

蓝鲸 Golang 开发规范

BlueKing Golang Code Convention

版本：v1.0.0



目录

1	前言	1
2	编程规约	2
2.1	命名风格	2
2.2	常量定义	5
2.3	代码格式	5
2.4	控制语句	6
2.5	注释规约	7
2.6	其它	9
3	异常	11
3.1	异常处理	11
3.2	日志规约	11
4	单元测试	12
5	版权声明	13
6	工程结构	13
附 1	版本历史	15
附 2	参考文档	15

Golang 开发规范

1 前言

代码的质量是软件产品质量的生命线，而开发规范是构建高质量代码的基石。

现代软件架构的系统性和复杂性需要多个组织、个人协同才能完成开发。那么如何高效的协同？构建高质量的代码，提高软件产品的质量呢？无规矩不成方圆，一个好的开发规范可以帮助我们向前更进一步。特别是随着开源社区的日趋活跃，统一风格、规范的代码也是我们和社区参与者沟通的途径和桥梁。

Golang 开发规范包含了编程规约、异常、单元测试、版权声明、工程结构等部分。根据约束力和重要程度，又将规约划分为“**强制**”和“**推荐**”两大类。对于部分规约，给出了提倡的、需要遵守的“**正例**”，部分规约给出了不建议使用、禁止使用的“**反例**”，目的是为了更好的解释这些规约。

制定 Golang 开发规范是为了敦促我们构建简洁、可维护、可测试、高效的代码，降低沟通和协调成本，提高版本交付质量和开发效率。

在编写规范的过程中参考了 Golang 官方社区相关的文档、规范，也参考了网络上的一些琐碎的文章，在此一并谢过。

2018.10.4

2 编程规约

2.1 命名风格

- 1) 【强制】代码中的命名均不能以**特殊字符**开始和结束, 包含常见的中划线、下划线等。

反例： `_name`; `-name`; `__name`; `--name`; `$name`; `%name`

- 2) 【强制】命名统一采用**英文**, 不能采用拼音等其他方式。

正例： `blueking`; `name`; `address`

反例： `lanjing`; `mingzi`; `dizhi`

- 3) 【强制】参数名、局部变量都统一使用 **lowerCamelCase(小驼峰)**风格。

正例：

```
func demo(ctx context.Context, name string) {  
    var localVar string  
    // other operations  
}
```

- 4) 【强制】全局变量统一使用**小驼峰**风格。包外引用需要提供相应的导出函数对外导出使用。

正例：

```
var globalVar string  
func Set(value string) {  
    globalVar = value  
}
```

- 5) 【强制】函数名和方法(method)的命名区分导出和非导出类型。对于包内非导出的函数或方法使用小驼峰风格。对于导出的函数或方法采用 **UpperCamelCase(大驼峰)**风格。

正例：

```
func localFunction() {  
    // do something  
}  
  
type sample struct {}  
func (s *sample) Show() {  
    fmt.Println("Hello Blueking.")  
}
```

6) 【强制】struct 的命名区分导出和非导出类型。对于包内非导出的 struct 使用小驼峰风格。对于导出的 struct 采用大驼峰风格。

7) 【强制】常量命名使用大驼峰风格，不要使用下划线分隔，力求语义表达完整清楚，不要嫌名字长。

正例：MaxConnectionCount

反例：MAX_CNNECTION_COUNT

8) 【强制】接口(interface)采用大驼峰的风格命名。具体细分为以下三种情况：

单个函数的接口名以“er”作为后缀，如 Reader, Writer。而接口的实现则去掉“er”。

正例：

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

两个函数的接口名缩合两个函数名

正例：

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

三个以上函数的接口名，类似于结构体名

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

9) 【强制】包名统一采用小写风格，使用短命名，不能包含特殊字符（下划线、中划线等）语义表达准确。建议最好是一个单词。

正例：

```
package client
package clientset
```

反例：

```
package Demo
package Demo_Case
```

10) 【强制】杜绝不规范的缩写，避免望文不生意。不必要进行缩写时，避免进行缩写。

对于专用名词如 URL/CMDB/GSE 等，在使用时要保证全名统一为大写或小写。不能出现部分大写和小写混用的情况。

正例：

```
func setURL() {
    // do something
}

func GetURL() {
    // do something
}
```

反例：

```
var sidecarUrl string

func ResetUrl() {
    // do something
}
```

11) 【推荐】遵循代码自注释的原则，在对变量、struct、interface 等进行命名时，推荐使用尽量完整的单词组合来表达其准确含义。

正例：

```
var agentName, userName string
type UserClientSetInterface interface {
    Create(name string) error
    Delete(name string) error
}
```

反例：

```
var aName, uName string
type Client interface {
    Create(name string) error
    Delete(name string) error
}
```

- 12) 【推荐】如果包、struct、interface 等使用了设计模式，在命名时需要体现出对应的设计模式。这有利于阅读者快速理解代码的设计架构和设计理念。

