golang 编码规范

注:此文档参考官方指南 Effective Golang 和 Golang Code Review Comments 进行整理，力图与官方及社区编码风格保持一致。

1. gofmt

大部分的格式问题可以通过 gofmt 解决，gofmt 自动格式化代码，保证所有的 go 代码一致的格式。

正常情况下，采用 Sublime 编写 go 代码时，插件 GoSublilme 已经调用 gofmt 对代码实现了格式化。

2. 注释

在编码阶段同步写好变量、函数、包注释，注释可以通过 godoc 导出生成文档。

注释必须是完整的句子，以需要注释的内容作为开头，句点作为结尾。

程序中每一个被导出的（大写的）名字，都应该有一个文档注释。

2.1 包注释

每个程序包都应该有一个包注释，一个位于 package 子句之前的块注释或行注释。

包如果有多个 go 文件，只需要出现在一个 go 文件中即可。

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

2.2 可导出方法

第一条语句应该为一条概括语句，并且使用被声明的名字作为开头。

```
// Compile parses a regular expression and returns, if successful, a Regexp
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, err error) {
```

3. 命名

使用短命名，长名字并不会自动使得事物更易读，文档注释会比格外长的名字更有用。

3.1 包名

包名应该为小写单词，不要使用下划线或者混合大小写。

3.2 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()
}
```

3.3 混合大小写

采用驼峰式命名

```
MixedCaps 大写开头，可导出
mixedCaps 小写开头，不可导出
```

4 控制结构

4.1 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
}
```

4.2 for

采用短声明建立局部变量

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

```
    sum += i
}
```

4.3 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
}
```

4.4 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)
```

5. 函数（必须）

5.1 函数采用命名的多值返回

5.2 传入变量和返回变量以小写字母开头

```
    func nextInt(b []byte, pos int) (value, nextPos int) {
```

在 godoc 生成的文档中，带有返回值的函数声明更利于理解

6. 错误处理

6.1 error 作为函数的值返回,必须对 error 进行处理

6.2 错误描述如果是英文必须为小写，不需要标点结尾

6.3 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
```

7. panic

尽量不要使用 panic，除非你知道你在做什么

8. import

对 import 的包进行分组管理，而且标准库作为第一组

```
package main

import (
    "fmt"
    "hash/adler32"
    "os"

    "appengine/user"
    "appengine/foo"

    "code.google.com/p/x/y"
```

```
        "github.com/foo/bar"
    )
```

goimports 实现了自动格式化

9. 缩写

采用全部大写或者全部小写来表示缩写单词

比如对于 url 这个单词，不要使用

UrlPony

而要使用

urlPony 或者 URLPony

10. 参数传递

10.1 对于少量数据，不要传递指针

10.2 对于大量数据的 struct 可以考虑使用指针

10.3 传入参数是 map，slice，chan 不要传递指针

因为 map，slice，chan 是引用类型，不需要传递指针的指针

11. 接受者

11.1 名称

统一采用单字母'p'而不是 this，me 或者 self

```
type T struct{}

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

11.2 类型

对于 go 初学者，接受者的类型如果不清楚，统一采用指针型

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

而不是

```
func (p T)Get(){}
```

在某些情况下，出于性能的考虑，或者类型本来就是引用类型，有一些特例

11.3 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())
}
```

11.4 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)
}
```

11.5 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()
}
```

11.6 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())
}
```