go小技巧

Go 箴言

- 不要通过共享内存进行通信，通过通信共享内存
- 并发不是并行
- 管道用于协调；互斥量（锁）用于同步
- 接口越大，抽象就越弱
- 利用好零值
- 空接口 `interface{}` 没有任何类型约束
- Gofmt 的风格不是人们最喜欢的，但 gofmt 是每个人的最爱
- 允许一点点重复比引入一点点依赖更好
- 系统调用必须始终使用构建标记进行保护
- 必须始终使用构建标记保护 Cgo
- Cgo 不是 Go
- 使用标准库的 `unsafe` 包，不能保证能如期运行
- 清晰比聪明更好
- 反射永远不清晰
- 错误是值
- 不要只检查错误，还要优雅地处理它们
- 设计架构，命名组件，（文档）记录细节
- 文档是供用户使用的
- 不要（在生产环境）使用 `panic()`

Author: Rob Pike
See more: https://go-proverbs.github.io/ (https://go-proverbs.github.io/)

Go 之禅

- 每个 package 实现单一的目的
- 显式处理错误
- 尽早返回，而不是使用深嵌套
- 让调用者处理并发（带来的问题）
- 在启动一个 goroutine 时，需要知道何时它会停止
- 避免 package 级别的状态
- 简单很重要
- 编写测试以锁定 package API 的行为
- 如果你觉得慢，先编写 benchmark 来证明
- 适度是一种美德
- 可维护性

Author: Dave Cheney
See more: https://the-zen-of-go.netlify.com/ (https://the-zen-of-go.netlify.com/)

代码

使用 `go fmt` 格式化

让团队一起使用官方的 Go 格式工具，不要重新发明轮子。
尝试减少代码复杂度。 这将帮助所有人使代码易于阅读。

多个 if 语句可以折叠成 switch

```
// NOT BAD
if foo() {
    // ...
} else if bar == baz {
    // ...
} else {
    // ...
}

// BETTER
switch {
case foo():
    // ...
case bar == baz:
    // ...
default:
    // ...
}
```

用 `chan struct{}` 来传递信号，`chan bool` 表达的不够清楚

当你在结构中看到 `chan bool` 的定义时，有时不容易理解如何使用该值，例如：

```
type Service struct {
    deleteCh chan bool // what does this bool mean?
}
```

但是我们可以将其改为明确的 `chan struct {}` 来使其更清楚：我们不在乎值（它始终是 `struct {}` ），我们关心可能发生的事件，例如：

```
type Service struct {
    deleteCh chan struct{} // ok, if event than delete something.
}
```

`30 * time.Second` 比 `time.Duration(30) * time.Second` 更好

你不需要将无类型的常量包装成类型，编译器会找出来。
另外最好将常量移到第一位：

```
// BAD
delay := time.Second * 60 * 24 * 60

// VERY BAD
delay := 60 * time.Second * 60 * 24

// GOOD
delay := 24 * 60 * 60 * time.Second
```

用 `time.Duration` 代替 `int64` + 变量名

```
// BAD
var delayMillis int64 = 15000

// GOOD
var delay time.Duration = 15 * time.Second
```

按类型分组 `const` 声明，按逻辑和/或类型分组 `var`

```
// BAD
const (
    foo = 1
    bar = 2
    message = "warn message"
)

// MOSTLY BAD
const foo = 1
const bar = 2
const message = "warn message"

// GOOD
const (
    foo = 1
    bar = 2
)

const message = "warn message"
```

这个模式也适用于 `var` 。

- ☐ 每个阻塞或者 IO 函数操作应该是可取消的或者至少是可超时的
- ☐ 为整型常量值实现 `Stringer` 接口
  - https://godoc.org/golang.org/x/tools/cmd/stringer (https://godoc.org/golang.org/x/tools/cmd/stringer)
- ☐ 检查 `defer` 中的错误

```
defer func() {
    err := ocp.Close()
    if err != nil {
        rerr = err
    }
}()
```

