

总结

这一讲，我向你介绍了九类常用的编码规范。但今天的最后，我要在这里提醒你一句：规范是人定的，你也可以根据需要，制定符合你项目的规范。这也是我在之前的课程里一直强调的思路。但同时我也建议你采纳这些业界沉淀下来的规范，并通过工具来确保规范的执行。

今天的内容就到这里啦，欢迎你在下面的留言区谈谈自己的看法，我们下一讲见。

精选留言：

- 不明真相的群众 2021-06-18 09:59:23
  日常催更 [1赞]

- Geek_b797c1 2021-06-17 14:26:00
  催更：更新太慢了 [1赞]

- Vackine 2021-06-17 10:55:10
  在1.2初始化结构体引用时，给的案例里面bad 和good的sval的方式是一模一样的啊？还有在切片初始化时不都建议提前制定容量，然后后面为什么在声明slice是时候，又不建议make的方式而用var 的方式？ [1赞]

  作者回复2021-06-18 13:30:20
  如果不指定切片的cap，建议用var s []string

- 文涛 2021-06-19 12:38:06
  最近刚入坑go不久，就尴尬了，不知道为什么go.mod 没有类似package.json或者pom.xml之类，可以定义引入的依赖包范围，这样子可以将test 用例用到的包仅仅用于测试，不是在build里体现。谢谢

- action 2021-06-17 16:27:14
  建议一行代码不超过 120 行？？

  作者回复2021-06-18 13:21:47
  typo，我们更新下

- 不明真相的群众 2021-06-17 16:01:02
  作为一个新手，只能在实际应用的时候 再看翻看这章节

- 叫我去学习好么 2021-06-17 15:09:59
  为什么函数参数尽量不用指针传递？如果是一个比较大的结构体，传指针不是更好吗？

  作者回复2021-06-18 13:22:26
  如果传指针存在被意外修改的风险，如果结构体很大，也可以传指针

- dch666 2021-06-17 14:01:23
  判断空字符串用 len(s) == 0 比 s == "" 好在哪呢

  作者回复2021-06-18 13:24:26
  用 s == ""

- Geek_11667c 2021-06-17 09:42:07
  站在巨人的肩膀，get！

- 夏夜星语 2021-06-17 09:33:06
  请问我们现在的iam 项目代码已经完全写完了嘛，以后就都是这种理论课嘛？

  作者回复2021-06-18 13:31:09
  也不算是理论课，会介绍iam背后的设计思路、开发经验、知识等

- Geek_eedcba 2021-06-17 08:52:08
  老师要是能提供相关的反例就更好了

- pedro 2021-06-17 08:24:14
  终极规范：当遇到规范问题不知道如何处理时，立马查看文档！