正例：

```
type ContainerFactory interface {
    Create(id string, config *configs.Config) (Container, error)
    Load(id string) (Container, error)
    StartInitialization() error
    Type() string
}

type PersonDecorator interface {
    SetName(name string) PersonDecorator
    SetAge(age uint) PersonDecorator
    Show()
}
```

2.2 常量定义

- 1) 【强制】对于多个具有枚举特性的类型，要求定义成为类型，并利用常量进行枚举。

正例：

```
type EventType string
const (
    Create EventType = "create"
    Update EventType = "update"
    Get     EventType = "get"
    Delete  EventType = "delete"
)
```

- 2) 【强制】不允许任何未经定义的常量直接在代码中使用。

2.3 代码格式

- 1) 【强制】采用 4 个空格的缩进，每个 tab 也代表 4 个空格。这是唯一能够保证在所有环境下获得一致展现的方法。
- 2) 【推荐】运算符(:=, =等)的左右两侧必须要加一个空格。
- 3) 【强制】提交的代码必须经过 gofmt 格式化。很多 IDE 支持自动 gofmt 格式化。

- 4) 【推荐】代码最大行宽为 120 列，超过换行。

2.4 控制语句

2.4.1 If

- 1) 【强制】if 接受一个初始化语句，对于返回参数不需要流入到下一个语句时，通过建立局部变量的方式构建 if 判断语句。

正例：

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

- 2) 【强制】在 if 语句中对异常处理等情况，如果成功的控制流是继续往下走，而对于异常处理等结束于 return 语句时，不能使用 else 语句。这样代码会非常易读。

正例：

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

反例：

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

```
    } else {  
        codeUsing(f, d)  
    }  
}
```

2.4.2 for

- 1) 【强制】采用简短声明建立局部变量。

正例：

```
for i := 0; i < 5; i++ {  
    codeUsing(i)  
}
```

反例：

```
var i int  
for i = 0; i < 5; i++ {  
    codeUsing(i)  
}
```

- 2) 【强制】对于遍历数据(如 map)的场景，如果只使用第一项，则直接丢弃第二项。

正例：

```
for key := range mapper {  
    codeUsing(key)  
}
```

反例：

```
for key, _ := range mapper {  
    codeUsing(key)  
}
```

2.5 注释规约

- 1) 【强制】注释必须是完整的句子，以句点作为结尾。
- 2) 【强制】使用行间注释时，如果注释行与上一行不属于同一层，不用空行。如果属于同行，则空一行再进行注释。

```
func demo() {
    // This is a start line of a new block, do not need a new line
    // with the previous code.
    say("knock, knock!")

    // This is the same block with the previous code,
    // you should insert a new line before you start a comment.
    echo("who is there ?")
}
```

- 3) 【强制】使用 `//` 进行注释时, 和注释语句之间必须有一个空格。增加可读性。

正例：

```
// validator is used to validate dns's format.
// should not contains dot, underscore character, etc.
func validator(dns string) error {
    // do validate.
}
```

- 4) 【强制】不要使用尾注释。

反例：

```
func show(name string) {
    display(name) // show a man's information
}
```

- 5) 【强制】使用 `/* */` 风格进行多行注释时, 首 `/*` 和尾 `*/` 两行内容中不能包含注释内容, 也不能包含额外的内容, 如星号横幅等。

正例：

```
/*
The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
*/
```

```
'(' regexp ')'
*/
```

- 6) **【强制】** 注释的单行长度最大不能超过 120 列，超过必须换行。一般以 80 列换行为宜。
- 7) **【推荐】** 注释内容尽量采用英文注释。不建议使用中文或其它语言。
- 8) **【推荐】** 函数与方法的注释需要以函数或方法的名称作为开头。

正例：

```
// HasPrefix tests whether the string s begins with prefix.
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[0:len(prefix)] == prefix
}
```

- 9) **【推荐】** 大段注释采用/* */风格进行注释。
- 10) **【推荐】** 包中的每一个导出的函数或方法都应该有相应的注释。
- 11) **【推荐】** 对于特别复杂的包说明，可以单独创建 [doc.go](#) 文件来详细说明。

2.6 其它

2.6.1 参数传递

- 1) **【推荐】** 对于少量数据，不要通过指针传递。
- 2) **【推荐】** 对于大量(≥ 4)的入参，考虑使用 struct 进行封装，并通过指针传递。
- 3) **【强制】** 传参是 map, slice, chan 不要使用指针进行传递。因为这三者是引用类型。

2.6.2 接受者(receiver)

- 1) **【推荐】** 名称统一采用 1~3 个字母，不宜太长。
- 2) **【推荐】** 对于绝在多数可以使用指针接受者的场景，推荐使用指针接受者(point

receiver)会更有效率。

- 3) 【强制】如果接受者是 map, slice, chan, 不能使用指针接受者。

正例：

```
package main

import (
    "fmt"
)

type queue chan interface{}

func (q queue) Push(i interface{}) {
    q <- i
}

func (q queue) Pop() interface{} {
    return <-q
}

func main() {
    c := make(queue, 1)
    c.Push("i")
    fmt.Println(c.Pop())
}
```