- ☐ 不要在 `checkErr` 函数中使用 `panic()` 或 `os.Exit()`
- ☐ 仅仅在很特殊情况下才使用 panic, 你必须要去处理 error
- ☐ 不要给枚举使用别名，因为这打破了类型安全
  - https://play.golang.org/p/MGbeDwtXN3 (https://play.golang.org/p/MGbeDwtXN3)

```
package main
type Status = int
type Format = int // remove `=` to have type safety

const A Status = 1
const B Format = 1

func main() {
    println(A == B)
}
```

- ☐

  如果你想省略返回参数，你最好表示出来

  - `_ = f()` 比 `f()` 更好
- ☐

  我们用 `a := []T{}` 来简单初始化 slice

- ☐

  用 range 循环来进行数组或 slice 的迭代

  - `for _, c := range a[3:7] {...}` 比 `for i := 3; i < 7; i++ {...}` 更好
- ☐

  多行字符串用反引号(`)

- ☐

  用 `_` 来跳过不用的参数

```
func f(a int, _ string) {}
```

- ☐ 如果你要比较时间戳，请使用 `time.Before` 或 `time.After`，不要使用 `time.Sub` 来获得 duration (持续时间)，然后检查它的值。
- ☐ 带有上下文的函数第一个参数名为 `ctx`，形如：`func foo(ctx Context, ...)`
- ☐ 几个相同类型的参数定义可以用简短的方式来进行

```
func f(a int, b int, s string, p string)


func f(a, b int, s, p string)
```

- ☐ 一个 slice 的零值是 nil
  - https://play.golang.org/p/pNT0d_Bunq (https://play.golang.org/p/pNT0d_Bunq)

```
var s []int
fmt.Println(s, len(s), cap(s))
if s == nil {
fmt.Println("nil!")
}
// Output:
// [] 0 0
// nil!
```

    - https://play.golang.org/p/meTInNyxtk (https://play.golang.org/p/meTInNyxtk)

```
var a []string
b := []string{}

fmt.Println(reflect.DeepEqual(a, []string{}))
fmt.Println(reflect.DeepEqual(b, []string{}))
// Output:
// false
// true
```

- ☐ 不要将枚举类型与 `<`, `>`, `<=` 和 `>=` 进行比较
  - 使用确定的值，不要像下面这样做:

```
value := reflect.ValueOf(object)
kind := value.Kind()
if kind >= reflect.Chan && kind <= reflect.Slice {
  // ...
}
```

- ☐ 用 `%+v` 来打印数据的比较全的信息
- ☐ 注意空结构 `struct{}`, 看 issue: https://github.com/golang/go/issues/23440 (https://github.com/golang/go/issues/23440)
  - more: https://play.golang.org/p/9C0puRUstrP (https://play.golang.org/p/9C0puRUstrP)

```
func f1() {
  var a, b struct{}
  print(&a, "\n", &b, "\n") // Prints same address
  fmt.Println(&a == &b)     // Comparison returns false
}

func f2() {
  var a, b struct{}
  fmt.Printf("%p\n%p\n", &a, &b) // Again, same address
  fmt.Println(&a == &b)          // ...but the comparison returns true
}
```

- ▢

  包装错误： http://github.com/pkg/errors (http://github.com/pkg/errors)

  - 例如: `errors.Wrap(err, "additional message to a given error")`
- ▢

  在 Go 里面要小心使用 `range` :

  - `for i := range a` and `for i, v := range &a` ，都不是 `a` 的副本
  - 但是 `for i, v := range a` 里面的就是 `a` 的副本
  - 更多: https://play.golang.org/p/4b181zkB1O (https://play.golang.org/p/4b181zkB1O)
- ▢

  从 map 读取一个不存在的 key 将不会 panic

  - `value := map["no_key"]` 将得到一个 0 值
  - `value, ok := map["no_key"]` 更好
- ▢

  不要使用原始参数进行文件操作

  - 而不是一个八进制参数 `os.MkdirAll(root, 0700)`
  - 使用此类型的预定义常量 `os.FileMode`
- ▢

  不要忘记为 `iota` 指定一种类型

  - https://play.golang.org/p/mZZdMaI92cI (https://play.golang.org/p/mZZdMaI92cI)

```
const (
  _ = iota
  testvar         // testvar 将是 int 类型
)
```

vs

```
type myType int
const (
  _ myType = iota
  testvar         // testvar 将是 myType 类型
)
```

