

1.1 百度 Go 编码规范 v1.3

- ✓ 本文内容较长，若您不关心细节，请直接跳到 "[7.相关工具](#)" 部分，包含了代码格式化工具和代码检查工具。

1.前言

[001] 关于本文

- 此编码风格指南早期版本基于 *BFE组Go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，避免陷入一些语言特有陷阱。
- 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。

[002] 规范级别

1. 本规范的分两个级别：
 - **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
2. 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
 - 编号由 规范级别+3位数字组成，如 **Rule001**, **Advice001**
 - 每个级别的 3 位数字都从001开始依次递增
 - 对于检查工具，当命中 **Rule** 级别规范后，标记任务失败；当命中 **Advice** 级别规范后只给出提示建议，不标记失败

[003] 适用范围

1. 不适用于头部带有 `DO NOT EDIT` 注释的生成类文件
2. 不适用于第三方依赖库
3. 不适用于主动使用 `//nolint` 注释跳过的代码段
 - a. 可以使用 `//nolint:{规则名}` 的方式 主动跳过检查，建议同时用注释给出理由
 - b. committer 和 reviewer 应清楚跳过后的影响
 - c. 对于新增/修改 nolint 规则的，自动化检查功能应给出提示建议，不将检查任务标记失败
4. 适用普通源文件和单测源文件，如 `user.go` 和 `user_test.go`

5. 适用于除去以上情形的其他所有情形

</> nolint 使用示例Go | 收起 ^

```
1 //nolint:gocyclo,Rule001 // 原因
2 func someLegacyFunction() *string {
3     // ...
4 }
```

2.命名规范

“计算机科学只存在两个难题：缓存失效和命名。”——Phil Karlton

好的命名是书写良好代码的关键。

[Rule001] 变量、常量、函数名使用驼峰法进行命名

驼峰法是Go中常用的命名方法。

特别注意的是缩写词如HTTP(Hyper Text Transfer Protocol)、ID(identifier)、URL(uniform resource locator)应该大写(遵循导出规则时需要全小写)。（参考）。

常用缩写名单：

"ACL", "API", "ASCII", "CPU", "CSS","DNS", "EOF", "GUID", "HTML", "HTTP","HTTPS", "ID", "IP",
"JSON", "QPS",

"RAM", "RPC", "SLA", "SMTP", "SQL","SSH", "TCP", "TLS", "TTL", "UDP","UI", "GID", "UID",
"UUID", "URI",

"URL", "UTF8", "VM", "XML", "XMPP","XSRF", "XSS", "SIP", "RTP", "AMQP","DB", "TS"

1	Good	Bad
	<div></> http.ServeHTTPGo 收起 ^</div> <pre>1 const FileNotExistsCode = 404 2 const DialTimeout = 1 3 4 func ServeHTTP(rw ResponseWriter, req *Request) { 5 } 6 7 type serverHandler struct {</pre>	<div></>Go 收起 ^</div> <pre>1 // 不被推荐的常量命名方式 2 const FILE_NOT_FOUND_CODE = 404 3 const Dial_Timeout = 1 4 5 // ServeHttp 应该命名为 ServeHTTP 6 func ServeHttp(rw ResponseWrite req *Request) { 7 }</pre>

```
8     srv *Server
9 }
10
```

```
8
9 type server_handler struct {
10     srv *Server
11 }
```

</> .golangci.yml 配置

Plain Text | 收起 ^

```
1 linters:
2   disable-all: true
3   enable:
4     - staticcheck      # 启用此 linter 以是该规则生效
```

[Rule002] error 类型的顶级变量请添加 `err` 或 `Err` 前缀

按照变量是否需要导出，对错误类型的变量添加 `err/Err` 前缀

1	Good	Bad
2	<pre></> net.Dial 1 var (2 ErrDial = errors.New("dial") 3 errNotFound = errors.New("not found") 4)</pre>	<pre></> 1 var (2 DialError = errors.New("dial") 3 notFound = errors.New(found") 4)</pre>

[Rule003] 禁止 receiver 使用 `self`、`this` 等其他语言惯用名

receiver 将 struct 和其对应的方法关联起来. 初学者通常会与其他语言中的 `this`、`self` 进行比较，但其他 OOP 语言中的 `this` / `self` 等通常是当前对象的指代，不仅包含了当前对象，还包含了许多额外的魔术方法。这在 Go 中并不恰当，而且还会牵扯到 Go 语言本身是否面向对象的问题。

1	Good	Bad
	<pre></> user.go 1 type User struct{ 2 id int64 3 } 4</pre>	<pre></> user.go 1 type User struct{ 2 id int64 3 } 4</pre>

<pre>5 // ID 返回用户ID 6 func (u *User) ID() int64 { 7 return u.id 8 }</pre>	<pre>5 // ID 返回用户ID 6 func (this *User) ID() int64 { 7 return this.id 8 }</pre>
3	
2	

[Rule004] receiver 的名称应该简短且保持一致

保持一致的代码风格能够增加代码的可读性与可维护性，同时也有助于提高代码的观赏性。

1	Good	Bad
	<pre></> user.go Go 收起 ^ 1 type User struct{ 2 id int64 3 name string 4 } 5 6 // ID 返回用户ID 7 func (u *User) ID() int64 { 8 return u.id 9 } 10 11 // Name 返回用户名 12 func (u *User) Name() string { 13 return u.name 14 } 15</pre>	<pre></> user.go Go 4 1 type User struct{ 2 id int64 3 name string 4 } 5 6 // ID 返回用户ID 7 func (u *User) ID() int64 { 8 return u.id 9 } 10 11 // Name 返回用户名 12 func (user *User) Name() string 13 return user.name 14 } 15 // 问题：名称不一致， 应统一使用 u 或 user</pre>

[Rule005] 包名应该简短、全小写、并且不要使用下划线

- 1. 简短、有含义地缩写并更多的站在调用方的角度进行命名，可以让调用方的代码更具美感。
 - a. 比如 `http.Serve/ net.Dial` 等都透漏着Go语言简单高效的语言关注点与理念。
- 2. 下划线、驼峰式等其他语言中惯用的包名命名方式不被 Go 语言推荐，应该避免使用。
- 3. 避免和标准库的包名重名
 - a. 一旦重名，会影响 IDE 自动导入、影响开发效率

	Good	Bad
1		
2	<ul style="list-style-type: none">timestrconvhttp	<ul style="list-style-type: none">computeServiceClientpriority_queue

[Rule006] 包内类型不应以包名为前缀

对于包内的 func/struct/var/const 等不应再添加 package name 作为前缀，但是包内可以包含与包名同名的类型。

	Good	Bad
1		
2	<pre></> net.go Go 收起 ^ 1 package net 2 3 func Dial(network, address string) (Conn, error) { 4 var d Dialer 5 return d.Dial(network, address) 6 }</pre>	<pre></> net.go Go 收起 ^ 1 package net 2 3 func NetDial(network, address string) (Conn, error) { 4 var d Dialer 5 return d.Dial(network, address) 6 }</pre>
3	通常调用方调用函数是使用包名进行调用，如以上代码使用 net.Dial("ip4:1", "192.0.2.1") 进行调用，所以在包内函数前加包名是冗余的。	调用的时候代码为 net.NetDial(xxx),可见“Net”是多余的。
4	<pre></> message/message.go Go 收起 ^ 1 package message 2 3 type Message struct{ 4 Name string 5 }</pre>	<pre></> message/ext.go Go 收起 ^ 1 package message 2 3 type MessageExt struct{ 4 Name string 5 }</pre>
5	可以有和 package 同名的类型。	应该将前缀去掉，上述 struct 名称 由 MessageExt 改为 Ext。

