

1.1 百度Golang编码规范V1.1

- 1. 前言
 - 1.1. 一般信息【重要必读】
 - 1.2. 如何使用本编程规范
 - 本规范的层次结构
 - 条目的级别和编号
 - 1.3. 说明
- 2. 语言规范
 - 2.1 true/false求值
 - 2.2 Receiver
 - 2.2.1 Receiver Type
 - 2.2.2 receiver命名
 - 2.3 申明空Slices
 - 2.4 Error Handler
 - 2.5 "{}"的使用
 - 2.6 embedding的使用
- 3. 风格规范
 - 3.1 Go文件Layout
 - 3.2 General Documentation Layout
 - 3.3 import规范
 - 3.4 Go函数Layout
 - 3.4.1 函数注释
 - 3.4.2 函数参数和返回值
 - 3.5 程序规模
 - 3.6 命名规范
 - 3.6.1 文件名
 - 3.6.2 函数名/变量名
 - 3.6.3 常量
 - 3.6.4 缩写词
 - 3.7 缩进
 - 3.8 空格
 - 3.9 括号
 - 3.10 注释
- 4. 编程实践
 - 4.1 error string
 - 4.2 Don't panic
 - 4.3 关于lock的保护
 - 4.4 日志的处理
 - 4.5 unsafe package

1. 前言

1.1. 一般信息【重要必读】

- 此编码风格指南主要基于BFE组Golang编程规范和CodeReviewComments制定。
- 这份文档存在的意义是让大家写出统一风格的代码，让百度的模块可维护性和可读性更好；
- 文档内容可能会与您的喜好冲突， 请尽量用包容的心态来接受; 不合理之处， 请反馈给 go-styleguide@baidu.com

1.2. 如何使用本编程规范

本规范的层次结构

- 本规范可分为三大部分，分别对Go语法、风格、编程实践作出规定与建议。
- 每一部分有若干专题，每一专题下有若干条目。
- 条目是规范的基本组成部分，每一条目由规定、定义、解释、示例、参考等项组成。

条目的级别和编号

- 本规范的条目分两个级别
 - **RULE**: 要求所有程序必须遵守，不得违反
 - **ADVICE**: 建议遵守，除非确有特殊情况
- 本规范所有条目都有编号，用来标识单条规范并用于自动化检查工具错误提示。

- 标号由"RULE"或"ADVICE"和三位数字标识，例如RULE001, ADVICE001

1.3. 说明

- 本规范由“百度Golang编程规范小组制定”，小组成员请参考[golang编程规范工作组](#)
- 特别感谢参与规范修订的以下同学：陶春华(taochunhua)、简恒懿(jianhengyi)、黄添来(huangtianlai01)、李景晨(lijingchen)、王炜煜(wangweiyu02)、李晓萌(lixiaomeng01)、赵新(zhaoxin08)。

2. 语言规范

2.1 true/false求值

- {RULE001} 当明确expr为bool类型时，禁止使用==或!=与true/false比较，应该使用expr或!expr
- {RULE002} 判断某个整数表达式expr是否为零时，禁止使用!expr，应该使用expr == 0

示例



GOOD:

```
var isWhiteCat bool
var num int
```

```
if isWhiteCat {
    // ...
}
```

```
if num == 0 {
    // ...
}
```



BAD:

```
var isWhiteCat bool
var num int
```

```
if isWhite == true {
    // ...
}
```

```
if !num {
    // ...
}
```

2.2 Receiver

2.2.1 Receiver Type

- {RULE003} 如果receiver是map、函数或者chan类型，类型不可以是指针
- {RULE004} 如果receiver是slice，并且方法不会进行reslice或者重新分配slice，类型不可以是指针
- {RULE005} 如果receiver是struct，且包含sync.Mutex类型字段，则必须使用指针避免拷贝。
- {ADVICE001} 如果receiver是比较大的struct/array，建议使用指针，这样会更有效率
- {ADVICE002} 如果receiver是struct、array或slice，其中指针元素所指的内容可能在方法内被修改，建议使用指针类型
- {ADVICE003} 如果receiver是比较小的struct/array，建议使用value类型

