Go语言(Golang)编码规范

书栈(BookStack.CN)

目　录

致谢

当前文档 《Go语言(Golang)编码规范》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-07-24。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/go-code-convention

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

编码规范

编码规范

本手册是由 无闻 根据自身项目实战经验讨论总结出来的 Go 语言编码规范，在一定程度上与 Go 官方的标准库 Code Review 规范有所不同。所述内容均为参考意见，并不一定属于主流方案或绝对正确。

版权声明

版权声明

作为开源项目，必须有相应的开源许可证才能算是真正的开源。在选择了一个开源许可证之后，需要在源文件中进行相应的版权声明才能生效。以下分别以 Apache License, Version 2.0 和 MIT 授权许可为例。

Apache License, Version 2.0

该许可证要求在所有的源文件中的头部放置以下内容才能算协议对该文件有效：

```
// Copyright [yyyy] [name of copyright owner]
//
// Licensed under the Apache License, Version 2.0 (the "License"): you may
// not use this file except in compliance with the License. You may obtain
// a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// 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.
```

其中，`[yyyy]` 表示该源文件创建的年份。紧随其后的是 `[name of copyright owner]`，即版权所有者。如果为个人项目，就写个人名称；若为团队项目，则宜写团队名称。

MIT License

一般使用 MIT 授权的项目，需在源文件头部增加以下内容：

```
// Copyright [yyyy] [name of copyright owner]. All rights reserved.
// Use of this source code is governed by a MIT-style
// license that can be found in the LICENSE file.
```

其中，年份和版权所有者的名称填写规则与 Apache License, Version 2.0 的一样。

其它说明

- 其它类型的开源许可证基本上都可参照以上两种方案。
- 如果存在不同作者或组织对同个源文件的修改，在协议兼容的情况下，可将首行变为多行，按照先后次序排放：

```
// Copyright 2011 Gary Burd
// Copyright 2013 Unknwon
```

- 在 README 文件最后中需要说明项目所采用的开源许可证：

```
## 授权许可

本项目采用 MIT 开源授权许可证，完整的授权说明已放置在 [LICENSE](LICENSE) 文件中。
```

项目结构

项目结构

以下为一般项目结构，根据不同的 Web 框架习惯，可使用括号内的文字替换；根据不同的项目类型和需求，可自由增删某些结构：

```
- templates (views)          # 模板文件
- public (static)            # 静态文件
    - css
    - fonts
    - img
    - js
- routers (controllers)      # 路由逻辑处理
- models                     # 数据逻辑层
- modules                    # 子模块
    - setting                # 应用配置存取
- cmd                        # 命令行程序命令
- conf                       # 默认配置
    - locale                 # i18n 本地化文件
- custom                     # 自定义配置
- data                       # 应用生成数据文件
- log                        # 应用生成日志文件
```

命令行应用

当应用类型为命令行应用时，需要将命令相关文件存放于 `/cmd` 目录下，并为每个命令创建一个单独的源文件：

```
/cmd
    dump.go
    fix.go
    serve.go
    update.go
    web.go
```

导入标准库、第三方或其它包

导入标准库、第三方或其它包

除标准库外，Go 语言的导入路径基本上依赖代码托管平台上的 URL 路径，因此一个源文件需要导入的包有 4 种分类：标准库、第三方包、组织内其它包和当前包的子包。

基本规则：

- 如果同时存在 2 种及以上，则需要使用分区来导入。每个分类使用一个分区，采用空行作为分区之间的分割。
- 在非测试文件（`*_test.go`）中，禁止使用 `.` 来简化导入包的对象调用。
- 禁止使用相对路径导入（`./subpackage`），所有导入路径必须符合 `go get` 标准。

下面是一个完整的示例：

```
import (
    "fmt"
    "html/template"
    "net/http"
    "os"

    "github.com/codegangsta/cli"
    "gopkg.in/macaron.v1"

    "github.com/gogits/git"
    "github.com/gogits/gfm"

    "github.com/gogits/gogs/routers"
    "github.com/gogits/gogs/routers/repo"
    "github.com/gogits/gogs/routers/user"
)
```

注释规范

注释规范

- 所有导出对象都需要注释说明其用途；非导出对象根据情况进行注释。
- 如果对象可数且无明确指定数量的情况下，一律使用单数形式和一般进行时描述；否则使用复数形式。
- 包、函数、方法和类型的注释说明都是一个完整的句子。
- 句子类型的注释首字母均需大写；短语类型的注释首字母需小写。
- 注释的单行长度不能超过 80 个字符。