3.文件规范

代码是写给人看的，其次才是让机器运行。

这部分主要是对代码目录布局、文件命名等进行规范，形成统一约定，让代码更易阅读。

[Rule101] 文件名应使用小写字母，并以 `.go` 为后缀，满足规则 `[a-z0-9_]+.go`

1	Good	Not Good
2	1. http_server.go 2. context.go 3. _router.go // 以下划线开头的文件编译时会忽略掉 4. hao123.go	1. httpServer.go // 不应使用大写字母 2. context.Go // Go 只识别 .go 后缀的文件 3. 世界.go // Go 也能识别

[Rule102] 所有源文件编码必须是 `UTF-8`

Go 只能处理 `utf-8` 编码的源文件。

[Rule103] 每行代码不超过 `160` 个字符

目前显示器的像素都很高，160 字符宽度正好，超过时应该换行。

但是，在一些例如 RSA 秘钥文件中换行时，需要注意换行符可能带来的代码逻辑变更。

</> golangci-lint 规则：YAML | 收起 ^

```
1  lll:
2    # max line length, lines longer will be reported. Default is 120.
3    # '\t' is counted as 1 character by default, and can be changed with the
   tab-width option
4    line-length: 160
5    # tab width in spaces. Default to 1.
6    tab-width: 1
```

[Rule104] 测试数据文件应放在单测文件 `_test.go` 同级目录下的 `testdata` 目录里

Go 默认约定单测文件和源码文件都在同一个目录下，这样更方便维护，如源码文件是 `hello.go`，则对应单测文件是 `hello_test.go`。

`testdata` 文件夹是 Go 语言中用于存放单测数据的文件，单测代码应不跨目录访问测试数据。访问其他 `pkg` 的 `testdata`，会增加维护成本，同时也可能是由于存在设计不合理。

单测的运行一般应该与运行目录无关，即在任意目录下执行 `go test`，单测都应可正常运行。

1	Good	Bad
2	<pre></> Go 收起 ^ 1 func TestSearcher(t *testing.T) { 2 closeFn := StartMockBNS("testdata/bnsdata") 3 defer closeFn() 4 // 测试逻辑 5 }</pre>	<pre></> Go 4 1 func TestSearcher(t *testing.T) 2 closeFn := StartMockBNS("../bns/testdata") 3 defer closeFn() 4 // 测试逻辑 5 }</pre>
3	测试代码和测试数据在同一个目录下，更方便维护。	访问了另一个 pkg 里的测试数据，给后续测试数据带来不便。

[Rule105] 应使用 `go mod` 管理依赖，并将 `go.mod` 和 `go.sum` 文件提交到代码库

每个代码库应有不少于一个 `go.mod` 文件。

`go.mod` 文件里有依赖的模块和其版本信息，结合 `go.sum` 的信息进行版本校验，如此才能保证依赖的版本是锁定的。

当项目无其他依赖时，允许无 `go.sum` 文件。

[Advice101] 单个文件不超过 2000 行

在Go中除 `_test`等特殊文件名外，代码逻辑只与文件内容有关。所以当文件内容过多时，应该按照内容拆封成多个文件，便于管理与维护。

[Advice102] 同一个 struct 的方法应在同一个文件里

若一个 struct的方法分布在多个文件里，会导致可读性变差。

若想拆分，也说明这个 struct 所附带的功能特别多导致代码量特别大，这个时候应该考虑这个 struct 职责定位是否合理，是否应该对其拆分为多个独立的功能。

[Advice103] pkg 的功能描述建议写到单独的 doc.go 文件

doc.go 文件内不得包含业务逻辑代码。

</>Go | 收起 ^

```
1 // Package gmetrics 提供了对 GDP 框架默认的指标采集能力
2 // 这里是更多描述信息...
3 package gmetrics
```

4.语言规范

这部分内容是关于 Go 语言的语法、函数调用、返回值、数据类型等语言特性层面的规范。期望通过规范，来避免入坑，写出易阅读、性能高、无 Bug 的代码。

[Rule201] 文件应通过 go vet 的检查

go vet 能分析代码，帮助我们发现一些潜在的问题 Bug，其中最典型的几个问题是：

- 1. copy locks：复制了锁，会导致锁状态不对，可能导致死锁
- 2. loop closure：发现在循环中使用 go 新启动 goroutine，参数引用错误的问题
- 3. lost cancel：未调用 context 的取消函数 cancel
- 4. struct tag: 检查 struct 的 tag 是否标准
- 5. std method：检查实现和标准库里同名的方法，返回值是否也一样

在代码目录里执行命令: go vet ./...

[Rule202] 禁止在 if、for 中对 bool 类型进行等值判断

if、for 语法支持 bool 值，无需再次进行比较。同时，如果参数比较的 bool 运算过多，建议简化拆分成多个表达式

1	Good	Bad
	<div></>Go 收起 ^</div> <pre>1 func foo() { 2 var ok bool 3 for ok { 4 println("true") 5 } 6 }</pre>	<div></>Go 收起 ^</div> <pre>1 func foo() { 2 var ok bool 3 for ok == true { 4 println("true") 5 } 6 }</pre>


```
7     if !ok{
8         println("false")
9     }
10 }
```

```
7     if ok == false{
8         println("false")
9     }
10 }
```

2

[Rule203] 当函数以 else 结尾时，应删除 else 语句直接 return

相比于 if else，if return 具有更少的层级，更好的可读性。

应尽快返回错误，避免嵌套过深。

1	Good	Bad
2	<pre></> Go 收起 ^ 1 func checkID(id int) error { 2 if id > 0 { 3 return nil 4 } 5 return errors.New("i'm failed") 6 } 7</pre>	<pre></> Go 收起 ^ 1 func checkID(id int) error { 2 if id > 0 { 3 return nil 4 } else { 5 return errors.New("i'm failed") 6 } 7 }</pre>

1	Good	Bad
	<pre></> Go 收起 ^ 1 // checkValid verifies a b c is valid data. 2 func checkValid(a,b,c int) error { 3 if a < 0 { 4 return fmt.Errorf("invalid a=%d", a) 5 } 6 if b < 0 { 7 return fmt.Errorf("invalid b=%d", b) 8 } 9 if c < 0 {</pre>	<pre></> Go 收起 ^ 1 // checkValid verifies a b c is valid data. 2 func checkValid(a, b, c int) error { 3 if a >= 0 { 4 if b >= 0 { 5 if c >= 0 { 6 return nil 7 } else { 8 return 9 fmt.Errorf("invalid c= %d", c) 10 } 11 } else {</pre>

```
10         return fmt.Errorf("invalid
11         c=%d", c)
12     }
13     return nil
14 }
15
16
17
18
```

```
11         return
12         fmt.Errorf("invalid b=%d", b)
13     }
14 }
15     return fmt.Errorf("invalid
16     a=%d", a)
17 }
```

[Rule204] error 类型始终放在返回参数末尾

	Good	Bad
1	<pre></> Go 收起 ^ 1 func MissileFactory(name string) (*DF, error) { 2 switch name { 3 case "df21d": 4 return new(DF21D), nil 5 case "df17": 6 return new(DF17), nil 7 } 8 return nil, errUnknownMissileName 9 }</pre>	<pre></> Go 收起 ^ 1 func MissileFactory(name string) (error, *DF) { 2 switch name { 3 case "df21d": 4 return nil, new(DF21D) 5 case "df17": 6 return nil, new(DF17) 7 } 8 return errUnknownMissileName, nil 9 }</pre>
2		

使用 `golangci-lint` 工具时，可以使用下的配置启动该规则检查

```
</> golangci-lint 配置 YAML | 收起 ^
1  revive:
2    # see https://github.com/mgechev/revive#available-rules for details.
3    ignore-generated-header: true
4    severity: warning
5    rules:
6      - name: error-return    # 此规则
7        severity: warning
```