解释

- 关于receiver的定义详见[Receiver定义](#)：The receiver is specified via an extra parameter section preceeding the method name. That parameter section must declare a single parameter, the receiver. Its type must be of the form T or *T (possibly using parentheses) where T is a type name. The type denoted by T is called the receiver base type; it must not be a pointer or interface type and it must be declared in the same package as the method. The method is said to be bound to the base type and the method name is visible only within selectors for that type.
- struct或者array中的元素个数超过3个，则认为比较大，反之，则认为比较小

2.2.2 receiver命名

- {ADVICE004} 尽量简短并有意义。

- {RULE006} 禁止使用 “this”、” self “等面向对象语言中特定的叫法。
- {ADVICE005} receiver的命名要保持一致性

示例



GOOD:

```
// call()和done()都使用了在上下文中有意义的"c"进行receiver命名
func (c Client) call() error {
    // ...
}

func (c Client) done() error {
    // ...
}
```



BAD:

```
// 1. "c"和"client"命名不一致: done()用了c, call()用了client
// 2. client命名过于冗余

func (c Client) done() error {
    // ...
}

func (client Client) call() error {
    // ...
}

// 不允许使用self
func (self Server) rcv() error {
    // ...
}

// 不允许使用this
func (this Server) call() error {
    // ...
}
```

2.3 申明空Slices

- {ADVICE006} 申明slice时，建议使用var方式申明，不建议使用大括号的方式
- var方式申明在slice不被append的情况下避免了内存分配

解释

示例



GOOD:

```
var t []string
```



BAD:

```
t := []string{}
```

2.4 Error Handler

- {RULE007} 对于返回值中的error，一定要进行判断和处理，不可以使用 ” _ “ 变量忽略error

2.5 "{"的使用

- {RULE008} struct、函数、条件判断中的“{”，不可以作为独立的一行

示例



GOOD:
if condition {
 // ...
} else {
 // ...
}



BAD:
// "{"不可以作为独立行
if condition
{
 // ...
} else
{
 // ...
}

2.6 embedding的使用

- {ADVICE007} embedding只用于"is a"的语义下，而不用于"has a"的语义下
- {ADVICE008} 一个定义内，多于一个的embedding尽量少用

解释

- 语义上embedding是一种“继承关系”，而不是“成员关系”
- 一个定义内有多个embedding，则很难判断某个成员变量或函数是从哪里继承得到的
- 一个定义内有多个embedding，危害和在python中使用“from xxx import *”是类似的

示例



GOOD:
type Automobile struct {
 // ...
}

type Engine struct {
 //
}

// 正确的定义
type Car struct {
 Automobile // Car is a Automobile
 engine Engine // Car has a Engine
}



BAD:
type Car struct {
 Automobile // Car is a Automobile
 Engine // Car has a Engine, but Car is NOT a Engine
}

3. 风格规范

3.1 Go文件Layout

- {ADVICE009} 建议文件按以下顺序进行布局
 - General Documentation: 对整个模块和功能的完整描述注释, 写在文件头部。
 - package: 当前package定义
 - imports: 包含的头文件
 - Constants: 常量
 - Typedefs: 类型定义
 - Globals: 全局变量定义
 - functions: 函数实现

示例



GOOD:

```
/* Copyright 2015 Baidu Inc. All Rights Reserved. */
/* bfe_server.go - the main structure of bfe-server */
/*
modification history
-----

2014/6/5, by Zhang San, create
*/
/*
DESCRIPTION
This file contains the most important struct 'BfeServer' of go-bfe and new/init method of the struct.
*/

package bfe_server

// imports
import (
    "fmt"
    "time"
)

import (
    "code.google.com/p/log4go"

    "icode.baidu.com/baidu/searchbox/golib"
)

import (
    "bfe_config/bfe_conf"
    "bfe_module"
)

const (
    version = "1.0.0.0"
)

// typedefs
type BfeModule interface {
    Init(cfg bfe_conf.BfeConfig, cbs *BfeCallbacks) error
}

type BfeModules struct {
    modules map[string]BfeModule
}

// vars
var errTooLarge = errors.New("http: request too large")

//functions
func foo() {
    //...
}
```

- {RULE009} 对于以上的各个部分，采用单个空行分割，同时：
 - 多个类型定义采用单个空行分割
 - 多个函数采用单个空行分割
- {ADVICE010} 函数内不同的业务逻辑处理建议采用单个空行分割
- {ADVICE011} 常量或者变量如果较多，建议按照业务进行分组，组间用单个空行分割


