spot-palm 编程规范

命名

选取一个合适的命名有时候确实是很难的，来看下有哪些可以帮我我们命名的技巧

1、命名的长度选择
关于命名长度，在能够表达含义的额情况下，命名当然是越短越好。在大多数的情况下，短的命名不如长的命名更能表达含义，很多书籍是不推荐使用缩写的。

尽管长的命名可以包含更多的信息，更能准确直观地表达意图，但是，如果函数、变量的命名很长，那由它们组成的语句就会很长。在代码列长度有限制的情况下，就会经常出现一条语句被分割成两行的情况，这其实会影响代码可读性。

所以有时候我们是可以适量的使用缩写的短命名

在什么场景下合适使用短命名

1、对于一些默认，大家都熟知的倒是可以使用缩写的命名，比如，sec 表示 second、str 表示 string、num 表示 number、doc 表示 document 等等

2、对于作用域比较小的变量，我们可以使用相对短的命名，比如一些函数内的临时变量，相对应的对于作用于比较大的，更推荐使用长命名

2、利用上下文简化命名
来看个栗子

```
type User struct {
    UserName      string
    UserAge       string
    UserAvatarUrl string
}
```

比如这个struct，我们已经知道这是一个 User 信息的 struct。里面用户的 name ,age，就没有必要加上user的前缀了

修改后的

```
type User struct {
    Name      string
    Age       string
    AvatarUrl string
```

```
}
```

当然这个在数据库的设计中也是同样有用

3、命名要可读、可搜索
“可读”，指的是不要用一些特别生僻、难发音的英文单词来命名。

我们在IDE中编写代码的时候，经常会用“关键词联想”的方法来自动补全和搜索。比如，键入某个对象“.get”，希望IDE返回这个对象的所有get开头的方法。再比如，通过在IDE搜索框中输入“Array”，搜索JDK中数组相关的函数和方法。所以，我们在命名的时候，最好能符合整个项目的命名习惯。大家都用“selectXXX”表示查询，你就不要用“queryXXX”；大家都用“insertXXX”表示插入一条数据，你就要不用“addXXX”，统一规约是很重要的，能减少很多不必要的麻烦。

4、如何命名接口
对于接口的命名，一般有两种比较常见的方式。一种是加前缀“I”，表示一个Interface。比如IUserService，对应的实现命名为UserService。另一种是不加前缀，比如UserService，对应的实现加后缀“Impl”，比如UserServiceImpl。

注释

我们接受一个项目的时候，经常会吐槽老项目注释不好，文档不全，那么如果注释都让我们去写，怎样的注释才是好的注释

有时候我们会在书籍或一些博客中看到，如果好的命名是不需要注释的，也就是代码即注释，如果需要注释了，就是代码的命名不好了，需要在命名中下功夫。

这种是有点极端了，命名再好，毕竟有长度限制，不可能足够详尽，而这个时候，注释就是一个很好的补充。

1、注释到底该写什么
我们写数注释的目的是让代码更易懂，注释一般包括三个方面，做什么、为什么、怎么做。

这是 golang 中 sync.map中的注释，也是分别从做什么、为什么、怎么做 来进行注释

```
// Map is like a Go map[interface{}]interface{} but is safe for concurrent
use
// by multiple goroutines without additional locking or coordination.
// Loads, stores, and deletes run in amortized constant time.
//
// The Map type is specialized. Most code should use a plain Go map instead,
// with separate locking or coordination, for better type safety and to make
it
// easier to maintain other invariants along with the map content.
```

```
//
// The Map type is optimized for two common use cases: (1) when the entry for
a given
// key is only ever written once but read many times, as in caches that only
grow,
// or (2) when multiple goroutines read, write, and overwrite entries for
disjoint
// sets of keys. In these two cases, use of a Map may significantly reduce
lock
// contention compared to a Go map paired with a separate Mutex or RWMutex.
//
// The zero Map is empty and ready for use. A Map must not be copied after
first use.
type Map struct {
    mu Mutex
    read atomic.Value // readOnly
    dirty map[interface{}]*entry
    misses int
}
```

