# ChainMaker项目Golang 代码规范

[Golang 代码规范](#header-n0)  
 [1. 前言](#header-n3)  
 [2. 代码风格](#header-n13)  
 [2.1 【必须】格式化](#header-n14)  
 [2.2 【推荐】换行](#header-n18)  
 [2.3 【必须】括号和空格](#header-n22)  
 [2.4 【必须】import规范](#header-n30)  
 [2.5 【必须】错误处理](#header-n51)  
 [2.5.1 【必须】error处理](#header-n52)  
 [2.5.2 【必须】panic处理](#header-n77)  
 [2.5.3 【必须】recover处理](#header-n89)  
 [2.6 【必须】单元测试](#header-n96)  
 [2.7 【必须】类型断言失败处理](#header-n104)  
 [3. 注释](#header-n109)  
 [3.1 【必须】包注释](#header-n119)  
 [3.2 【必须】结构体注释](#header-n126)  
 [3.3 【必须】方法注释](#header-n137)  
 [3.4 【必须】变量&常量注释](#header-n146)  
 [3.5 【必须】类型注释](#header-n155)  
 [4. 命名规范](#header-n164)  
 [4.1 【推荐】包命名](#header-n166)  
 [4.2 【必须】文件命名](#header-n180)  
 [4.3 【必须】结构体命名](#header-n186)  
 [4.4 【推荐】接口命名](#header-n197)  
 [4.5 【必须】变量命名](#header-n210)  
 [4.6 【必须】常量命名](#header-n229)  
 [4.7 【必须】函数命名](#header-n242)  
 [5. 控制结构](#header-n248)  
 [5.1 【推荐】if](#header-n249)  
 [5.2 【推荐】for](#header-n262)  
 [5.3 【必须】range](#header-n267)  
 [5.4 【必须】switch](#header-n276)  
 [5.5 【推荐】return](#header-n281)  
 [5.6 【必须】goto](#header-n286)  
 [6. 函数](#header-n290)  
 [6.1 【推荐】函数参数](#header-n291)  
 [6.2 【必须】defer](#header-n305)  
 [6.3 【必须】方法的接收器](#header-n316)  
 [6.4 【推荐】代码行数](#header-n322)  
 [6.5 【必须】嵌套](#header-n328)  
 [6.6 【推荐】变量声明](#header-n334)  
 [6.7 【必须】魔法数字](#header-n338)  
 [7. 依赖管理](#header-n347)  
 [7.1 【必须】go1.11以上必须使用go modules模式：](#header-n348)  
 [7.2 【推荐】代码提交](#header-n350)  
 [8. 应用服务](#header-n359)  
 [8.1 【推荐】应用服务接口建议有README.md](#header-n360)  
 [8.2 【必须】应用服务必须要有接口测试。](#header-n364)  
 [附：常用工具](#header-n366)

## 1. 前言

为形成公司统一的Go编码风格，以保障公司项目代码的易维护性和编码安全性，特制定本规范。

每项规范内容，给出了要求等级，其定义为：

* **必须（Mandatory）**：用户必须采用；
* **推荐（Preferable）**：用户理应采用，但如有特殊情况，可以不采用；
* **可选（Optional）**：用户可参考，自行决定是否采用；

## 2. 代码风格

### 2.1 【必须】格式化

* 代码都必须用gofmt格式化。

### 2.2 【推荐】换行

* 建议一行代码不要超过120列，超过的情况，使用合理的换行方法换行。

### 2.3 【必须】括号和空格

* 遵循gofmt的逻辑。
* 运算符和操作数之间要留空格。
* 作为输入参数或者数组下标时，运算符和运算数之间不需要空格，紧凑展示。

### 2.4 【必须】import规范

* 使用goimports自动格式化引入的包名。
* goimports或者gofmt会自动把依赖包按首字母排序，并对包进行分组管理，默认分为本地包（标准库、内部包）、第三方包。
* 不要使用相对路径引入包：

// 不要采用这种方式  
import (  
 "../net"  
)

* 应该使用完整的路径引入包：

import (  
 "xxxx.com/proj/net"  
)

* 包名和git路径名不一致时，使用别名代替

import (  
 opentracing "github.com/opentracing/opentracing-go"  
)