3.2 General Documentation Layout

- {ADVICE012} 建议每个文件开头部分包括文件copyright说明（copyright）
- {ADVICE013} 建议每个文件开头部分包括文件标题（Title）
- {ADVICE014} 建议每个文件开头部分包括修改记录（Modification History）
- {ADVICE015} 建议每个文件开头部分包括文件描述（Description）

解释

- Title中包括文件的名称和文件的简单说明
 - Title应该在一行内完成
- Modification History记录文件的修改过程，并且只记录最主要的修改
 - 当书写新的函数模块时，只需要使用形如"Add func10"这样的说明
 - 如果后面有对函数中的算法进行了修改，需要指出修改的具体位置和修改方法
 - Modification History的具体格式为：<修改时间>, <修改人>, <修改动作>
- Description 详细描述文件的功能和作用

示例

 GOOD:

```
/* Copyright 2015 Baidu Inc. All Rights Reserved. */
/* bfe_server.go - the main structure of bfe-server */
/*
modification history
-----
2014/6/5, by Zhang San, create
*/
/*
DESCRIPTION
This file contains the most important struct 'BfeServer' of go-bfe and new/init method of the struct.
*/

package bfe_server

func func10() {
    // ...
}
```

 BAD:


```
package bfe_server

func func10() {
    // ...
}
```

3.3 import规范

- {RULE010} 需要按照如下顺序进行头文件import，并且每个import部分内的package需按照字母升序排列
 - 系统package
 - 第三方的package
 - 程序自己的package
- {RULE011} 每部分import间用单个空行进行分隔

示例：



```

GOOD:
import (
    "fmt"
    "time"
)

import (
    "code.google.com/p/log4go"

    "icode.baidu.com/baidu/searchbox/golib"
)

import (
    "bfe_config/bfe_conf"
    "bfe_module"
)

GOOD:
import (
    "fmt"
    "time"

    "code.google.com/p/log4go"

    "bfe_config/bfe_conf"
    "bfe_module"
)

```



```

BAD:
// 同一类package内import顺序需按字母升序排序
import (
    "time"
    "fmt"
)

// 不同类的package import顺序出错（先第三方package，再程序自己的package）
import (
    "bfe_config/bfe_conf"
    "bfe_module"
)

import (
    "code.google.com/p/log4go"
)

BAD:
import (
    // 同一类package内import顺序出错
    "time"
    "fmt"

    // 不同类的package import顺序出错（先第三方package，再程序自己的package）
    "bfe_config/bfe_conf"
    "bfe_module"

    "code.google.com/p/log4go"
)

```

3.4 Go函数Layout

3.4.1 函数注释

- {ADVICE016} 函数的注释，建议包括以下内容
 - **Description:** 对函数的完整描述，主要包括函数功能和使用方法
 - **Params:** 对参数的说明
 - **Returns:** 对返回值的说明

示例

```

✓ /*
 * Init - initialize log lib
 *
 * PARAMS:
 * - levelStr: "DEBUG", "TRACE", "INFO", "WARNING", "ERROR", "CRITICAL"
 * - when:
 *   "M", minute
 *   "H", hour
 *   "D", day
 *   "MIDNIGHT", roll over at midnight
 * - backupCount: If backupCount is > 0, when rollover is done, no more than
 *   backupCount files are kept - the oldest ones are deleted.
 *
 * RETURNS:
 * nil, if succeed
 * error, if fail
 */
func Init(levelStr string, when string, backupCount int) error {
    // ...
}

```

3.4.2 函数参数和返回值

- {ADVICE017} 对于“逻辑判断型”的函数，返回值的意义代表“真”或“假”，返回值类型定义为bool
- {ADVICE018} 对于“操作型”的函数，返回值的意义代表“成功”或“失败”，返回值类型定义为error
 - 如果成功，则返回nil
 - 如果失败，则返回对应的error值
- {ADVICE019} 对于“获取数据型”的函数，返回值的意义代表“有数据”或“无数据/获取数据失败”，返回值类型定义为（data, error）
 - 正常情况下，返回为：（data, nil）
 - 异常情况下，返回为：（data, error）
- {RULE012} 函数返回值小于等于3个，大于3个时必须通过struct进行包装
- {ADVICE020} 函数参数不建议超过3个，大于3个时建议通过struct进行包装