包级别

- 包级别的注释就是对包的介绍，只需在同个包的任一源文件中说明即可有效。
- 对于 `main` 包，一般只有一行简短的注释用以说明包的用途，且以项目名称开头：

```
// Gogs (Go Git Service) is a painless self-hosted Git Service.
package main
```

- 对于一个复杂项目的子包，一般情况下不需要包级别注释，除非是代表某个特定功能的模块。
- 对于简单的非 `main` 包，也可用一行注释概括。
- 对于相对功能复杂的非 `main` 包，一般都会增加一些使用示例或基本说明，且以 `Package <name>` 开头：

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
```

```
14.           '$'
15.           '.'
16.           character
17.           '[' [ '^' ] character-ranges ']'
18.           '(' regexp ')'
19.    */
20.    package regexp
```

- 特别复杂的包说明，可单独创建 `doc.go` 文件来加以说明。

结构、接口及其它类型

- 类型的定义一般都以单数形式描述：

```
1.    // Request represents a request to run a command.
2.    type Request struct { ...
```

- 如果为接口，则一般以以下形式描述：

```
1.    // FileInfo is the interface that describes a file and is returned by
    Stat and Lstat.
2.    type FileInfo interface { ...
```

函数与方法

- 函数与方法的注释需以函数或方法的名称作为开头：

```
1.    // Post returns *BeegoHttpRequest with POST method.
```

- 如果一句话不足以说明全部问题，则可换行继续进行更加细致的描述：

```
1.    // Copy copies file from source to target path.
2.    // It returns false and error when error occurs in underlying function
    calls.
```

- 若函数或方法为判断类型（返回值主要为 `bool` 类型），则以 `<name> returns true if` 开头：

```
1.    // HasPrefix returns true if name has any string in given slice as
    prefix.
2.    func HasPrefix(name string, prefixes []string) bool { ...
```

其它说明

- 当某个部分等待完成时，可用 `TODO:` 开头的注释来提醒维护人员。
- 当某个部分存在已知问题进行需要修复或改进时，可用 `FIXME:` 开头的注释来提醒维护人员。
- 当需要特别说明某个问题时，可用 `NOTE:` 开头的注释：

```
1.    // NOTE: os.Chmod and os.Chtimes don't recognize symbolic link,
2.    // which will lead "no such file or directory" error.
3.    return os.Symlink(target, dest)
```

命名规则

命名规则

文件名

- 整个应用或包的主入口文件应当是 `main.go` 或与应用名称简写相同。例如：`Gogs` 的主入口文件名为 `gogs.go` 。

函数或方法

- 若函数或方法为判断类型（返回值主要为 `bool` 类型），则名称应以 `Has` ， `Is` ， `Can` 或 `Allow` 等判断性动词开头：

```
1.  func HasPrefix(name string, prefixes []string) bool { ... }
2.  func IsEntry(name string, entries []string) bool { ... }
3.  func CanManage(name string) bool { ... }
4.  func AllowGitHook() bool { ... }
```

常量

- 常量均需使用全部大写字母组成，并使用下划线分词：

```
1.  const APP_VER = "0.7.0.1110 Beta"
```

- 如果是枚举类型的常量，需要先创建相应类型：

```
1.  type Scheme string
2.
3.  const (
4.      HTTP  Scheme = "http"
5.      HTTPS Scheme = "https"
6.  )
```

- 如果模块的功能较为复杂、常量名称容易混淆的情况下，为了更好地区分枚举类型，可以使用完整的前缀：

```
1.  type PullRequestStatus int
2.
```

```
3.  const (
4.      PULL_REQUEST_STATUS_CONFLICT PullRequestStatus = iota
5.      PULL_REQUEST_STATUS_CHECKING
6.      PULL_REQUEST_STATUS_MERGEABLE
7.  )
```

变量

- 变量命名基本上遵循相应的英文表达或简写。
- 在相对简单的环境（对象数量少、针对性强）中，可以将一些名称由完整单词简写为单个字母，例如：
  - `user` 可以简写为 `u`
  - `userID` 可以简写 `uid`
- 若变量类型为 `bool` 类型，则名称应以 `Has`，`Is`，`Can` 或 `Allow` 开头：