* 【可选】匿名包的引用建议使用一个新的分组引入，并在匿名包上写上注释说明。

import (  
 // standard package & inner package  
 "encoding/json"  
 "myproject/models"  
 "myproject/controller"  
 "strings"  
   
 // third-party package  
 "git.obc.im/obc/utils"  
 "git.obc.im/dep/beego"  
 "git.obc.im/dep/mysql"  
 opentracing "github.com/opentracing/opentracing-go"  
  
 // anonymous import package  
 // import filesystem storage driver  
 \_ "git.code.oa.com/org/repo/pkg/storage/filesystem  
)

### 2.5 【必须】错误处理

#### 2.5.1 【必须】error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。
* 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。

// 不要采用这种方式  
func do() (error, int) {  
  
}  
  
// 要采用下面的方式  
func do() (int, error) {  
  
}

* 错误描述不需要标点结尾。
* 采用独立的错误流进行处理。

// 不要采用这种方式  
if err != nil {  
 // error handling  
} else {  
 // normal code  
}  
  
// 而要采用下面的方式  
if err != nil {  
 // error handling  
 return // or continue, etc.  
}  
// normal code

* 如果返回值需要初始化，则采用下面的方式：

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

* 错误返回的判断独立处理，不与其他变量组合逻辑判断。

// 不要采用这种方式：  
x, y, err := f()  
if err != nil || y == nil {  
 return err // 当y与err都为空时，函数的调用者会出现错误的调用逻辑  
}  
  
// 应当使用如下方式：  
x, y, err := f()  
if err != nil {  
 return err  
}  
if y == nil {  
 return fmt.Errorf("some error")  
}

* 【推荐】建议go1.13以上，error生成方式为：fmt.Errorf("module xxx: %w", err)

#### 2.5.2 【必须】panic处理

* 在业务逻辑处理中禁止使用panic。
* 在main包中只有当完全不可运行的情况可使用panic，例如：文件无法打开，数据库无法连接导致程序无法正常运行。
* 对于其它的包，可导出的接口不能有panic，只能在包内使用。
* 建议在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。
* panic捕获只能到goroutine最顶层，每个自行启动的goroutine，必须在入口处捕获panic，并打印详细堆栈信息或进行其它处理。

#### 2.5.3 【必须】recover处理

* recover用于捕获runtime的异常，禁止滥用recover。
* 必须在defer中使用，一般用来捕获程序运行期间发生异常抛出的panic或程序主动抛出的panic。

package main  
  
import (  
 "log"  
)  
  
func main() {  
 defer func() {  
 if err := recover(); err != nil {  
 // do something or record log  
 log.Println("exec panic error: ", err)  
 // log.Println(debug.Stack())  
 }  
 }()  
   
 getOne()  
   
 panic(11) //手动抛出panic  
}  
  
// getOne 模拟slice越界 runtime运行时抛出的panic  
func getOne() {  
 defer func() {  
 if err := recover(); err != nil {  
 // do something or record log  
 log.Println("exec panic error: ", err)  
 // log.Println(debug.Stack())  
 }  
 }()  
   
 var arr = []string{"a", "b", "c"}  
 log.Println("hello,", arr[4])  
}  
  
// 执行结果：  
// 2020/01/02 17:18:53 exec panic error: runtime error: index out of range  
// 2020/01/02 17:18:53 exec panic error: 11

### 2.6 【必须】单元测试

* 单元测试文件名命名规范为example\_test.go。
* 测试用例的函数名称必须以Test开头，例如TestExample。
* 每个重要的可导出函数都要首先编写测试用例，测试用例和正规代码一起提交方便进行回归测试。

### 2.7 【必须】类型断言失败处理

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

// 不要采用这种方式  
t := i.(string)  
  
// 而要采用下面的方式  
t, ok := i.(string)  
if !ok {  
 // 优雅地处理错误  
}

## 3. 注释

1. 在编码阶段同步写好变量、函数、包注释，注释可以通过godoc导出生成文档。
2. 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。
3. 程序中每一个被导出的(大写的)名字，都应该有一个文档注释。
4. 所有注释掉的代码在提交code review前都应该被删除，除非添加注释讲解为什么不删除， 并且标明后续处理建议(比如删除计划)。