不要在你不拥有的结构上使用 `encoding/gob`

在某些时候，结构可能会改变，而你可能会错过这一点。因此，这可能会导致很难找到 bug。

不要依赖于计算顺序，特别是在 return 语句中。

```
// BAD
return res, json.Unmarshal(b, &res)

// GOOD
err := json.Unmarshal(b, &res)
return res, err
```

防止结构体字段用纯值方式初始化，添加 `_ struct {}` 字段：

```
type Point struct {
  X, Y float64
  _    struct{} // to prevent unkeyed literals
}
```

对于 `Point {X：1，Y：1}` 都可以，但是对于 `Point {1,1}` 则会出现编译错误：

```
./file.go:1:11: too few values in Point literal
```

当在你所有的结构体中添加了 `_ struct{}` 后，使用 `go vet` 命令进行检查，（原来声明的方式）就会提示没有足够的参数。

为了防止结构比较，添加 `func` 类型的空字段

```
type Point struct {
  _ [0]func() // unexported, zero-width non-comparable field
  X, Y float64
}
```

`http.HandlerFunc` 比 `http.Handler` 更好

用 `http.HandlerFunc` 你仅需要一个 func， `http.Handler` 需要一个类型。

移动 `defer` 到顶部

这可以提高代码可读性并明确函数结束时调用了什么。

JavaScript 解析整数为浮点数并且你的 int64 可能溢出

用 `json:"id,string"` 代替

```
type Request struct {
  ID int64 `json:"id,string"`
}
```

并发

- ☐ 以线程安全的方式创建单例（只创建一次）的最好选择是 `sync.Once`
  - 不要用 flags, mutexes, channels or atomics
- ☐ 永远不要使用 `select{}`, 省略通道， 等待信号
- ☐ 不要关闭一个发送（写入）管道，应该由创建者关闭
  - 往一个关闭的 channel 写数据会引起 panic
- ☐ `math/rand` 中的 `func NewSource(seed int64) Source` 不是并发安全的，默认的 `lockedSource` 是并发安全的, see issue: https://github.com/golang/go/issues/3611 (https://github.com/golang/go/issues/3611)
  - 更多: https://golang.org/pkg/math/rand/ (https://golang.org/pkg/math/rand/)
- ☐ 当你需要一个自定义类型的 atomic 值时，可以使用 atomic.Value (https://godoc.org/sync/atomic#Value)

性能

- ☐ 不要省略 `defer`
  - 在大多数情况下 200ns 加速可以忽略不计
- ☐ 总是关闭 http body `defer r.Body.Close()`
  - 除非你需要泄露 goroutine
- ☐ 过滤但不分配新内存

```
b := a[:0]
for _, x := range a {
    if f(x) {
      b = append(b, x)
    }
}
```

为了帮助编译器删除绑定检查，请参见此模式 `_ = b [7]`

- ☐ `time.Time` 有指针字段 `time.Location` 并且这对 go GC 不好
  - 只有使用了大量的 `time.Time` 才（对性能）有意义，否则用 timestamp 代替
- ☐ `regexp.MustCompile` 比 `regexp.Compile` 更好
  - 在大多数情况下，你的正则表达式是不可变的，所以你最好在 `func init` 中初始化它
- ☐ 请勿在你的热点代码中过度使用 `fmt.Sprintf`. 由于维护接口的缓冲池和动态调度，它是很昂贵的。
  - 如果你正在使用 `fmt.Sprintf("%s%s", var1, var2)`, 考虑使用简单的字符串连接。
  - 如果你正在使用 `fmt.Sprintf("%x", var)`, 考虑使用 `hex.EncodeToString` or `strconv.FormatInt(var, 16)`
- ☐ 如果你不需要用它，可以考虑丢弃它，例如 `io.Copy(ioutil.Discard, resp.Body)`
  - HTTP 客户端的传输不会重用连接，直到body被读完和关闭。

```
res, _ := client.Do(req)
io.Copy(ioutil.Discard, res.Body)
defer res.Body.Close()
```