```
1.  var isExist bool
2.  var hasConflict bool
3.  var canManage bool
4.  var allowGitHook bool
```

- 上条规则也适用于结构定义：

```
1.  // Webhook represents a web hook object.
2.  type Webhook struct {
3.      ID          int64 `xorm:"pk autoincr"`
4.      RepoID      int64
5.      OrgID       int64
6.      URL         string `xorm:"url TEXT"`
7.      ContentType HookContentType
8.      Secret      string `xorm:"TEXT"`
9.      Events      string `xorm:"TEXT"`
10.     *HookEvent  `xorm:"-"`
11.     IsSSL       bool `xorm:"is_ssl"`
12.     IsActive    bool
13.     HookTaskType HookTaskType
14.     Meta        string     `xorm:"TEXT"` // store hook-specific
    attributes
15.     LastStatus  HookStatus // Last delivery status
16.     Created     time.Time  `xorm:"CREATED"`
17.     Updated     time.Time  `xorm:"UPDATED"`
18. }
```

变量命名惯例

变量名称一般遵循驼峰法，但遇到特有名词时，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`。
- 其它情况都应当使用该名词原有的写法，如 `APIClient`、`repoID`、`UserID`。

下面列举了一些常见的特有名词：

```
// A GonicMapper that contains a list of common initialisms taken from golang/lint
var LintGonicMapper = GonicMapper{
    "API":   true,
    "ASCII": true,
    "CPU":   true,
    "CSS":   true,
    "DNS":   true,
    "EOF":   true,
    "GUID":  true,
    "HTML":  true,
    "HTTP":  true,
    "HTTPS": true,
    "ID":    true,
    "IP":    true,
    "JSON":  true,
    "LHS":   true,
    "QPS":   true,
    "RAM":   true,
    "RHS":   true,
    "RPC":   true,
    "SLA":   true,
    "SMTP":  true,
    "SSH":   true,
    "TLS":   true,
    "TTL":   true,
    "UI":    true,
    "UID":   true,
    "UUID":  true,
    "URI":   true,
    "URL":   true,
    "UTF8":  true,
    "VM":    true,
```

```
33.     "XML":   true,
34.     "XSRF":  true,
35.     "XSS":   true,
36. }
```

声明语句

声明语句

函数或方法

函数或方法的参数排列顺序遵循以下几点原则（从左到右）：

1. 参数的重要程度与逻辑顺序
2. 简单类型优先于复杂类型
3. 尽可能将同种类型的参数放在相邻位置，则只需写一次类型

示例

以下声明语句，`User` 类型要复杂于 `string` 类型，但由于 `Repository` 是 `User` 的附属品，首先确定 `User` 才能继而确定 `Repository`。因此，`User` 的顺序要优先于 `repoName`。

```
1.  func IsRepositoryExist(user *User, repoName string) (bool, error) { ...
```

代码指导

代码指导

基本约束

- 所有应用的 `main` 包需要有 `APP_VER` 常量表示版本，格式为 `X.Y.Z.Date [Status]`，例如：`0.7.6.1112 Beta`。
- 单独的库需要有函数 `Version` 返回库版本号的字符串，格式为 `X.Y.Z[.Date]`。
- 当单行代码超过 80 个字符时，就要考虑分行。分行的规则是以参数为单位将从较长的参数开始换行，以此类推直到每行长度合适：

```
1.    So(z.ExtractTo(
2.        path.Join(os.TempDir(), "testdata/test2"),
3.        "dir/", "dir/bar", "readonly"), ShouldBeNil)
```

- 当单行声明语句超过 80 个字符时，就要考虑分行。分行的规则是将参数按类型分组，紧接着的声明语句的是一个空行，以便和函数体区别：

```
1.    // NewNode initializes and returns a new Node representation.
2.    func NewNode(
3.        importPath, downloadUrl string,
4.        tp RevisionType, val string,
5.        isGetDeps bool) *Node {
6.
7.        n := &Node{
8.            Pkg: Pkg{
9.                ImportPath: importPath,
10.               RootPath:   GetRootPath(importPath),
11.               Type:       tp,
12.               Value:      val,
13.           },
14.           DownloadURL: downloadUrl,
15.           IsGetDeps:   isGetDeps,
16.       }
17.       n.InstallPath = path.Join(setting.InstallRepoPath, n.RootPath) +
    n.ValSuffix()
18.       return n
19.   }
```

- 分组声明一般需要按照功能来区分，而不是将所有类型都分在一组：