[Rule205] 函数返回值中的 error 的必须处理，defer 调用除外

1. 函数返回的 error 表示了处理过程是否有异常，需要作出判断，并对异常情况作出响应的处理，否则极易产生 Bug。
2. 但是像 defer file.Close() 这种调用是也是很常见的，这种不做处理，产生的危害一般较小（也是有危害的）。
3. defer 调用时的 error，可以打印到日志或者 metrics 计数，以方便排查故障原因。
4. 若要明确忽略错误，可采用将 error 赋值给 `"_"` 的方式。

</> 处理可选参数Go | 收起 ^

```
1 num, _ := strconv.Atoi(req.GetQuery("num")) // 当参数 num 不存在是读取到空字符串
2 // num 参数无关紧要，传错了也没关系的时候，就可以这样处理
3 if num > 0 {
4     // do something
5 }
```

</> 处理必填参数Go | 收起 ^

```
1 num, err := strconv.Atoi(req.GetQuery("num")) // 当参数 num 不存在是读取到空字符串
2 if err != nil {
3     // do something
4     return err
5 }
```

Good	Bad
<div><div></> 未忽略 defer errorGo 收起 ^</div><div><pre>1 func main() { 2 defer HandlerDefer(hello) 3 println("ok") 4 } 5 6 func hello() error{ 7 return errors.New("i'm failed") 8 } 9 10 // HandlerDefer 执行 Defer 函数调用 11 func HandlerDefer(fn func()error){</pre></div></div>	<div><div></> 忽略 defer errorGo 收起 ^</div><div><pre>1 func main() { 2 defer hello() 3 println("ok") 4 } 5 6 func hello() error{ 7 return errors.New("i'm failed") 8 } 9 10 11</pre></div></div>

2023/4/2 21:12

1.1 百度 Go 编码规范 v1.3

```
12     err:=fn()
13     if err==nil{
14         return
15     }
16     log.Output(2,"handler defer
failed:"+err.Error())
17 }
18
19 // go run main.go
20 // ok
21 // xxx main.go:4: handler defer
failed:i'm failed
22
23 // 当执行失败后，有日志记录下该结果
```

```
12
13 // go run main.go
14 // ok
15
16 // 执行完成后，hello 方法返回了erro
17 // 没有任何反馈，我们并不知道
18
19
20
21
22
23
24
25
```

</> golangci-lint 配置YAML | 收起 ^

```
1  errcheck:
2      # report about not checking of errors in type assertions: `a := b.
(MyStruct)`;
3      # default is false: such cases aren't reported by default.
4      check-type-assertions: false
5
6      # report about assignment of errors to blank identifier: `num, _ :=
strconv.Atoi(numStr)`;
7      # default is false: such cases aren't reported by default.
8      check-blank: false
```

[Rule206] 包装 error 时，应使用 fmt.Errorf 并配合 %w

- 1. 使用 `fmt.Errorf("%w, 补充的错误信息",err)` 包装后的 error，还可以很方便的 Unwrap 出原始的 error
 - a. 若使用 `fmt.Errorf("%s, 补充的错误信息",err.Error())` 的方式，新的 error 只能通过字符串比较的方式来判断原始 error
- 2. 若有多层函数调用，建议对下层函数返回的 error 包装后在返回
 - a. 包装的时候可以补充一些参数信息以方便排查

1	Good	Bad
	<div></>Go 收起 ^</div>	<div></>Go 收起 ^</div>

```
1 fmt.Errorf("%w, id=%d", err, 1)
```

```
1 fmt.Errorf("%s, id=%d",  
    err.Error(), 1)
```

2

以下为 error 的一些具体应用:

1.通过解析 sql.DB 对象返回的 error, 得到 SQL Server 返回的错误码

</> gdp/mysql/error.go

Go | 收起 ^

```
1 // ErrorCode 获取 mysql 的 err code  
2 // 非 mysql 专属错误将使用 ral.ErrorCode 解析  
3 func ErrorCode(err error) int32 {  
4     if err == nil {  
5         return 0  
6     }  
7     var mye *mysqlDriver.MySQLError  
8     if errors.As(err, &mye) {  
9         return int32(mye.Number)  
10    }  
11    return ral.ErrorCode(err)  
12 }
```

1

2.定义带错误编码的 error:

</> gdp/net/ral/error.go

Go | 收起 ^

```
1 import(  
2     "icode.baidu.com/baidu/gdp/extension/gerror"  
3 )  
4  
5 var (  
6     // ErrConfig 配置错误  
7     ErrConfig = gerror.NewCodeError(1000, "invalid config")  
8  
9     // ErrParam 参数错误  
10    ErrParam = gerror.NewCodeError(1001, "invalid param")  
11 )
```

[Rule207] 当入参包含 context.Context 时, 总是作为第一个参数

Context 在程序内部使用的非常的广泛，总是作为第一个参数，可以有效的减轻记忆负担，提升编程的效率。

	Good	Bad
1		
2	<pre></> Go 收起 ^ 1 func Query(ctx context.Context,id int64)(*User,error){ 2 3 }</pre>	<pre></> Go 4 1 func Query(id int64,ctx context.Context)(*User,error){ 2 3 }</pre>

[Rule208] 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

对包含锁的 struct 值进行拷贝会导致锁失效，容易产生 Bug（死锁）。go vet 可以对这种情况进行检查。

	Good	Bad
1		
2	<pre></> user.go Go 收起 ^ 1 package user 2 3 import "sync" 4 5 type User struct { 6 mux sync.Mutex 7 } 8 9 func Save(u *User) error { 10 // do something 11 return nil 12 }</pre>	<pre></> user.go Go 4 1 package user 2 3 import "sync" 4 5 type User struct { 6 mux sync.Mutex 7 } 8 9 func Save(u User) error { 10 // do something 11 return nil 12 }</pre>
3	<p>go vet user.go 不会报错</p> <p>以指针形式传递错误不会有任何问题。</p>	<p>go vet user.go 会报错：</p> <p>./user.go:9:13: Save passes lock by value: comma arguments.User contains sync.Mutex</p>

可以利用这个实现 struct 不允许值拷贝的特性。

```
</> 让 struct 不能被拷贝值 Go | 收起 ^
```

```
1 // noCopy may be embedded into structs which must not be copied
2 // after the first use.
3 //
4 // See https://golang.org/issues/8005#issuecomment-190753527
5 // for details.
6 type noCopy struct{}
7
8 // Lock is a no-op used by -copylocks checker from `go vet`.
9 func (*noCopy) Lock() {}
10 func (*noCopy) Unlock() {}
11
12 type User struct {
13     _ noCopy
14     Name string
15 }
```

[Rule209] 如果 receiver 是 map、函数或者 chan 类型，类型不可以是指针

	Good	Bad
1	<pre></> Go 收起 ^ 1 // Data 详细数据 2 type Data map[string]interface{} 3 4 // String 序列化 5 func (d Data) String() string { 6 bf, err := json.Marshal(d) 7 if err != nil { 8 return err.Error() 9 } 10 return string(bf) 11 }</pre>	<pre></> Go 收起 ^ 1 // Data 详细数据 2 type Data map[string]interface{} 3 4 // String 序列化 5 func (d *Data) String() string { 6 bf, err := json.Marshal(d) 7 if err != nil { 8 return err.Error() 9 } 10 return string(bf) 11 }</pre>
2		

[Rule210] 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

[Rule211] 禁止直接在 for 循环中使用 defer

直接在 for 循环中使用 defer 很可能使得 defer 不能执行，导致内存泄露或者其他资源问题，所以应该将 defer 放到外层。

若确实需要使用 defer，可以将逻辑封装为一个独立函数或者使用闭包。

	Good	Bad
1	<pre></> Go 收起 ^ 1 func readFiles(files []string) { 2 for i:=0;i<len(files);i++){ 3 readFile(files[i]) 4 } 5 } 6 7 func readFile(name string){ 8 f,err:=os.Open(name) 9 if err!=nil{ 10 println(err.Error()) 11 return 12 } 13 defer f.Close() 14 bf,err:=io.ReadAll(f) 15 if err!=nil{ 16 println(err.Error()) 17 }else{ 18 println(string(bf)) 19 } 20 } 21</pre>	<pre></> Go 4 1 func readFiles(files []string) { 2 for i:=0;i<len(files);i++){ 3 f,err:=os.Open(files[i]) 4 if err!=nil{ 5 println(err.Error()) 6 continue 7 } 8 9 // bug here 10 // 在循环中的 defer 只有在 结束后才会执行 11 // 若 files 很多，会导致大 件句柄未及时释放 12 defer f.Close() 13 14 bf,err:=io.ReadAll(f) 15 if err!=nil{ 16 println(err.Error()) 17 }else{ 18 println(string(bf)) 19 } 20 } 21 }</pre>