- 4) 【强制】如果接受者是包含有锁(sync.Mutex 等), 必须使用指针接受者。

2.6.3 struct 定义

- 1) 【强制】对于对外提供的 SDK 包中需要导出的 struct 结构, 必须添加编译保护。避免用户无意识引起的初始化错误。

正例：

```
type Person struct {
    Name string
    Address string
    ID uint
    Age uint
}
```

```
// this is a protection field which will let  
// the compiler catch all the unconverted  
// literals.  
_ struct{}  
}
```

3 异常

3.1 异常处理

- 1) **【强制】** 程序中出现的任何异常都必须处理，不能忽略。
- 2) **【强制】** 错误描述必须为小写，不需要标点结尾。
- 3) **【推荐】** 程序中尽量避免使用 panic 来进行异常处理。对于必须要使用 panic 进行异常处理的场景，应该保证能够在单元测试或集成测试中覆盖到此场景。同时要在非测试场景下，启用 recover 能力。

3.2 日志规约

- 1) **【强制】** 日志采用分级打印方式，包含 Info, Warning, Error 和自定义等级。统一使用 blog 日志包进行日志的管理。
- 2) **【强制】** 日志的内容要详尽，至少包含这几个要素：谁在什么情况下，因为什么原因，出现了什么异常，会引起什么问题。方便异常的定位。
- 3) **【强制】** Info 级别用于打印在程序运行过程中必须要打印的日志信息。不能包含调试等日志信息。
- 4) **【强制】** Warning 级别用于打印程序运行过程中出现的异常，但不影响程序的正常运行，需要通过 Warning 级别日志进行提示。

- 5) **【强制】** Error 级别用于打印程序运行过程中出现的会影响业务正常运行逻辑的异常。
- 6) **【推荐】** 对于自定义等级的日志，默认 3 级为 debug 日志。自定义日志的级别可根据自身需求进行调整。
- 7) **【推荐】** 底层公共库中的异常应该抛出，不建议在公共库中打印相关的日志信息，应该由上层的逻辑层处理异常，并打印日志信息。

4 单元测试

- 1) **【强制】** 单元测试必须遵守 AIR 原则，即具有自动化、独立性、可重复执行的特点。
- 2) **【强制】** 单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。输出结果禁止进行人肉验证。
- 3) **【强制】** 保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间禁止互相调用，也不能依赖执行的先后次序。
- 4) **【强制】** 新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。
- 5) **【推荐】** 对于不可测的代码建议做必要的重构，使代码变得可测，避免为了达到测试要求而编写不规范测试代码。
- 6) **【推荐】** 在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好 覆盖所有测试用例。

5 版权声明

1) **【强制】**对于开源的项目，必须在每个文件头中添加对应的版本声明。可以借助于 IDE 中的自动添加功能，为每个文件添加版权声明头。以 cmdb 的开源声明为例：

正例：

```
/*
 * Tencent is pleased to support the open source community by making 蓝鲸
available.
 * Copyright (C) 2017-2018 THL A29 Limited, a Tencent company. All
rights reserved.
 * Licensed under the MIT License (the "License"); you may not use this
file except
 * in compliance with the License. You may obtain a copy of the License
at
 * http://opensource.org/licenses/MIT
 * Unless required by applicable law or agreed to in writing, software
distributed under
 * the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND,
 * either express or implied. See the License for the specific language
governing permissions and
 * limitations under the License.
 */
```

6 工程结构

1) **【强制】**工程目录名称和文件名只能使用英文小写字母命名，如果有多个单词，中间可以使用下划线进行分隔。命名尽量望文生义。

2) **【强制】**工程中引入(import)的包，不能使用相对路径。必须使用相对于 GOPATH 的完整路径。

反例：

```
import (
    "fmt"
```

```
"errors"

"../../apimachinery/discovery"
)
```

3) 【强制】工程中引入的包，需要按照“标准库包、工程内部包、第三方包”的顺序进行组织。三种包之间用空行进行分隔，这样在 gofmt 时不会打乱三种包之间的顺序。

正例：

```
import (
    "fmt"
    "context"
    "errors"

    "configcenter/src/apimachinery/discovery"
    "configcenter/src/apimachinery/rest"
    "configcenter/src/apimachinery/util"

    "github.com/juju/ratelimit"
)
```

附 1：版本历史

版本号	更新日期	维护者	备注
V1.0.0	2018.10.4	蓝鲸	整理发布第一个版本。共七个章节。

附 2：参考文档

- 1) https://golang.org/doc/effective_go.html
- 2) <https://github.com/golang/go/wiki/CodeReviewComments>