有些人认为，注释是要提供一些代码没有的额外信息，所以不要写“做什么、怎么做”，这两方面在代码中都可以体现出来，只需要写清楚“为什么”，表明代码的设计意图即可。

不过写了注释可能有以下几个优点

1、注释比代码承载的信息更多

函数和变量如果命名得好，确实可以不用再在注释中解释它是做什么的。但是，对结构体来说，包含的信息比较多，一个简单的命名就不够全面详尽了。这个时候，在注释中写明“做什么”就合情合理了。

2、注释起到总结性作用、文档的作用

在注释中，关于具体的代码实现思路，我们可以写一些总结性的说明、特殊情况的说明。这样能够让阅读代码的人通过注释就能大概了解代码的实现思路，阅读起来就会更加容易。

3、一些总结性注释能让代码结构更清晰

对于逻辑比较复杂的代码或者比较长的函数，如果不好提炼、不好拆分成小的函数调用，那我们可以借助总结性的注释来让代码结构更清晰、更有条理。

2、注释是不是越多越好
注释本身有一定的维护成本，所以并非越多越好。结构体和函数一定要写注释，而且要写得尽可能全面、详细，而函数内部的注释要相对少一些，一般都是靠好的命名、提炼函数、解释性变量、总结性注释来提高代码可读性。

代码风格

1、函数多大才合适
函数的代码太多和太少，都是不太好的

太多了：

一个方法上千行，一个函数几百行，逻辑过于繁杂，阅读代码的时候，很容易就会看了后面忘了前面

太少了：

在代码总量相同的情况下，被分割成的函数就会相应增多，调用关系就会变得更复杂，阅读某个代码逻辑的时候，需要频繁地在n多方法或者n多函数之间跳来跳去，阅读体验也不好。

多少最合适的呢?

不过很难给出具体的值，有的地方会讲，那就是不要超过一个显示屏的垂直高度。比如，在我的电脑上，如果要让一个函数的代码完整地显示在IDE中，那最大代码行数不能超过50。

2、一行代码多长最合适
这个也没有一个完全的准侧，毕竟语言不同要求也是不同的

当然有个通用的原则：一行代码最长不能超过IDE显示的宽度。

太长了就不方便代码的阅读了

3、善用空行分割单元块
也就是垂直留白，不太建议我们的代码写下来，一个函数或方法中一行空格也没余，通常会根据不同的语义，一个小模块的内容完了，通过空白空格进行分割。

```
// Store sets the value for a key.
func (m *Map) Store(key, value interface{}) {
    read, _ := m.read.Load().(readOnly)
    if e, ok := read.m[key]; ok && e.tryStore(&value) {
        return
    }

    m.mu.Lock()
    // ...
    m.mu.Unlock()
}
```

这里上锁的代码就和上文进行了空格

当然有的地方会讲首行不空格，这也是对的，函数头部的空行是没有任何用的。

编程技巧

1、把代码分割成更小的单元块
善于将代码中的模块进行抽象，能够方便我们的阅读

所以，我们要有模块化和抽象思维，善于将大块的复杂逻辑提炼成小的方法或函数，屏蔽掉细节，让阅读代码的人不至于迷失在细节中，这样能极大地提高代码的可读性。不过，只有代码逻辑比较复杂的时候，我们其实才建议把对应的逻辑提炼出来。

2、避免函数或方法参数过多
函数包含3、4个参数的时候还是能接受的，大于等于5个的时候，我们就觉得参数有点过多了，会影响到代码的可读性，使用起来也不方便。

针对这种情况有两种处理方法

1、考虑函数是否职责单一，是否能通过拆分成多个函数的方式来减少参数。

2、将函数的参数封装成对象。

栗子

```
func updateBookshelf(userId, deviceId string, platform, channel, step int) {
    // ...
}

// 修改后
type UpdateBookshelfInput struct {
    UserId   string
    DeviceId string
    Step     int
    Platform int
    Channel  int
}

func updateBookshelf(input *UpdateBookshelfInput) {
    // ...
}
```

3、勿用函数参数来控制逻辑
不要在函数中使用布尔类型的标识参数来控制内部逻辑，true的时候走这块逻辑，false的时候走另一块逻辑。这明显违背了单一职责原则和接口隔离原则。