### 3.1 【必须】包注释

* 每个包都应该有一个包注释。
* 包如果有多个go文件，只需要出现在一个go文件中（一般是和包同名的文件）即可，格式为：“// Package 包名 包信息描述”。

// Package math provides basic constants and mathematical functions.  
package math  
  
// 或者  
  
/\*  
Package template implements data-driven templates for generating textual  
output such as HTML.  
....  
\*/  
package template

### 3.2 【必须】结构体注释

* 每个需要导出的自定义结构体或者接口都必须有注释说明。
* 注释对结构进行简要介绍，放在结构体定义的前一行。
* 格式为："// 结构体名 结构体信息描述"。
* 结构体内的可导出成员变量名，如果是个生僻词，或者意义不明确的词，就必须要给出注释，放在成员变量的前一行。

// User 用户结构定义了用户基础信息  
type User struct {  
 Name string  
 Email string  
 // Demographic 族群  
 Demographic string  
}

### 3.3 【必须】方法注释

* 每个需要导出的函数或者方法（结构体或者接口下的函数称为方法）都必须有注释。
* 注释描述函数或方法功能、调用方等信息。
* 格式为："// 函数名 函数信息描述"。

// NewtAttrModel 是属性数据层操作类的工厂方法  
func NewAttrModel(ctx \*common.Context) \*AttrModel {  
 // TODO  
}

### 3.4 【必须】变量&常量注释

* 每个需要导出的常量和变量都必须有注释说明。
* 该注释对常量或变量进行简要介绍，放在常量或者变量定义的前一行或同一行的末尾。
* 格式为："// 变量名 变量信息描述"。

// FlagConfigFile 配置文件的命令行参数名  
const FlagConfigFile = "--config"  
  
const FlagConfigFile = "--config" // 配置文件的命令行参数名  
  
// FullName 返回指定用户名的完整名称  
var FullName = func(username string) string {  
 return fmt.Sprintf("fake-%s", username)  
}

### 3.5 【必须】类型注释

* 每个需要导出的类型定义(type definition)和类型别名(type aliases)都必须有注释说明。
* 该注释对类型进行简要介绍，放在定义的前一行。
* 格式为："// 类型名 类型信息描述"。

// StorageClass 存储类型  
type StorageClass string  
  
// FakeTime 标准库时间的类型别名  
type FakeTime = time.Time

## 4. 命名规范

命名是代码规范中很重要的一部分，统一的命名规范有利于提高代码的可读性，好的命名仅仅通过命名就可以获取到足够多的信息。

### 4.1 【推荐】包命名

* 保持 package 的名字和目录一致。
* 尽量采取有意义、简短的包名，尽量不要和标准库冲突。
* 包名应该为小写单词，不要使用下划线或者混合大小写，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。
* 简单明了的包命名，如：time、list、http。
* 不要使用无意义的包名，如：util、common、misc。

### 4.2 【必须】文件命名

* 采用有意义，简短的文件名。
* 文件名应该采用小写，并且使用下划线分割各个单词。

### 4.3 【必须】结构体命名

* 采用驼峰命名方式，首字母根据访问控制采用大写或者小写。
* 结构体名应该是名词或名词短语，如 Customer、WikiPage、Account、AddressParser。它不应是动词。
* 避免使用Data、Info 这类意义太宽泛的结构体名。
* 结构体的申明和初始化格式采用多行，例如：

// User 多行申明  
type User struct {  
 Name string  
 Email string  
}  
  
// 多行初始化  
u := User{  
 UserName: "john",  
 Email: "john@example.com",  
}

### 4.4 【推荐】接口命名

* 命名规则基本保持和结构体命名规则一致。
* 单个函数的结构名以 "er" 作为后缀，例如Reader，Writer。

// Reader 字节数组读取接口  
type Reader interface {  
 // Read 读取整个给定的字节数据并返回读取的长度  
 Read(p []byte) (n int, err error)  
}

* 两个函数的接口名综合两个函数名。
* 三个以上函数的接口名，类似于结构体名。

// Car 小汽车结构申明  
type Car interface {  
 // Start ...  
 Start([]byte)  
 // Stop ...  
 Stop() error  
 // Recover ...  
 Recover()  
}