[Rule212] 函数的圈复杂度不得高于30

1. 圈复杂度(Cyclomatic complexity)是一种代码复杂度的衡量标准，详见[百科解释](#)。
2. 圈复杂度大说明程序代码可能质量低且难于测试和维护，可能是程序的分支较多，功能未合理拆分。
3. 30 是一个基本值，之所以不定义的更小，是考虑到若还是拆的更小，可能带来性能的损耗

4. 该规则可以使用 golangci-lint 的规则-gocyclo，若需要跳过规则检查，可使用 `//`

`nolint:gocyclo`

</> golangci-lint 的规则

Go | 收起 ^

```
1 gocyclo:
2   # Minimal code complexity to report.
3   # Default: 30 (but we recommend 10-20)
4   min-complexity: 30
```

[Rule213] 给 struct 赋值时，不可以省略字段名

原因：可读性差、可扩展性差（添加、调整字段顺序会导致原有代码不可编译）

[Advice201] 如果 receiver 是比较大的 struct/array，建议使用指针，这样会更有效率

元素个数 > 3 个认为是比较大

[Advice202] 如果 receiver 是比较小的 struct/array，建议使用 value 类型

[Advice203] 申明 slice 时，建议使用 var 方式申明；若能预估大小，建议预分配大小

- var 方式申明在 slice 不被 append 的情况下避免了内存分配
- 若可预估到实际大小，可以采用预分配 capacity 的方式申明，避免 slice 扩容，一次大内存开辟通常比多次 growslice 更快

1

Good

Bad

</> Go | 收起 ^

</> Go | 收起 ^

```
1 var ids []int
2
3 // 另一个例子:
4 names := make([]string, 0,
5   len(users))
5 // 预分配，避免了扩容
```

```
1 ids := []int{} // size 为 0
2
3 // 另一个例子:
4 names := make([]string, 0) // size
5   为 0
5 // 循环中需要扩容
```

```
6 for i := 0; i < len(users); i++ {
7     if len(users[i].Name) > 2 {
8         names = append(names,
9             users[i].Name)
10    }
```

```
6 for i := 0; i < len(users); i++
7     if len(users[i].Name) > 2 {
8         names = append(names,
9             users[i].Name)
10    }
```

2

[Advice204]一个 struct 定义内，最多 embedding 1 个 struct

- 1. embedding 只用于" is a " 的语义下，而不适用于 "has a" 的语义下。
- 2. 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。

</>Go | 收起 ^

```
1 // Animal 动物
2 type Animal struct {
3     // Name 名字
4     Name string
5     // ...
6 }
7
8 // Eye 眼睛
9 type Eye struct {
10    // ....
11 }
```

	Good	Bad
1		
2	<div><div></>Go 收起 ^</div><pre>1 type Dog struct { 2 Animal // 是 Animal 3 eye Eye // 有 Eye 4 }</pre></div>	<div><div></>Go 收起 ^</div><pre>1 type Dog struct { 2 Animal // 是 Animal 3 Eye // 是 Eye ? 4 }</pre></div>

[Advice205]函数参数不建议超过 5 个，大于 5 个时建议通过 struct 进行包装

[Advice206] 函数返回值小于等于 3 个，大于 3 个时建议通过struct进行包装

5.风格规范

优秀的代码是它自己最好的文档。

这部分对代码注释、代码结构等风格进行规范约束

[Rule301] 使用 tab 进行缩进，并统一格式化

- 1. 应使用格式化工具（如 Go 自带的 gofmt ）对代码格式化，并开启 “simplify code” 选项让代码保持简洁
 - a. 可以使用 gofmt -s -w file.go 完成单个文件格式化
 - b. 也可以使用其他格式化工具，详见下文的 “7.相关工具” 部分
- 2. 使用 golangci-lint 检查，没有 gosimple 规则报错

	Good	Bad
1	<pre></> for range Go 收起 ^ 1 tk := time.NewTicker(time.Second) 2 defer tk.Stop() 3 4 for range tk.C { 5 println("hello") 6 } 7 8</pre>	<pre></> Go 收 1 tk := time.NewTicker(time.Second) 2 defer tk.Stop() 3 for { 4 select { 5 case <-tk.C: 6 println("hello") 7 } 8 }</pre>
2		

[Rule302] 所有导出的类型都需要被注释, 并以该类型名为注释的开头

只要是导出类型的, 如常量、变量、函数、struct 等都需要添加注释、以说明该对象的用途。

注释的格式为: `//{一个空格}{对象名}{注释内容}`

若是函数, 还应在注释中写明入参和返回值的参数说明。

</> GoodGo | 收起 ^

```
1 // MinAge 允许报名的最小年龄
2 const MinAge = 3
3
4 // SayHello 在终端输出 "Hello World"
5 func SayHello() {
6     println("Hello World")
7 }
```

[Rule303] 全局的同一类型的常量和变量, 应定义在同一个分组里

定义在同一个分组, 同时也能表明这个类型的常量只有这一些, 可读性更好。

	Good	Bad
1	<div><div></>Go 收起 ^</div><div><pre>1 // StreamStatus 流状态 2 type StreamStatus int64 3 4 // stream 状态 5 const (6 // StreamStatusNeedShake 等待握手 7 StreamStatusNeedShake 8 StreamStatus = iota 9 // StreamStatusReady 就绪 10 StreamStatusReady 11 12 // StreamStatusOnClose 准备关闭 13 StreamStatusOnClose 14)</pre></div></div>	<div><div></>Go 收起 ^</div><div><pre>1 // StreamStatus 流状态 2 type StreamStatus int64 3 4 // StreamStatusNeedShake 等待握手 5 const StreamStatusNeedShake 6 StreamStatus =0 7 // StreamStatusReady 就绪 8 const StreamStatusReady 9 StreamStatus =1 10 // StreamStatusOnClose 准备关闭 11 const StreamStatusOnClose 12 StreamStatus =2 13</pre></div></div>

[Rule304] 包的注释应以 “*Package*” 为前缀

正确的格式是：`//{一个空格}Package{一个空格}{pkg name}{ 注释内容}`，同时上述内容和定义的 “package gmetrics” 中间是不能有空行的，若需要空行，应该是 “`//`”。

这部分的内容会被 `godoc` 识别，并展现在文档的头部。

	Good	Bad
1		
2	<pre>1 // Package gmetrics 提供了对 GDP 框架 默认指标采集能力 2 package gmetrics</pre>	<pre>1 // 期望这里是 pkg 的注释 2 package gmetrics 3</pre>

实际效果如下：

	有 包 注 释	无 包 注 释
1	<div><div>Package gmetrics</div><div>import "icode.baidu.com/baidu/gdp/metrics/gmetrics"</div><div>Overview Index Examples Subdirectories</div><div>Overview ▾ 以下为 pkg 注释内容 Package gmetrics 提供了对 GDP 框架默认的指标采集能力 按照此标准 (https://ku.baidu-int.com/d/5sBEBX8ndReIY5)实现，若想了解指标含义请看此文档。 已和云上百度智能监控很轻松的在智能监控上自动采集并对指标进行展现。 如不是使用此器</div></div>	<div><div>Overview ▾ 无 pkg 注释，这里是空的</div><div>Index ▾</div></div>
2		