- ☐ 不要在循环中使用 defer，否则会导致内存泄露
  - 因为这些 defer 会不断地填满你的栈（内存）
- ☐ 不要忘记停止 ticker, 除非你需要泄露 channel

```
ticker := time.NewTicker(1 * time.Second)
defer ticker.Stop()
```

- ☐ 用自定义的 marshaler 去加速 marshaler 过程
  - 但是在使用它之前要进行定制！例如：https://play.golang.org/p/SEm9Hvsi0r (https://play.golang.org/p/SEm9Hvsi0r)

```
func (entry Entry) MarshalJSON() ([]byte, error) {
  buffer := bytes.NewBufferString("{")
  first := true
  for key, value := range entry {
      jsonValue, err := json.Marshal(value)
      if err != nil {
          return nil, err
      }
      if !first {
          buffer.WriteString(",")
      }
      first = false
      buffer.WriteString(key + ":" + string(jsonValue))
  }
  buffer.WriteString("}")
  return buffer.Bytes(), nil
}
```

- ☐

  `sync.Map` 不是万能的，没有很强的理由就不要使用它。

  - 了解更多: https://github.com/golang/go/blob/master/src/sync/map.go#L12 (https://github.com/golang/go/blob/master/src/sync/map.go#L12)
- ☐

  在 `sync.Pool` 中分配内存存储非指针数据

  - 了解更多: https://github.com/dominikh/go-tools/blob/master/cmd/staticcheck/docs/checks/SA6002 (https://github.com/dominikh/go-tools/blob/master/cmd/staticcheck/docs/checks/SA6002)
- ☐

  为了隐藏逃生分析的指针，你可以小心使用这个函数：：

  - 来源: https://go-review.googlesource.com/c/go/+/86976 (https://go-review.googlesource.com/c/go/+/86976)

```
// noescape hides a pointer from escape analysis.  noescape is
// the identity function but escape analysis doesn't think the
// output depends on the input. noescape is inlined and currently
// compiles down to zero instructions.
//go:nosplit
func noescape(p unsafe.Pointer) unsafe.Pointer {
  x := uintptr(p)
  return unsafe.Pointer(x ^ 0)
}
```

- ▢

  对于最快的原子交换，你可以使用这个 `m := (*map[int]int)(atomic.LoadPointer(&ptr))`

- ▢

  如果执行许多顺序读取或写入操作，请使用缓冲 I/O

  - 减少系统调用次数
- ▢

  有 2 种方法清空一个 map：

  - 重用 map 内存 （但是也要注意 m 的回收）

```
for k := range m {
  delete(m, k)
}
```

- 分配新的

```
m = make(map[int]int)
```

模块

- ▢ 如果你想在 CI 中测试 `go.mod` （和 `go.sum` ）是否是最新 https://blog.urth.org/2019/08/13/testing-go-mod-tidiness-in-ci/ (https://blog.urth.org/2019/08/13/testing-go-mod-tidiness-in-ci/)

构建

- ▢ 用这个命令 `go build -ldflags="-s -w" ...` 去掉你的二进制文件
- ▢ 拆分构建不同版本的简单方法
  - 用 `// +build integration` 并且运行他们 `go test -v --tags integration .`
- ▢ 最小的 Go Docker 镜像
  - https://twitter.com/bbrodriges/status/873414658178396160 (https://twitter.com/bbrodriges/status/873414658178396160)
  - `CGO_ENABLED=0 go build -ldflags="-s -w" app.go && tar C app | docker import - myimage:latest`

- ☐ run go format on CI and compare diff
  - 这将确保一切都是生成的和承诺的
- ☐ 用最新的 Go 运行 Travis-CI，用 `travis 1`
  - 了解更多：https://github.com/travis-ci/travis-build/blob/master/public/version-aliases/go.json (https://github.com/travis-ci/travis-build/blob/master/public/version-aliases/go.json)
- ☐ 检查代码格式是否有错误 `diff -u <(echo -n) <(gofmt -d .)`