```
  const (
      // Default section name.
      DEFAULT_SECTION = "DEFAULT"
      // Maximum allowed depth when recursively substituing variable
names.
      _DEPTH_VALUES = 200
  )

  type ParseError int

  const (
      ERR_SECTION_NOT_FOUND ParseError = iota + 1
      ERR_KEY_NOT_FOUND
      ERR_BLANK_SECTION_NAME
      ERR_COULD_NOT_PARSE
  )
```

- 当一个源文件中存在多个相对独立的部分时，为方便区分，需使用由 ASCII Generator 提供的句型字符标注（示例： `Comment` ）：

```
  // _________                                          __
  // \_   ___ \  ____   _____   _____   ____   _____/  |_
  // /    \  \/ /  _ \ /     \ /     \_/ __ \ /    \   __\
  // \     \___(  <_> )  Y Y  \  Y Y  \  ___/|   |  \  |
  //  \______  /\____/|__|_|  /__|_|  /\___  >___|  /__|
  //         \/             \/      \/     \/     \/
```

- 函数或方法的顺序一般需要按照依赖关系由浅入深由上至下排序，即最底层的函数出现在最前面。例如，下方的代码，函数 `ExecCmdDirBytes` 属于最底层的函数，它被 `ExecCmdDir` 函数调用，而 `ExecCmdDir` 又被 `ExecCmd` 调用：

```
  // ExecCmdDirBytes executes system command in given directory
  // and return stdout, stderr in bytes type, along with possible error.
  func ExecCmdDirBytes(dir, cmdName string, args ...string) ([]byte,
[]byte, error) {
      ...
  }
```

```
// ExecCmdDir executes system command in given directory
// and return stdout, stderr in string type, along with possible error.
func ExecCmdDir(dir, cmdName string, args ...string) (string, string, error) {
    bufOut, bufErr, err := ExecCmdDirBytes(dir, cmdName, args...)
    return string(bufOut), string(bufErr), err
}

// ExecCmd executes system command
// and return stdout, stderr in string type, along with possible error.
func ExecCmd(cmdName string, args ...string) (string, string, error) {
    return ExecCmdDir("", cmdName, args...)
}
```

- 结构附带的方法应置于结构定义之后，按照所对应操作的字段顺序摆放方法：

```
type Webhook struct { ... }
func (w *Webhook) GetEvent() { ... }
func (w *Webhook) SaveEvent() error { ... }
func (w *Webhook) HasPushEvent() bool { ... }
```

- 如果一个结构拥有对应操作函数，大体上按照 `CRUD` 的顺序放置结构定义之后：

```
func CreateWebhook(w *Webhook) error { ... }
func GetWebhookById(hookId int64) (*Webhook, error) { ... }
func UpdateWebhook(w *Webhook) error { ... }
func DeleteWebhook(hookId int64) error { ... }
```

- 如果一个结构拥有以 `Has` 、 `Is` 、 `Can` 或 `Allow` 开头的函数或方法，则应将它们至于所有其它函数及方法之前；这些函数或方法以 `Has` 、 `Is` 、 `Can` 、 `Allow` 的顺序排序。

- 变量的定义要放置在相关函数之前：

```
var CmdDump = cli.Command{
    Name:  "dump",
    ...
    Action: runDump,
    Flags:  []cli.Flag{},
}

func runDump(*cli.Context) { ...
```

- 在初始化结构时，尽可能使用一一对应方式：

```
AddHookTask(&HookTask{
    Type:        HTT_WEBHOOK,
    Url:         w.Url,
    Payload:     p,
    ContentType: w.ContentType,
    IsSsl:       w.IsSsl,
})
```

测试用例

测试用例

- 单元测试都必须使用 `GoConvey` 编写，且辅助包覆盖率必须在 `80%` 以上。

使用示例

- 为辅助包书写使用示例的时，文件名均命名为 `example_test.go` 。
- 测试用例的函数名称必须以 `Test_` 开头，例如：`Test_Logger` 。
- 如果为方法书写测试用例，则需要以 `Text_<Struct>_<Method>` 的形式命名，例如：`Test_Macaron_Run` 。