### 4.5 【必须】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。
* 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；
  + 其他情况都应该使用该名词原有的写法，如APIClient、repoID、UserID；
  + 错误示例：UrlArray，应该写成urlArray或者URLArray；
* 若变量类型为bool类型，则名称应以Has，Is，Can或者Allow开头。
* 私有全局变量和局部变量规范一致，均以小写字母开头。
* 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)。

### 4.6 【必须】常量命名

* 常量均需遵循驼峰式。

// AppVersion 应用程序版本号定义  
const AppVersion = "1.0.0"

* 如果是枚举类型的常量，需要先创建相应类型：

// Scheme 传输协议  
type Scheme string  
  
const (  
 // HTTP 表示HTTP明文传输协议  
 HTTP Scheme = "http"  
 // HTTPS 表示HTTPS加密传输协议  
 HTTPS Scheme = "https"  
)

* 私有全局常量和局部变量规范一致，均以小写字母开头。

const appVersion = "1.0.0"

### 4.7 【必须】函数命名

* 函数名必须遵循驼峰式，首字母根据访问控制决定使用大写或小写。
* 代码生成工具自动生成的代码可排除此规则(如协议生成文件xxx.pb.go, gotests自动生成文件xxx\_test.go里面的下划线)。

## 5. 控制结构

### 5.1 【推荐】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  
}

* if 对两个值进行判断时，约定如下顺序：变量在左，常量在右

// 不要采用这种方式  
if nil != err {  
 // error handling  
}  
  
// 不要采用这种方式  
if 0 == errorCode {  
 // do something  
}  
  
// 而要采用下面的方式  
if err != nil {  
 // error handling  
}   
  
// 而要采用下面的方式  
if errorCode == 0 {  
 // do something  
}

* if 对于bool类型的变量，应直接进行真假判断

var allowUserLogin bool  
// 不要采用这种方式  
if allowUserLogin == true {  
 // do something  
}  
  
// 不要采用这种方式  
if allowUserLogin == false {  
 // do something  
}  
  
// 而要采用下面的方式  
if allowUserLogin {  
 // do something  
}  
  
// 而要采用下面的方式  
if !allowUserLogin {  
 // do something  
}

### 5.2 【推荐】for

* 采用短声明建立局部变量：

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

### 5.3 【必须】range

* 如果只需要第一项（key），就丢弃第二个：

for key := range m {  
 if key.expired() {  
 delete(m, key)  
 }  
}

* 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 【必须】switch

* 要求必须有 default：

switch os := runtime.GOOS; os {  
 case "darwin":  
 fmt.Println("OS X.")  
 case "linux":  
 fmt.Println("Linux.")  
 default:  
 // freebsd, openbsd,  
 // plan9, windows...  
 fmt.Printf("%s.\n", os)  
}

### 5.5 【推荐】return

* 尽早return，一旦有错误发生，马上返回：

f, err := os.Open(name)  
if err != nil {  
 return err  
}  
  
d, err := f.Stat()  
if err != nil {  
 f.Close()  
 return err  
}  
  
codeUsing(f, d)

### 5.6 【必须】goto

* 业务代码禁止使用goto，其他框架或底层源码推荐尽量不用

## 6. 函数

### 6.1 【推荐】函数参数

* 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，其它情况不建议使用命名返回。

// Parent1 ...  
func (n \*Node) Parent1() \*Node  
  
// Parent2 ...  
func (n \*Node) Parent2() (\*Node, error)  
  
// Location ...  
func (f \*Foo) Location() (lat, long float64, err error)

* 传入变量和返回变量以小写字母开头。
* 参数数量均不能超过5个。
* 尽量用值传递，非指针传递。
* 传入参数是map，slice，chan，interface不要传递指针。

### 6.2 【必须】defer

* 当存在资源管理时，应紧跟defer函数进行资源的释放。
* 判断是否有错误发生之后，再defer释放资源。

resp, err := http.Get(url)  
if err != nil {  
 return err  
}  
// 如果操作成功，再defer Close()  
defer resp.Body.Close()