可以拆分成两个函数分别调用

栗子

```
func sendVip(userId string, isNewUser bool) {
    // 是新用户
    if isNewUser {
        // ...
    } else {
        // ...
    }
}

// 修改后
func sendVip(userId string) {
    // ...
}

func sendNewUserVip(userId string) {
    // ...
}
```

不过，如果函数是private私有函数，影响范围有限，或者拆分之后的两个函数经常同时被调用，我们可以酌情考虑不用拆分。

4、函数设计要职责单一
对于函数的设计我们也要尽量职责单一，避免设计一个大而全的函数，可以根据不同的功能点，对函数进行拆分。

举个栗子：我们来校验下我们的额一些用户属性，当然这个校验就省略成判断是否为空了

```
func validate(name, phone, email string) error {
    if name == "" {
        return errors.New("name is empty")
    }

    if phone == "" {
        return errors.New("phone is empty")
    }

    if email == "" {
        return errors.New("name is empty")
    }
    return nil
}
```

修改过就是

```
func validateName(name string) error {
    if name == "" {
        return errors.New("name is empty")
    }

    return nil
}

func validatePhone( phone string) error {
    if phone == "" {
        return errors.New("phone is empty")
    }

    return nil
}

func validateEmail(name, phone, email string) error {
    if email == "" {
        return errors.New("name is empty")
    }
    return nil
}
```

5、移除过深的嵌套层次
代码嵌套层次过深往往是因为if-else、switch-case、for循环过度嵌套导致的。过深的嵌套，代码除了不好理解外，嵌套过深很容易因为代码多次缩进，导致嵌套内部的语句超过一行的长度而折成两行，影响代码的整洁。

对于嵌套代码的修改，大概有四个方向可以考虑

举个栗子：

这段代码中，有些地方是不太合适的，我们从下面的四个方向来分析

```
func sum(sil []*User, age int) int {
    count := 0
    if len(sil) == 0 || age == 0 {
        return count
    } else {
        for _, item := range sil {
            if item.Age > age {
                count++
            } else {
                // do something
                // ....
            }
```

```
        }
    }
    return count
}
```

1、去掉多余的if或else语句

修改为

```
func sum(sil []*User, age int) int {
    count := 0
    if len(sil) != 0 && age == 0 {
        for _, item := range sil {
            if item.Age > age {
                count++
            } else {
                // do something
                // ....
            }
        }
    }
    return count
}
```

2、使用编程语言提供的continue、break、return关键字，提前退出嵌套

```
func sum(sil []*User, age int) int {
    count := 0
    if len(sil) != 0 && age == 0 {
        for _, item := range sil {
            if item.Age > age {
                count++
                continue
            }
            // do something
            // ....
        }
    }
    return count
}
```

3、调整执行顺序来减少嵌套

```
func sum(sil []*User, age int) int {
    count := 0
    if len(sil) == 0 || age == 0 {
```

```
        return count
    }

    for _, item := range sil {
        if item.Age > age {
            count++
            continue
        }
        // do something
        // ....
    }

    return count
}
```

4、将部分嵌套逻辑封装成函数调用，以此来减少嵌套

```
func sum(sil []*User, age int) int {
    count := 0
    if len(sil) == 0 || age == 0 {
        return count
    }

    for _, item := range sil {
        if item.Age > age {
            count++
            continue
        }
        dealUser(item, age)
    }

    return count
}

func dealUser(user *User, age int) {
    if user.Age > age {
        return
    }

    // do something
    // ....
}
```

6、学会使用解释性变量

常用的用解释性变量来提高代码的可读性的情况有下面2种

1、常量取代魔法数字

```
func CalculateCircularArea(radius float64) float64 {

    return 3.1415 * radius * radius
}

// 修改后
const PI = 3.1415
func CalculateCircularArea(radius float64) float64 {

    return PI * radius * radius
}
```

2、使用解释性变量来解释复杂表达式

```
if appOnlineTime.Before(userId.Timestamp()) {
    appOnlineTime = userId.Timestamp()
}

// 修改后
isBeforeRegisterTime := appOnlineTime.Before(userId.Timestamp())
if isBeforeRegisterTime {
    appOnlineTime = userId.Timestamp()
}
```