# Golang Practice

The bigger the interface, the weaker the abstraction.

工作区模式：
https://github.com/jincheng9/go-tutorial/tree/main/workspace/senior/p25

**乱用 Interface 的场景**
interface 在 Go 代码里经常被乱用，不少 C#或者 Java 开发背景的人在转 Go 的时候，通常会先把接口类型抽象好，再去定义具体的类型。
然而，这并不是 Go 里推荐的。
Don't design with interfaces, discover them.
—Rob Pike
正如 Rob Pike 所说，不要一上来做代码设计的时候就先把 interface 给定义了。
除非真的有需要，否则是不推荐一开始就在代码里使用 interface 的。
最佳实践应该是先不要想着 interface，因为过度使用 interface 会让代码晦涩难懂。
我们应该先按照没有 interface 的场景去写代码，如果最后发现使用 interface 能带来额外的好处，再去使用 interface。

GRPC:https://github.com/jincheng9/go-tutorial/tree/main/workspace/rpc/02

**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

在编译期检查接口的实现
```
var _ io.Reader = (*MyFastReader)(nil)
```

len(nil) = 0

匿名结构很酷
```
var hits struct {
    sync.Mutex
    n int
}
hits.Lock()
hits.n++
hits.Unlock()
```

获得调用堆栈，我们可以使用 runtime.Caller

要 marshal 任意的 JSON， 你可以 marshal 为 map[string]interface{}{}
从一个 slice 生成简单的随机元素

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

打印堆栈：
```
import(
  "runtime/debug"
)
...
debug.PrintStack()
```

Performance and concurrency are important attributes, but not as important as *simplicity*, *readability*, and *productivity*.

- 返回 struct 内部信息的方法，使用值调用
- 修改 struct 内部信息的方法，使用指针调用

slices、map 传递的时候虽然是值传递，但是由于内部是结构体，因此在函数内部对其进行变更的时候进行变更的时候，会影响到外部。

转换字符串的时候，strconv 的性能好于 fmt

断言失败也需要显式处理，否则会触发 panic
避免内部直接使用 panic

包名（下面规则进行选择）
全部小写，无大写以及下划线
不使用复数
不使用 common、util、shared、lib 这类信息不足的包名
同个包下面可以根据职责分多文件

按调用循序排序，后调用的在后面

nil 是合法的 slice，为一个空的 slice，应该使用 len 来判空
对零值变量使用 var 而不是:=

接口命名
- 单个函数的接口名以"er"作为后缀，例如 **type Reader interface {...}**
- 两个函数的接口名综合两个函数名，例如 **type WriteFlusher interface {...}**
- 三个以上函数的接口名，类似于结构体名，例如 **type Car interface {...}**

特殊名词的首字母缩写需要按照规范来，例如 **URLProxy 或者 urlProxy** 不要命名为 **UrlProxy**。

任何一个 goroutine 都应该有 recover 来保护程序不会因为 panic 而 crash，因为任何

一个 goroutine 如果抛 panic 但没有 recover 整个程序会 crash

如果 package 的名字由多个单词组成，需要全部小写，且中间不要用任何符号做分隔。

1. 被 defer 的函数的参数在执行到 defer 语句的时候就被确定下来了。
   func a() {
2.  i := 0
3.  defer fmt.Println(i) // 最终打印 0
4.  i++
5.  return
6. }

   上例中，被 defer 的函数 fmt.Println 的参数 i 在执行到 defer 这一行的时候，i 的值是 0，fmt.Println 的参数就被确定下来是 0 了，因此最终打印的结果是 0，而不是 1。

- Go 里没有构造函数，Go 的 new 不会去调用构造函数。C++ 的 new 是会调用对应类型的构造函数。

make 有如下几个特点：
1. 分配和初始化内存。
2. 只能用于 slice, map 和 chan 这 3 个类型，不能用于其它类型。
   如果是用于 slice 类型，make 函数的第 2 个参数表示 slice 的长度，这个参数必须给值。
3. 返回的是原始类型，也就是 slice, map 和 chan，不是返回指向 slice, map 和 chan 的指针。

- 为什么针对 slice, map 和 chan 类型专门定义一个 make 函数?
  答案：这是因为 slice, map 和 chan 的底层结构上要求在使用 slice，map 和 chan 的时候必须初始化，如果不初始化，那 slice，map 和 chan 的值就是零值，也就是 nil。我们知道：
  - map 如果是 nil，是不能往 map 插入元素的，插入元素会引发 panic
  - chan 如果是 nil，往 chan 发送数据或者从 chan 接收数据都会阻塞
  - slice 会有点特殊，理论上 slice 如果是 nil，也是没法用的。但是 append 函数处理了 nil slice 的情况，可以调用 append 函数对 nil slice 做扩容。但是我们使用 slice，总是会希望可以自定义长度或者容量，这个时候就需要用到 make。