测试

- ☐ 测试名称 `package_test` 比 `package` 要好
- ☐ `go test -short` 允许减少要运行的测试数

```
func TestSomething(t *testing.T) {
  if testing.Short() {
    t.Skip("skipping test in short mode.")
  }
}
```

- ☐ 根据系统架构跳过测试

```
if runtime.GOARM == "arm" {
  t.Skip("this doesn't work under ARM")
}
```

- ☐ 用 `testing.AllocsPerRun` 跟踪你的内存分配
  - https://godoc.org/testing#AllocsPerRun (https://godoc.org/testing#AllocsPerRun)
- ☐ 多次运行你的基准测试可以避免噪音。
  - `go test -test.bench=. -count=20`

工具

- ☐

  快速替换 `gofmt -w -l -r "panic(err) -> log.Error(err)" .`

- ☐

  `go list` 允许找到所有直接和传递的依赖关系

  - `go list -f '{{ .Imports }}' package`
  - `go list -f '{{ .Deps }}' package`
- ☐

  对于快速基准比较，我们有一个 `benchstat` 工具。

  - https://godoc.org/golang.org/x/perf/cmd/benchstat (https://godoc.org/golang.org/x/perf/cmd/benchstat)
- ☐

go-critic (https://github.com/go-critic/go-critic) linter 从这个文件中强制执行几条建议

- ☐

  `go mod why -m <module>` 告诉我们为什么特定的模块在 `go.mod` 文件中。

- ☐

  `GOGC=off go build ...` 应该会加快构建速度 source (https://twitter.com/mvdan_/status/1107579946501853191)

- ☐

  内存分析器每 512KB 记录一次分配。你能通过 `GODEBUG` 环境变量增加比例，来查看你的文件的更多详细信息。

  - 来源：https://twitter.com/bboreham/status/1105036740253937664 (https://twitter.com/bboreham/status/1105036740253937664)

- ☐

  `go mod why -m <module>` 告诉我们为什么特定的模块是在 `go.mod` 文件中。

其他

- ☐ dump goroutines https://stackoverflow.com/a/27398062/433041 (https://stackoverflow.com/a/27398062/433041)

```
go func() {
  sigs := make(chan os.Signal, 1)
  signal.Notify(sigs, syscall.SIGQUIT)
  buf := make([]byte, 1<<20)
  for {
    <-sigs
    stacklen := runtime.Stack(buf, true)
    log.Printf("=== received SIGQUIT ===\n*** goroutine dump...\n%s\n*** end\n"  ,
buf[:stacklen])
  }
 }()
```

- ☐ 在编译期检查接口的实现

```
var _ io.Reader = (*MyFastReader)(nil)
```

- ☐ len(nil) = 0
  - https://golang.org/pkg/builtin/#len (https://golang.org/pkg/builtin/#len)
- ☐ 匿名结构很酷

```
var hits struct {
  sync.Mutex
  n int
}
hits.Lock()
hits.n++
hits.Unlock()
```

- [ ]

  `httputil.DumpRequest` 是非常有用的东西，不要自己创建

  - https://godoc.org/net/http/httputil#DumpRequest (https://godoc.org/net/http/httputil#DumpRequest)
- [ ]

  获得调用堆栈，我们可以使用 `runtime.Caller`

  - https://golang.org/pkg/runtime/#Caller (https://golang.org/pkg/runtime/#Caller)
- [ ]

  要 marshal 任意的 JSON， 你可以 marshal 为 `map[string]interface{}{}`

- [ ]

  配置你的 `CDPATH` 以便你能在任何目录执行 `cd github.com/golang/go`

  - 添加这一行代码到 `bashrc` (或者其他类似的) `export CDPATH=$CDPATH:$GOPATH/src`
- [ ]

  从一个 slice 生成简单的随机元素

  - `[]string{"one", "two", "three"}[rand.Intn(3)]`

转自：https://github.com/cristaloleg/go-advice/blob/master/README_ZH.md (https://github.com/cristaloleg/go-advice/blob/master/README_ZH.md)

最后编辑: kuteng 文档更新时间: 2021-02-22 19:43 作者：kuteng