* 禁止在循环中使用defer，举例如下：

// 不要这样使用  
func filterSomething(values []string) {  
 for \_, v := range values {  
 fields, err := db.Query(v) // 示例，实际不要这么查询，防止sql注入  
 if err != nil {  
 // xxx  
 }  
 defer fields.Close()  
 // 继续使用fields  
 }  
}  
  
// 应当使用如下的方式：  
func filterSomething(values []string) {  
 for \_, v := range values {  
 func() {  
 fields, err := db.Query(v) // 示例，实际不要这么查询，防止sql注入  
 if err != nil {  
 ...  
 }  
 defer fields.Close()  
 // 继续使用fields  
 }()  
 }  
}

### 6.3 【必须】方法的接收器

* 接收器的命名在函数超过20行的时候不要用单字符。
* 命名不能采用me，this，self这类易混淆名称。

### 6.4 【推荐】代码行数

* 【必须】文件长度不能超过800行。
* 【推荐】函数长度不能超过80行。

### 6.5 【必须】嵌套

* 嵌套深度不能超过4层：

// AddArea 添加成功或出错  
func (s \*BookingService) AddArea(areas ...string) error {  
 s.Lock()  
 defer s.Unlock()  
   
 for \_, area := range areas {  
 for \_, has := range s.areas {  
 if area == has {  
 return srverr.ErrAreaConflict  
 }  
 }  
 s.areas = append(s.areas, area)  
 s.areaOrders[area] = new(order.AreaOrder)  
 }  
 return nil  
}

// 建议调整为这样：  
  
// AddArea 添加成功或出错  
func (s \*BookingService) AddArea(areas ...string) error {  
 s.Lock()  
 defer s.Unlock()  
   
 for \_, area := range areas {  
 if s.HasArea(area) {  
 return srverr.ErrAreaConflict  
 }  
 s.areas = append(s.areas, area)  
 s.areaOrders[area] = new(order.AreaOrder)  
 }  
 return nil  
}  
  
// HasArea ...  
func (s \*BookingService) HasArea(area string) bool {  
 for \_, has := range s.areas {  
 if area == has {  
 return true  
 }  
 }  
 return false  
}

### 6.6 【推荐】变量声明

* 变量声明尽量放在变量第一次使用前面，就近原则。

### 6.7 【必须】魔法数字

* 如果魔法数字出现超过2次，则禁止使用。

func getArea(r float64) float64 {  
 return 3.14 \* r \* r  
}  
func getLength(r float64) float64 {  
 return 3.14 \* 2 \* r  
}

* 用一个常量代替：

// PI ...  
const PI = 3.14  
  
func getArea(r float64) float64 {  
 return PI \* r \* r  
}  
  
func getLength(r float64) float64 {  
 return PI \* 2 \* r  
}

## 7. 依赖管理

### 7.1 【必须】go1.11以上必须使用go modules模式：

go mod init git.code.oa.com/group/myrepo

### 7.2 【推荐】代码提交

* 建议所有不对外开源的工程的module name使用git.code.oa.com/group/repo，方便他人直接引用
* 建议使用go modules作为依赖管理的项目不提交vendor目录
* 建议使用go modules管理依赖的项目，不要将go.sum文件添加到.gitignore规则中

## 8. 应用服务

### 8.1 【推荐】应用服务接口建议有README.md

* 其中建议包括服务基本描述、使用方法、部署时的限制与要求、基础环境依赖（例如最低go版本、最低外部通用包版本）等。

### 8.2 【必须】应用服务必须要有接口测试。

## 附：常用工具

go 语言本身在代码规范性这方面也做了很多努力，很多限制都是强制语法要求，例如左大括号不换行，引用的包或者定义的变量不使用会报错，此外 go 还是提供了很多好用的工具帮助我们进行代码的规范。

* gofmt，大部分的格式问题可以通过gofmt解决， gofmt 自动格式化代码，保证所有的 go 代码与官方推荐的格式保持一致，于是所有格式有关问题，都以 gofmt 的结果为准。
* goimports ，此工具在 gofmt 的基础上增加了自动删除和引入包。
* go vet ，vet工具可以帮我们静态分析我们的源码存在的各种问题，例如多余的代码，提前return的逻辑，struct的tag是否符合标准等。编译前先执行代码静态分析。
* golint ，类似javascript中的jslint的工具，主要功能就是检测代码中不规范的地方。