示例：

```

✓ GOOD:
type student struct {
    name  string
    email string
    id    int
    class string
}

// bool作为逻辑判断型函数的返回值
func isWhiteCat() bool {
    // ...
}

// error作为操作型函数的返回值
func deleteData() error {
    // ...
}

// 利用多返回值的语言特性
func getData() (student, error) {
    // ...
}

```




BAD:

```
type student struct {
    name    string
    email   string
    id      int
    class   string
}

// 使用int而非bool作为逻辑判断函数的返回值
func isWhiteCat() int {
    // ...
}

// 操作型函数没有返回值
func deleteData() {
    // ...
}

// 没有充分利用go多返回值的特点
func getData() student {
    // ...
}

// 返回值>3
func getData() (string, string, int, string, error) {
    // ...
}
```

3.5 程序规模

- {RULE013} 每行代码不超过100个字符。
- {RULE014} 每行注释不超过100个字符。
- {ADVICE021} 函数不超过100行。
- {ADVICE022} 文件不超过2000行。

解释

- 现在宽屏比较流行，所以从传统的80个字符限制扩展到100个字符
- 函数/文件太长一般说明函数定义不明确/程序结构划分不合理，不利于维护

3.6 命名规范

3.6.1 文件名

- {RULE015} 文件名都使用小写字母，如果需要，可以使用下划线分割
- {RULE016} 文件名的后缀使用小写字母

示例:



GOOD:

```
// 可以使用下划线分割文件名
web_server.go

// 文件名全部小写
http.go
```



BAD:

```
// 文件名不允许出现大写字母
webServer.go

// 文件名后缀不允许出现大写字母
http.GO
```

3.6.2 函数名/变量名

- {RULE017} 采用驼峰方式命名，禁止使用下划线命名。首字母是否大写，根据是否需要外部访问来决定

示例：



GOOD:

```
// 本package可以访问的函数
func innerFunc() bool {
    // ...
}

// 其余package可以访问的函数
func OuterFunc() error {
    // ...
}
```



BAD:

```
// 禁止用下划线分割
func inner_Func() bool {
    var srv_name string
    // ...
}

// 禁止用下划线分割
// 其余package可以访问的函数
func Outer_Func() error {
    // ...
}
```

3.6.3 常量

- {ADVICE023} 建议都使用大写字母，如果需要，可以使用下划线分割
- {ADVICE024} 尽量不要在程序中直接写数字，特殊字符串，全部用常量替代

解释

- go标准库中常量也有驼峰的命名方式，故这里不做强制限制。

示例



GOOD:

```
// 大写字母
const METHOD = "Get"
// 下划线分割
const HEADER_USER_AGENT = "User-Agent"

// go标准库中的命名方式
const defaultUserAgent = "Go 1.1 package http"
```



BAD:

```
// 全部为下划线分割的小写字母
const header_user_agent = "User-Agent"
```

3.6.4 缩写词

- {RULE018} 缩写词要保持命名的一致性。
 - 同一变量字母大小写的一致性
 - 不同变量间的一致性

示例:



```
GOOD:
var URL string

var ID int
var appID int

type ServeURL struct {
    // ...
}
```



```
BAD:
var Url string

// not consistent
var ID int
var appid int

type ServUrl struct {
    // ...
}
```

3.7 缩进

- {RULE019} 使用tab进行缩进。
- {RULE020} 跨行的缩进使用gofmt的缩进方式。
- {RULE021} 设置tabstop=4

解释

- 要求设置tabstop=4是考虑到不同编辑器跨行字符串对齐显示的一致性，比如下面的例子:

```
func main() {
    rows, err := e.db.Query(`SELECT id, name, state, create_ts, start_ts
                            FROM workflow
                            WHERE state=? AND create_ts=?
                            ORDER BY start_ts DESC`)
}
```

示例:



```
GOOD:
func longFunctionName(var_one, var_two,
    var_three, var_four) {
    // ...
}
```



```
BAD:
func longFunctionName(var_one, var_two,
    var_three, var_four) {
    // ...
}
```

示例

- {ADVICE025} 错误处理时缩进错误处理代码，对正常代码保持最少的缩进。



```
GOOD:
if err != nil {
    // error handling
    return // or continue, etc.
}
// normal code
```