[Rule305] 函数之间应有1 个空行

	Good	Bad
1		
2	<pre>1 func hello() { 2 // do something 3 } 4 5 func world() { 6 // do something</pre>	<pre>1 func hello() { 2 // do something 3 } 4 func world() { 5 // do something 6 }</pre>

7 }

7

[Rule306] 类型定义之间应有 1 个空行

Good

</>

Go | 收起 ^

```
1 type (  
2     // Connector 网络连接器  
3     Connector interface {  
4         // Pick 暴露 address picker 的 Pick 方法给 Request 使用  
5         Pick(context.Context, ...interface{}) (net.Addr, error)  
6  
7         // Connect 创建一个新的网络连接  
8         // 应确保当返回 error==nil 时, conn 不为 nil  
9         Connect(ctx context.Context, addr net.Addr) (net.Conn,  
10        error)  
11    }  
12  
13    // HasStrategy 带有负载均衡策略的 Connector 的策略名称  
14    HasStrategy interface {  
15        Strategy() string  
16    }  
17 )
```

Bad

</> 定义之间没有空行

Go | 收起 ^

```
1 type (  
2     Connector interface {  
3         // Pick 暴露 address picker 的 Pick 方法给 Request 使用  
4         Pick(context.Context, ...interface{}) (net.Addr, error)  
5  
6         // Connect 创建一个新的网络连接  
7         // 应确保当返回 error==nil 时, conn 不为 nil  
8         Connect(ctx context.Context, addr net.Addr) (net.Conn,  
9        error)  
10    }  
11  
12    // HasStrategy 带有负载均衡策略的 Connector 的策略名称  
13    HasStrategy interface {  
14        Strategy() string  
15    }  
16 )
```

[Rule307] import 需要按照标准库、第三方库、项目自身库的顺序分组排列，每组之间一个空行

1. 每组需按照升序排序
2. import 不是必须的，当有的时候，可以允许 1 组、2 组、3 组
3. 是否 标准库 pkg 的判断方法：
 - a. 通过扫描 `$GOROOT/src` 得到列表，在其中的都是
4. 是第三方库还是项目自身库的判断方法：
 - a. import 的 pkg 是否属于当前代码所属的 Module(go.mod 文件内定义的)，若是则是项目自身库
5. 建议使用 `go_fmt` (https://github.com/fsgo/go_fmt) 格式化代码。(goimports 也基本能按照上述规则排序)
 - a. 运行 `go_fmt` 可以格式化
 - b. 运行 `go_fmt -d ./...` 可以进行格式的检查

</> 格式良好的 import 内容 Go | 收起 ^

```
1 package connpool
2
3 import (
4     "context"
5     "net"
6     "strings"
7     "sync"
8
9     "icode.baidu.com/baidu/gdp/extension/messenger"
10    "icode.baidu.com/baidu/gdp/extension/observer"
11    "icode.baidu.com/baidu/gdp/extension/option"
12    "icode.baidu.com/baidu/gdp/logit"
13
14    "icode.baidu.com/baidu/gdp/net/internal/nethelper"
15 )
```

[Rule308] 禁止使用点号格式 import

会导致代码可读性极差。

</> BadGo | 收起 ^

```
1 import (  
2     "github.com/stretchr/testify/require"  
3 )  
4  
5 func TestHello(t *testing.T) {  
6     NoError(t,Hello())  
7 }  
8 // NoError 方法实际是 require 这个空间下的,  
9 // 导入当前空间后,人眼无法识别出函数的来源
```

[Rule309] 使用"_" import 的包，需要予以注释说明原因

</> GoodGo | 收起 ^

```
1 import (  
2     _ "icode.baidu.com/baidu/gdp/net/discoverer/bns" // 注册服务发现-  
3     bns  
4 )
```

[Rule310] error string不得以大写字母开头，结尾不带标点符号

1	Good	Bad
2	<div><div></>Go 收起 ^</div><div><pre>1 errors.New("something bad")</pre></div></div>	<div><div></>Go 收起 ^</div><div><pre>1 fmt.Errorf("Something bad.")</pre></div></div>

[Advice301] 函数内不同的业务逻辑处理建议采用单个空行分割

[Advice302] 尽量不要在程序中直接写数字，特殊字符串，而是用常量替代

若是一个值会在不同的包里传递和判断，还是建议用常量替代会更好。

CR 时没必要一杆子拍死，有些逻辑本身就很简单，封闭性很好的代码不用常量本身就很好阅读和理解。

[Advice303] 推荐使用单行注释 `"//"`，而不是 `"/* */"`

大多数场景，采用 `"//"` 的注释看着更整洁，但是`"/* */"`这种多行注释也可以使用，如 `cgo` 代码使用 `"/* */"`反而更清晰。

[Advice304] Copyright 应放在文件的头部

- 1. 可以没有 Copyright
- 2. 一般是第一行，也可以是其他行
 - a. 比如工具生成的代码文件首行一般会是这样：`// Code generated by protoc-gen-go. DO NOT EDIT.`
- 3. 需要在 `package` 之上

</> GoodGo | 收起 ^

1

```
1 // Copyright(C) 2022 Baidu Inc. All Rights Reserved.
2 // Date: 2022/09/07
3
4 package demo
```

[Advice305] 在函数内部不使用分组方式定义常量和变量

Good	Bad
<div><div></>Go 收起 ^</div><div><pre>1 func hello() { 2 var addr net.Addr 3 var err error 4 }</pre></div></div>	<div><div></>Go 收起 ^</div><div><pre>1 func hello() { 2 var (3 addr net.Addr 4 err error 5) 6 }</pre></div></div>

```
5 // do toomething
6 }
```

```
5 )
6 // do toomething
7 }
```

2

6.编程实践

Go：简洁但并不简单

[001] 变量、常量的分组

- 1. 同类型的、有关联的可以定义到一组，能加强代码的可阅读性，定义在一组的常量在使用 godoc 查看时，也是在同一个区域。若是不相关联的应分别定义。
- 2. 若只有一个变量也不应该使用分组定义的方式。
- 3. 在函数内部不应使用分组定义的方式。

	Good	Bad
1	<div><div></> 一个变量的情况Go 收起 ^</div><div><pre>1 2 var header *packing.Header 3</pre></div></div>	<div><div></> 一个变量的情况Go 收起 ^</div><div><pre>1 var (2 header *packing.Header 3)</pre></div></div>
2		
3	<div><div></> 多个不相干的Go 收起 ^</div><div><pre>1 // 如下为全局变量 2 3 var globalID int 4 5 var lastErr error</pre></div></div>	<div><div></> 多个不相干的Go 收起 ^</div><div><pre>1 // 如下为全局变量 2 var (3 globalID int 4 lastErr error 5)</pre></div></div>

[002] iota 的使用

- 1. iota 让连续常量值定义更方便，但是带来了值的不确定性(依赖顺序)

- 2. 若是一个常量的值会被序列化存储，或者是跨进程传递（通过 RPC 传递），那么这个常量不建议使用 `iota` 的方式来定义常量的组。风险在于若调整常量的顺序、或者删除其中一项，其值会发生变化，从而引发 Bug。
- 3. 0 值一般当做缺省值，是默认值，当 `struct` 对应字段未赋值时，程序也可以正常的处理，具备更好的向下兼容性。
 - a. 如 `Unknown`、`Unlock` 等

1	Good	Bad
	<pre></> Go 收起 ^ 1 type status int 2 3 const (4 statusFail status = iota 5 statusSuccess 6)</pre>	<pre></> Go 收起 ^ 1 // Status 查询状态 2 type Status int 3 4 const (5 // Ok 查询成功 6 OK Status = iota 7 8 // Timeout 查询超时 9 Timeout 10) 11 12 // Reply RPC 接口响应结果 13 type Reply struct { 14 Status Status 15 `json:"status"` 16 Msg string 17 `json:"msg"` 18 Data interface{} 19 `json:"data"` 20 }</pre>
2		

