

Golang代码规范

Posted on August 2, 2017

Golang代码规范

参考https://golang.org/doc/effective_go.html (https://golang.org/doc/effective_go.html)

项目目录结构规范

```
PROJECT_NAME
├── README.md 介绍软件及文档入口
├── bin 编译好的二进制文件, 执行./build.sh自动生成, 该目录也用于程序打包
├── build.sh 自动编译的脚本
├── doc 该项目的文档
├── pack 打包后的程序放在此处
├── pack.sh 自动打包的脚本, 生成类似xxxx.20170713_14:45:35.tar.gz的文件, 放在pack文件下
├── src 该项目的源代码
│   ├── main 项目主函数
│   ├── model 项目代码
│   ├── research 在实现该项目中探究的一些程序
│   └── vendor 存放go的库
│       ├── github.com/xxx (http://github.com/xxx) 第三方库
│       └── xxx.com/obc (http://xxx.com/obc) 公司内部的公共库
```

项目的目录结构尽量做到简明、层次清楚

文件名命名规范

用小写，尽量见名思义，看见文件名就可以知道这个文件下的大概内容，对于源代码里的文件，文件名要很好的代表了一个模块实现的功能。

命名规范

包名

包名用小写,使用短命名,尽量和标准库不要冲突

接口名

单个函数的接口名以”er”作为后缀，如Reader,Writer

接口的实现则去掉“er”

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

两个函数的接口名综合两个函数名

```
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()  
}
```

变量

全局变量：采用驼峰命名法，仅限在包内的全局变量，包外引用需要写接口，提供调用 局部变量：驼峰式，小写字母开头

常量

常量：大写，采用下划线

import 规范

import在多行的情况下，goimports会自动帮你格式化，在一个文件里面引入了一个package，建议采用如下格式：

```
import (  
    "fmt"  
)
```

如果你的包引入了三种类型的包，标准库包，程序内部包，第三方包，建议采用如下方式进行组织你的包：

```
import (  
    "encoding/json"  
    "strings"  
  
    "myproject/models"  
    "myproject/controller"  
    "git.obc.im/obc/utils (http://git.obc.im/obc/utils)"  
  
    "git.obc.im/dep/beego (http://git.obc.im/dep/beego)"  
    "git.obc.im/dep/mysql (http://git.obc.im/dep/mysql)"  
)
```

在项目中不要使用相对路径引入包：

// 这是不好的导入

```
import "../net"
```

// 这是正确的做法

```
import "xxxx.com/proj/net (http://xxxx.com/proj/net)"
```

函数名

函数名采用驼峰命名法，尽量不要使用下划线

错误处理

error作为函数的值返回,必须尽快对error进行处理

采用独立的错误流进行处理

不要采用这种方式

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

而要采用下面的方式

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

如果返回值需要初始化，则采用下面的方式

```
x, err := f()  
if err != nil {  
    // error handling  
    return  
}  
// use x
```

Panic

在逻辑处理中禁用panic

在main包中只有当实在不可运行的情况采用panic，例如文件无法打开，数据库无法连接导致程序无法正常运行，但是对于其他的package对外的接口不能有panic，只能在包内采用。建议在main包中使用log.Fatal来记录错误，这样就可以由log来结束程序。

Recover

recover用于捕获runtime的异常，禁止滥用recover，在开发测试阶段尽量不要用recover，recover一般放在你认为会有不可预期的异常的地方。

```
func server(workChan <-chan *Work) {  
    for work := range workChan {  
        go safelyDo(work)  
    }  
}  
  
func safelyDo(work *Work) {  
    defer func() {  
        if err := recover(); err != nil {  
            log.Println("work failed:", err)  
        }  
    }()  
    // do 函数可能会有不可预期的异常  
    do(work)  
}
```

Defer

defer在函数return之前执行，对于一些资源的回收用defer是好的，但也禁止滥用defer，defer是需要消耗性能的,所以频繁调用的函数尽量不要使用defer。

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close() // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

控制结构

if

if接受初始化语句，约定如下方式建立局部变量

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

for

采用短声明建立局部变量

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

range

如果只需要第一项（key），就丢弃第二个：

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

如果只需要第二项，则把第一项置为下划线

```
sum := 0
for _, value := range array {
    sum += value
}
```

return

尽早return：一旦有错误发生，马上返回

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

方法的接收器

名称 一般采用strcut的第一个字母且为小写，而不是this，me或者self

```
type T struct{}  
func (p *T)Get(){}
```

如果接收者是map,slice或者chan，不要用指针传递

```
//Map  
package main  
  
import (  
    "fmt"  
)  
  
type mp map[string]string  
  
func (m mp) Set(k, v string) {  
    m[k] = v  
}  
  
func main() {  
    m := make(mp)  
    m.Set("k", "v")  
    fmt.Println(m)  
}
```

```
//Channel  
package main  
  
import (  
    "fmt"  
)  
  
type ch chan interface{}  
  
func (c ch) Push(i interface{}) {  
    c <- i  
}  
  
func (c ch) Pop() interface{} {  
    return <-c  
}  
  
func main() {  
    c := make(ch, 1)  
    c.Push("i")  
    fmt.Println(c.Pop())  
}
```

如果需要对slice进行修改，通过返回值的方式重新赋值

```
//Slice
package main

import (
    "fmt"
)

type slice []byte

func main() {
    s := make(slice, 0)
    s = s.addOne(42)
    fmt.Println(s)
}

func (s slice) addOne(b byte) []byte {
    return append(s, b)
}
```

如果接收者是含有sync.Mutex或者类似同步字段的结构体，必须使用指针传递避免复制


```
package main

import (
    "sync"
)

type T struct {
    m sync.Mutex
}

func (t *T) lock() {
    t.m.Lock()
}

/*
Wrong !!!
func (t T) lock() {
    t.m.Lock()
}
*/

func main() {
    t := new(T)
    t.lock()
}
```

如果接收者是大的结构体或者数组，使用指针传递会更有效率。

```
package main

import (
    "fmt"
)

type T struct {
    data [1024]byte
}

func (t *T) Get() byte {
    return t.data[0]
}



func main() {
    t := new(T)
    fmt.Println(t.Get())
}
```

← **PREVIOUS POST** ([HTTPS://SHEEPBAO.GITHUB.IO/POST/GOLANG_UDP_PROGRAMING/](https://sheepbao.github.io/post/golang_udp_programing/))

NEXT POST → ([HTTPS://SHEEPBAO.GITHUB.IO/POST/GOLANG_CODE_SLICE/](https://sheepbao.github.io/post/golang_code_slice/))

0 Comments sheepbao

 Login ▾

 Recommend  Share

Sort by Best ▾






Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Be the first to comment.

 Subscribe  Add Disqus to your siteAdd DisqusAdd  Privacy

• 2018 • boya blog (<https://sheepbao.github.io/>)

Hugo v0.26 (<http://gohugo.io>) powered • Theme by Beautiful Jekyll (<http://deanattali.com/beautiful-jekyll/>) adapted to
Beautiful Hugo (<https://github.com/halogenica/beautifulhugo>)