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

3.8 空格

- {RULE022} 圆括号、方括号、花括号内侧都不加空格
- {RULE023} 逗号、冒号（slice中冒号除外）前不加空格，后边加一个空格
- {RULE024} 所有二元运算符前后各加一个空格（作为函数参数时除外）



```
GOOD:
var (
    s = make([]int, 10)
)

func foo() {
    m := map[string]string{"language": "golang"}
    r := 1 + 2
    func1(1+2)
    fmt.Println(m["language"])
}
```



```
BAD:
var (
    s = make( []int , 10 )
)

func foo() {
    m := map[string]string{ "language" : "golang" }
    r := 1+2
    func1(1 + 2)
    fmt.Println(m[ "language" ])
}
```

3.9 括号

- {ADVICE026} 除非用于明确算术表达式优先级，否则尽量避免冗余的括号





GOOD:

```

if x {
}

func func1() int {
    var num int
    return num
}

```



BAD:

```

if (x) {
}

func func1 int {
    var num int
    return (num)
}

```

3.10 注释

- {ADVICE027} 单行注释，采取“//”或者“/*...*/”的注释方式。
- {ADVICE028} 多行注释，采取每行开头“//”或者用“/* ... */”包括起来的注释（“/*”和“*/”作为独立的行）
- {ADVICE029} 紧跟在代码之后的注释，使用“//”

解释

- 大多数情况下，使用“//”更方便

示例



/* This is the correct format for a single-line comment */

// This is the correct format for a single-line comment

/*

 * This is the correct format for a multiline comment

 * in a section of code.

*/

// This is the correct format for a multiline comment

// in a section of code.

var a int // this is the correct format for a

 // multiline comment in a declaration

BOOL b // standard comment at the end of a line



BAD:

/* This is an incorrect format for a multiline comment

 * in a section of code.*/

var a int /* this is an incorrect comment format */

4. 编程实践

4.1 error string

- {ADVICE030} error string 尽量使用小写字母，并且结尾不带标点符号

解释

- 因为可能error string会用于其它上下文中

示例



GOOD:
`fmt.Errorf("something bad")`



BAD:
`fmt.Errorf("Something bad")`

4.2 Don't panic

- {RULE025} 除非出现不可恢复的程序错误，不要使用panic，用多返回值和error。

4.3 关于lock的保护

- {ADVICE031}
如果临界区内的逻辑较复杂、无法完全避免panic的发生，则要求适用defer来调用Unlock，即使在临界区过程中发生了panic，也会在函数退出时调用Unlock释放锁

解释

- go提供了recover，可以对panic进行捕获，但如果panic发生在临界区内，则可能导致对锁的使用没有释放
 - 这种情况下，即使panic不会导致整个程序的奔溃，也会由于“锁不释放”的问题而使临界区无法被后续的调用访问

示例



```
GOOD:
func doDemo() {
    lock.Lock()
    defer lock.Unlock()

    // 访问临界区
}
```



```
BAD:
func doDemo() {
    lock.Lock()

    // 访问临界区

    lock.Unlock()
}
```

- {ADVICE032} 上述操作如果造成临界区扩大后，需要建立单独的一个函数访问临界区。

解释

- 对于如下的代码：

```
func doDemo() {
    lock.Lock()

    // step1: 临界区内的操作

    lock.Unlock()

    // step2: 临界区外的操作
}
```

- 如果改造为defer的方式，变为如下代码，实际上扩大了临界区的范围（step2的操作也被放置在临界区了）

```
func doDemo() {
    lock.Lock()
    defer lock.Unlock()

    // step1: 临界区内的操作

    // step2: 临界区外的操作
}
```

- 需要使用单独的匿名函数，专门用于访问临界区：

```
func doDemo() {
    func() {
        lock.Lock()
        defer lock.Unlock()

        // step1: 临界区内的操作操作
    }()

    // step2: 临界区外的操作
}
```

4.4 日志的处理

- {ADVICE033} 建议使用公司golang-lib中的log库，log库地址：待补充。

4.5 unsafe package

- {ADVICE034} 除非特殊原因，不建议使用unsafe package
 - 比如进行指针和数值uintptr之间转换就是一个特殊原因