[003] switch 的使用

- 1. 类型断言的时候可以直接赋值，而不是到 `case` 语句里重新类型转换，如此代码更简洁

1	Good	Bad
	<pre></> Go 收起 ^ 1 switch v := val.(type) {</pre>	<pre></> Go 收起 ^ 1 switch val.(type) {</pre>

```
2 case int:
3     return v, true
4 case string:
5     num, err := strconv.Atoi(v)
6     return num, err==nil
7 }
```

```
2 case int:
3     return val.(int), true
4 case string:
5     num, err :=
6     strconv.Atoi(val.(string))
7     return num, err==nil
8 }
```

</> golangci-lint 配置

YAML | 收起 ^

```
1 gosimple:
2   # Select the Go version to target. The default is '1.13'.
3   go: "1.19"
4   # https://staticcheck.io/docs/options#checks
5   checks: [ "all" ]
```

[004] 函数参数和返回值

- 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
 - 这种函数取名建议使用前缀 Is、Has 等，如 IsPublic()、HasMoney()
- 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
 - 如 os.Open 返回的第二个参数 error 就表示是否 open 成功
- 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)
- return 时，慎用缺省方式，即函数有多个返回值时，直接 return，而不携带返回值，此方式可读性极差，易出 bug。

</> return 用缺省方式

Go | 收起 ^

```
1 func userList() (list []*User, err error) {
2     // do something
3     if someFail() {
```

```
4         return
5     }
6     // ...
7     return
8 }
9 // 返回的实际结果较难看清
10 // 更大可能返回的值没有赋值
```

[005] Don't panic

- 1. 除非出现不可恢复的程序错误，不要使用panic。
- 2. 函数执行失败可通过返回值的 error 告知调用方。
- 3. 若函数有 panic 的可能，建议以 MustXXX 的方式命名，如 标准库的 `regexp.MustCompile` 。
- 4. 若是使用 go 关键字启动了新的协程，应使用 `recover()` 函数对可能的 panic 进行捕捉。
- 5. 可以使用 GDP 框架提供的 `gtask.NoPanic` 对函数进行包装 或者 `gtask.Group` ， 详见此。
- 6. 类型转换失败是会 panic 的。
 - a. 若值可能有多种类型，建议使用 `val,ok:=obj.(MyType)`，而不是 `val:=obj.(MyType)`
- 7. 遵循防御性编码原则，当操作有多个层级的结构体时，需要对每个层级做空指针或者空数据判别，特别是在处理复杂的页面结构时。

1 示例：处理类型断言失败

2	Good	Bad
3	<pre></> Go 收起 ^ 1 if _, ok := i.(string) ; !ok { 2 // handle the error gracefully 3 }</pre>	<pre></> Go 收起 ^ 1 t := i.(string) 2</pre>

```
</> 遵循防御性编码原则，需要对每个层级做空指针或者空数据判别 Go | 收起 ^
1 type section struct {
2     item *sectionItem
3 }
4
5 type sectionItem struct {
6     path string
7 }
```

```
8
9 func actionPath(section *section) (path string, err error) {
10     if section == nil || section.Item == nil {
11         return "", "", errors.New("section or section.Item is nil")
12     }
13
14     // ...
15 }
```

[006] 关于 Lock 的使用

1. 同一个变量会在不同协程里同时 读和写 的情况的时候，需要使用锁来保护或者使用并发安全的数据类型
2. 可以使用 `sync.Map`、`atomic.Value`、`atomic.Pointer` 等并发安全的数据类型或原子操作
 - a. go1.19 新增的 `atomic.Pointer` 比 `atomic.Value` 使用更方便，性能更高
 - b. go1.19 新增的 `atomic.Bool`、`atomic.Int64` 等类型使用更方便
3. `sync.Mutex` 和 `sync.RWMutex` 不初始化也可以使用
4. 应注意锁的临界区，避免临界区扩大
 - a. 临界区扩大会导致加锁的时间变长，其他协程更长时间获取不到锁，性能变差
5. 如果临界区内的逻辑较复杂，无法完全避免 panic 的发生，则应使用 `defer` 来调用 `Unlock`，
 - a. 若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，有效避免出现死锁。
6. 锁不可导出，不可作为参数传递，并遵循“谁上锁，谁解锁”原则
7. `sync.WaitGroup`、锁(`sync.Mutex` 和 `sync.RWMutex`)，不可作为参数或返回值传递
 - a. 这相当于将自家大门的锁和钥匙交给外人，是很危险的行为
 - b. 这可能是由于函数的职责不清晰、不单一所导致

1 示例：临界区扩大

2

Good

Bad

</>

Go | 收起 ^

```
1 var nextID int
2 var lock sync.Mutex // nextID 的锁
3
4 func sendRequest() {
5     lock.Lock()
6     nextID++
7     id := nextID
8     lock.Unlock() // 完成保护后立即释
                    放锁
```

</>

Go | 收起 ^

```
1 var nextID int
2 var lock sync.Mutex // nextID 的
3
4 func sendRequest() {
5     lock.Lock()
6     // 锁的保护范围扩大到整个函数，包
    RPC 调用
7     defer lock.Unlock()
8
```

<pre>9 10 req := newRequest(id) 11 err := ral.RAL(xxx, req) // RPC 调用, 耗时1秒 12 // do something 13 }</pre>	<pre>9 nextID++ // 访问被锁保护的對象 10 req := newRequest(nextID) 11 err := ral.RAL(xxx, req) // 调用, 耗时1秒 12 // do something 13 } 14</pre>
---	--

上述例子还可以进一步优化：

</> Go | 收起 ^

```
1 var nextID int
2 var lock sync.Mutex // nextID 的锁
3
4 // pickNextID 读取下一个 ID
5 // 若是访问逻辑更复杂, 存在 panic 的风险,
6 // 应采用 defer lock.Unlock() 来解锁
7 func pickNextID() int {
8     lock.Lock()
9     nextID++
10    id := nextID
11    lock.Unlock()
12    return id
13 }
14
15 func sendRequest() {
16     id := pickNextID()
17     req := newRequest(id)
18     err := ral.RAL(xxx, req) // RPC 调用, 耗时1秒
19     // do something
20 }
```

示例：使用 defer 解锁，避免 panic 导致无法解锁的问题

Good

Bad

</> Go | 收起 ^

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

</> Go | 收起 ^

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

```

defer lock.Unlock()

// 访问临界区
}
2
3 // 若出现 panic, 锁也会释放
4 // panic 只影响了这一个请求

```

```

2
// 访问临近区
3
lock.Unlock()
4
5 // 若出现 panic, 锁不会释放
6 // 即使 recover 了, 由于死锁后续请求
  不能正常处理

```

1 Bad 示例: sync.WaitGroup 作为参数传递

</>

Go | 收起 ^

```

1 func handle(wg *sync.WaitGroup) {
2     wg.Add(1)
3     defer wg.Done()
4
5     // do something
6 }

```

[007] 日志的处理

1. 建议使用 GDP 的 logit: baidu/gdp/logit
2. 若是 ToB 项目, 可以使用 <https://github.com/rs/zerolog>

[008] 稳定性指标监控

1. 建议使用 prometheus 的监控方案, 更节省资源。
 - a. 目前 GDP 框架已经支持, 详见[此文档](#)。
2. 应对 goroutine 数添加监控报警
 - a. 请根据应用实际情况设置阈值, 一般不会高于 1000
 - b. 锁未释放、死锁、未设置超时、超时过大、资源使用完未 Close/Stop 等会导致 goroutine 数上升
 - c. pprof 能查看进程当前有哪些 goroutine


[009] unsafe package

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


[010] 单元测试

1. 单测文件和代码文件放在同一个目录下，如代码文件为 `hello.go`，则单测文件为 `hello_test.go`
2. 单测所需要的数据文件放在同目录下的 `testdata` 目录里（Rule104）
3. 不好写单测的代码不是好代码
 - a. 可能代码结构不合理、函数职责不单一、依赖不合理
4. 应该对执行的结果进行断言判断正确性，而不是使用 `print` 进行观察。
5. 推荐使用 <https://github.com/stretchr/testify> 来断言。
 - a. 如判断应该没有错误：`require.NoError(t,err)`
6. 推荐采用依赖倒置的方案来解决依赖问题，而不是使用 `monkey` 的方案。
7. 不要依赖一个外部服务
 - a. 依赖外部服务会导致单测执行场地受限、单测执行结果受到外部影响。如下游服务升级导致本应该成功的 `case` 失败或者本应该失败的 `case` 成功了。
 - b. HTTP 协议的测试可以使用 标准库的 `httptest` 包，支持 `Client` 和 `Server` 的测试
 - c. GDP 框架对测试有比较多的支持，如 `RAL`、`Redis`、`MySQL` 都是可以很方便的测试，[详见文档](#)
8. 执行单测时应添加 `-race` 参数以检查潜在的数据竞争问题
9. 对于采用 `protobuf` 等插件自动生成的代码可以不用编写单测
10. 单测覆盖率：团队可根据自身情况要求模块整体、新增代码的单测覆盖率
 - a. 建议设置为 60% 以上
 - b. 可以直接在 `icode` 上配置规则，该功能位于：设置》提交规则》开启 单元测试检查，会生成报告。
 - c. 不要为了覆盖率写无效的单测
11. 推荐使用 `test table` 的方案编写测试用例
12. 测试用例不仅要覆盖正常的情况，还需要考虑各种异常

☒ 开启 单元测试检查，会生成报告 

不影响合入

增量行覆盖率目标

80% 

在 icode 上配置规则

</> 执行单测的命令示例: Bash | 收起 ^

```
1 go test -race -timeout 30s -cover ./...
2
3 # 执行单测，并保存覆盖率文件
4 go test -race -timeout 30s -cover ./... -coverprofile=cover.out
5 # 以 HTML 格式，使用浏览器查看单测覆盖率。注意不要在远程终端执行。
6 go tool cover -html=cover.out
```

[011] 正则表达式

- 1. 正则表达式的性能比较一般，若模块要求高性能，应避免使用。
- 2. 使用前应对正则预编译，避免使用的时候临时编译。

1	Good	Bad
	<pre></> Go 收起 ^ 1 var nameReg = regexp.MustCompile(`^hello\d+\$`) 2 3 func isNameMatchV1(name string) bool { 4 return nameReg.MatchString(name) 5 } 6 // 若正则编译失败，在程序启动阶段立马能发现</pre>	<pre></> Go 收起 ^ 1 func isNameMatchV2(name string) bool { 2 nameReg := regexp.MustCompile(`^hello\d+\$`) 3 return nameReg.MatchString(name) 4 } 5 // 有 2 个问题： 6 // 1.正则编译失败会 panic 7 // 2.每次现编译性能差</pre>

</> 上述两种的性能对比（测试用例为 “hello123”） Plain Text 收起 ^			
1	cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz		
2	Benchmark_isNameMatchV1-8	11613613	108.0 ns/op
	0 allocs/op		0 B/op
3	Benchmark_isNameMatchV2-8	304123	3751 ns/op
	49 allocs/op		3809 B/op

可见两者性能相差巨大，达到 38 倍。

[012] 注意循环闭包问题

for 循环时会产生一个临时变量，该变量在每一次循环结束都会被重新赋值。启动 goroutine 是一个异步操作，当代码执行到函数内部时，变量很可能已经被重新赋值。这个bug非常常见，见以下示例。

1	Good	Bad
	<pre></> Go 收起 ^</pre>	<pre></> Go 收起 ^</pre>

2

```
1 for _, val := range []int{1, 2, 3}
2 {
3     go func(v int) {
4         fmt.Println(v)
5     }(val)
6 }
6 // 输出: 1, 2,3
```

```
1 for _, val := range []int{1, 2,
2 {
3     go func() {
4         fmt.Println(val)
5     }()
6 }
6 // 输出: 3,3,3
```

[013] nil 值的使用

interface 类型只有当其 iface 和 eface 同时为nil 时才是真的 nil。

若将一个具体的实现（如 struct）传递为 interface，虽然类型的值为 nil，但 interface 的值不为 nil。

1

Good

Bad

2

```
</> Go | 收起 ^
1 func Foo() error {
2     return nil
3 }
4
5 func main() {
6     err := Foo()
7     fmt.Println(err) //
8     <nil>
9     fmt.Println(err == nil) // true
10 }
11
12
13
14
```

```
</> Go | 4
1 func Foo() error {
2     var err *os.PathError
3     // ...
4     fmt.Println("a=", err==nil)
5     // a= true
6     return err
7 }
8 func main() {
9     err := Foo()
10    fmt.Println("b=", err)
11    // b= <nil>
12    fmt.Println("c=", err == nil)
13    // c= false
14    fmt.Println("d=", err ==
15    (*os.PathError)(nil))// d= true
16 }
17 // 注意：上述第 4、11 行一样的判断逻辑
18 // 但是结果不一样
```

[014] 安全

数据安全：

1. 请注意打印的日志、接口输出的数据不要触犯《[百度安全红线](#)》
 - a. 比如 BDUSS 不可以输出到页面，不可以打印到日志
 - b. 更多安全细则请从“[百度安全统一服务平台](#)”获取
2. 应用的调试接口、metrics Exporter 接口、pprof 接口 等不可在外网能访问到
 - a. 这类接口需添加权限校验限制
 - b. 建议直接在 BFE 上配置对应的拒绝访问规则
3. 查询数据库时建议使用 SQLBuilder 或者 ORM，避免注入漏洞
4. 建议数据库(包括 MySQL、Redis 等)应配置 IP 白名单和密码
5. 建议将密码/秘钥使用 [SSM 系统](#)(或类似系统)托管而不是提交到代码库
 - a. GDP 框架有支持 SSM 系统，[详见此文档](#)
6. 代码执行 shell 注意操作风险

Bad

</> 风险-执行任意命令:

Go | 收起 ^

```
1 err := exec.CommandContext(req.Query("cmd")).Run()
```

</> 函数内的资源对象使用defer回收

Go | 收起 ^

```
1 func do() {
2     conn := pool.Get()
3     defer pool.Put(conn)
4
5     // ...
6
7     return
8 }
```

应用安全/稳定:

1. 建议升级到 go1.19，并使用 `runtime.SetMemoryLimit` 限制应用内存，以降低 OOM 的风险

[015] 字符串

1. 不区分大小写的比较可以使用 `strings.EqualFold`、`bytes.EqualFold`。
2. bytes 的对比可以使用 `bytes.Equal` 方法。
3. 字符串拼接应避免使用 `fmt.Sprintf` 等 `fmt` 的方法，性能较差，请使用 `bytes.Buffer`、`strings.Builder`、`strings.Join` 等方式，对于简单的，还可以直接使用 + 连接 2 个字符串。

	Good	Bad
1	<div><div></> 字符串不区分大小写对比Go 收起 ^</div><div><pre>1 var x, y string 2 3 if strings.EqualFold(x, y)</pre></div></div>	<div><div></>Go 收起 ^</div><div><pre>1 var x, y string 2 3 if strings.ToLower(x) == strings.ToLower(y)</pre></div></div>
3	<div><div></> []bytes 对比Go 收起 ^</div><div><pre>1 if bytes.Equal(x, y) {}</pre></div></div>	<div><div></>Go 收起 ^</div><div><pre>1 if bytes.Compare(x, y) == 0 {}</pre></div></div>

[016] JSON 编解码

- PHP 输出的 JSON 数据很容易出现格式问题，导致解析失败
 - 空 array 输出是数组格式，如 `[]`，array有值的时候输出的是对象格式，如 `{"hello":"world"}`（建议由 PHP 程序修复成本最低）
- 遇到不确定结构和类型的字段（如同一个字段一会是 数字 1，一会是字符串 “1”；一会是数组、一会是对象）
 - 可以定义为 `json.RawMessage` 类型而不是 `interface{}` 类型，这样可以根据业务场景，做二次 unmarshal 而且性能比 `interface` 快很多。
 - 可以实现 `json.Marshaler` 和 `json.Unmarshaler` 类型以实现自定义解析、编码过程
 - 内置的 `json.Number` 类型支持解析字符串和非字符串格式的数字，如 1, "1",123.1
- 可以使用 `baidu/gdp/exjson` 中已定义的一些自定义类型
 - ExString：支持将 “abcde”，12345, true, false, null 解析为字符串
 - ExInt64：支持将 12345, “12345”, false(0), null(0), “12345.1”, “12345.0”, 12345.1 解析为int64
 - ExBool: 支持将 “true”, 1, true, false, null 解析为 bool
 - ExDuration：支持将 "1day","300ms" 解析为 time.Duration
- 需要输出带引号的数字时，可以在`tag`中添加 string 标记实现

</>Go | 收起 ^

```
1 type User struct {
2     ID int64 `json:"id,string"`
3 }
4
5 func main() {
6     bf, _ := json.Marshal(User{ID: 999})
7     println(string(bf))
8 }
```

```
8 }
9 // {"id":"999"}
```

[017] 注意隐匿的map、slice 的并发读写问题

大家都知道在go中 `map`、`slice` 是不可以并发读写的, 但通常在业务使用中存在多次隐匿传递容易导致大家忽略该问题。由于 `map` 并发读写是 `fatal` 错误, 无法被 `recover`, 容易引发更大的风险, 需要重点防范。

1

Bad

</>

Go | 收起 ^

```
1 var globalConfig = map[string]string{"cache": "read"}
2
3 func (s *sev) ServeHTTP(w http.ResponseWriter, r *http.Request) {
4     confCopied := globalConfig // 变量copy等操作
5
6     if _, ok := confCopied["cache"]; ok {
7         // 此处会发生fatal错误导致服务退出, 并且无法被 recover
8         // fatal error: concurrent map read and map write
9         confCopied["temp"] = "something"
10    }
11 }
```

2

7.相关工具

工欲善其事，必先利其器。

所有相关工具都要求使用 Go 1.19 版本编译

- Go 1.19在格式化代码时也会格式化注释(详见 [Go 1.19 Release Notes](#)), 使用相同版本可以避免自动化检查工具和提交的代码出现 Diff

[001] 代码格式化

1	名称	介绍
	gorgeous (推	功能说明:

荐)

- 1. 格式化 import
- 2. 格式化注释
- 3. 代码格式化、简化
- 4. 自动重写低效代码

能自动识别哪些文件有修改并只对这部分文件格式化。

项目地址：

https://github.com/fsgo/go_fmt

安装：

go install github.com/fsgo/go_fmt/cmd/gorgeous@latest

使用：

gorgeous # 格式化本次修改过的代码(通过 git status 判断)
gorgeous ./... # 格式化当前以及子目录下所有代码
gorgeous -d ./... # 检查哪些代码未格式化

icoding VS-Code 默认已经集成。若本地 VS-Code 需要配置，可按照如下步骤：

- ①安装命令：go install github.com/fsgo/go_fmt/cmd/goformat@latest
- ②修改配置的 Go: Format Tool ， 设置为 "goformat"

gofumpt

功能说明：

一个更严格的 gofmt

项目地址：

<https://github.com/mvdan/gofumpt>

安装：

go install mvdan.cc/gofumpt@latest

使用：

gofumpt -l -w .

gofmt

功能说明：

- 1. 代码格式化、简化

Go SDK 内置

使用：

5		<code>gofmt -w -s main.go # 格式化(含简化)并保存指定的文件</code>
	goimports	<div>功能说明：<ol style="list-style-type: none">格式化 import代码格式化、简化格式化检查</div> <div>项目地址：https://github.com/golang/tools</div> <div>安装： <code>go install golang.org/x/tools/cmd/goimports@latest</code></div> <div>使用： <code>goimports -w main.go</code></div>

[002] 代码检查-开源

1	名称	介绍
2	gorgeous	同上
3	vet (推荐)	<div>功能说明： Go SDK 内置的静态代码分析检查工具。能帮助我们发现一些潜在的问题 Bug，其中最典型的几个问题：<ol style="list-style-type: none">copy locks：复制了锁，会导致锁状态不对，可能导致死锁loop closure：发现在循环中使用 go 新启动 goroutine，参数引用错误的问题lost cancel：未调用 context 的取消函数 cancelstruct tag：检查 struct 的 tag 是否标准std method：检查实现和标准库里同名的方法，返回值是否也一样</div> <div>使用： <code>go vet ./...</code></div>
	golangci-lint (推荐)	<div>功能说明： <code>golangci-lint</code> is a fast Go linters runner. It runs linters in parallel, uses caching, supports <code>yaml</code> config, has integrations with all major IDE and has dozens of linters included.</div> <div>项目地址：</div>
4		

<https://github.com/golangci/golangci-lint/>

安装:

go install github.com/golangci/golangci-lint/cmd/golangci-lint@master

使用:

默认配置文件在 ~/.golangci.yml 以及当前目录的 .golangci.yml
golangci-lint run ./...

revive

功能说明:

Fast, configurable, extensible, flexible, and beautiful linter for Go. Drop-in replacement of golint. **Revive provides a framework for development of custom rules, and lets you define a strict preset for enhancing your development & code review processes.**

项目地址:

<https://github.com/mgechev/revive>

安装

go install github.com/mgechev/revive@master

使用:

revive ./...

staticcheck (推荐)

功能说明

Go static analysis, detecting bugs, performance issues, and much more.

项目地址:

<https://github.com/dominikh/go-tools>

安装:

go install honnef.co/go/tools/cmd/staticcheck@latest

使用:

staticcheck ./...

</> 推荐的 golangci-lint 规则(放在 ~/.golangci.yml):

YAML | 收起 ^

1 output:

```
2  #format: json
3  print-issued-lines: true
4  linters:
5    # enable-all: true
6    # disable:
7    #   - deadcode
8    disable-all: true
9    enable:
10     - stylecheck
11     - revive
12     - gosimple
13     - gofmt
14     - lll
15     - errcheck
16     - errorlint
17     - govet
18     - gocyclo
19     - goimports
20  linters-settings:
21    lll:
22      # max line length, lines longer will be reported. Default is 120.
23      # '\t' is counted as 1 character by default, and can be changed with the
24      # tab-width option
25      line-length: 160
26      # tab width in spaces. Default to 1.
27      tab-width: 1
28    errcheck:
29      # report about not checking of errors in type assertions: `a := b.
30      # (MyStruct)`;
31      # default is false: such cases aren't reported by default.
32      check-type-assertions: false
33      # report about assignment of errors to blank identifier: `num, _ :=
34      # strconv.Atoi(numStr)`;
35      # default is false: such cases aren't reported by default.
36      check-blank: false
37    gocyclo:
38      # Minimal code complexity to report.
39      # Default: 30 (but we recommend 10-20)
40      min-complexity: 30
```

[003] 代码检查-百度内

基于本规范定制开发的，和 icode、ipipe 深度整合的自动化检查工具。

所有的 Go 项目已默认启用，违反规范将不能提交, [详见此：icode Go 编码规范检查](#)。

注意，该工具目前基于 Go1.19，若是低版本 go 格式化后的代码，检查也提示未格式化（主要是注释部分），请使用 Go1.19 版本的 gofmt 格式化代码（其他格式化工具则需要由 Go1.19编译）

8. 代码共享/复用

鼓励复用已有的代码、鼓励内部模块开源。

8.1 优秀内部开源模块

8.2 优秀外部开源模块