1. channel 被关闭后：
   - 往被关闭的 channel 发送数据会触发 panic。
   - 从被关闭的 channel 接收数据，会先读完 channel 里的数据。如果数据读完了，继续从 channel 读数据会拿到 channel 里存储的元素类型的零值。

data, ok := <- c

对于上面的代码，如果channel c关闭了，继续从c里读数据，当c里还有数据时，data就是对应读到的值，ok的值是true。如果c的数据已经读完了，那data就是零值，ok的值是false。

- channel被关闭后，如果再次关闭，会引发panic。

1. select的运行机制如下：
   - 选取一个可执行不阻塞的case分支，如果多个case分支都不阻塞，会随机选一个case分支执行，和case分支在代码里写的顺序没关系。
   - 如果所有case分支都阻塞，会进入default分支执行。
   - 如果没有default分支，那select会阻塞，直到有一个case分支不阻塞。

**map变量是指向runtime.hmap的指针,在golang中map，slice都是特殊类型的指针，**
**对里面取对象其实就是在做指针便宜 *(p+4) 类似这样的操作，这样可以避免原始指针偏移的乱用**
公有和私有代码划分

## 按职责拆分

Go 语言在拆分模块时就使用了完全不同的思路，虽然 MVC 架构模式是在我们写 Web 服务时无法避开的，但是相比于横向地切分不同的层级，Go 语言的项目往往都按照职责对模块进行拆分

**Being confused about nil vs. empty slice (#22)**
To prevent common confusions such as when using the encoding/json or the reflect package, you need to understand the difference between nil and empty slices. Both are zero-length, zero-capacity slices, but only a nil slice doesn't require allocation.
**Not properly checking if a slice is empty (#23)**
To check if a slice doesn't contain any element, check its length. This check works regardless of whether the slice is nil or empty. The same goes for maps. To design unambiguous APIs, you shouldn't distinguish between nil and empty slices.
**Map and memory leaks (#28)**
A map can always grow in memory, but it never shrinks. Hence, if it leads to some memory issues, you can try different options, such as forcing Go to recreate the map or using pointers.
**Ignoring that elements are copied in range loops (#30)**
The value element in a range loop is a copy. Therefore, to mutate a struct, for example, access it via its index or via a classic for loop (unless the element or the field you want to modify is a pointer).
**Using defer inside a loop (#35)**

Extracting loop logic inside a function leads to executing a defer statement at the end of each iteration.

- 过滤但不分配新内存

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

- 不要忘记停止 ticker, 除非你需要泄露 channel

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

用这个命令 go build -ldflags="-s -w" ... 去掉你的二进制文件

**【必须】nil指针判断**

- 进行指针操作时，必须判断该指针是否为nil，防止程序panic，尤其在进行结构体Unmarshal时

```
// good
c := cors.New(cors.Options{
    AllowedOrigins:   []string{"http://qq.com", "https://qq.com"},
    AllowCredentials: true,
    Debug:            false,
})
```

Panic应该是一种静态行为，你明确知道在什么情况下触发（比如断电，网络失败，磁盘损坏）

- 编译器在编译期决定变量分配在stack还是heap上，需要做逃逸分析(escape analysis)，逃逸分析在编译阶段就完成了。

译器会尽可能把能分配在栈上的对象分配在栈上，避免堆内存频繁GC垃圾回收带来的系统开销，影响程序性能(只有heap内存空间才会发生GC)。

```
func getFooValue() foo {
    var result foo
```

```
    // Do something
    return result
}
```
变量result定义的时候会在这个goroutine的stack上分配result的内存空间

goroutine的stack空间足够存储，如果foo占用的空间过大，在stack里存储不了，就会分配内存到heap上。

```
func getFooPointer() *foo {
    var result foo
    // Do something
    return &result
}
```
函数getFooPointer因为返回的是一个指针，如果变量result分配在stack上，那函数返回后，result的内存空间会被释放，就会导致接受函数返回值的变量无法访问原本result的内存空间，成为一个悬浮指针(dangling pointer)。
所以这种情况会发生内存逃逸，result会分配在heap上，而不是stack上。

发生逃逸行为，Go编译器会将变量存储在heap上

- 函数内局部变量在函数外部被引用
- 接口(interface)类型的变量
- size未知或者动态变化的变量，如slice，map，channel，[]byte等
- size过大的局部变量，因为stack内存空间比较小。

- go build -gcflags="-m"，可以展示逃逸分析、内联优化等各种优化结果。
- go build -gcflags="-m -l"，-l会禁用内联优化，这样可以过滤掉内联优化的结果展示，让我们可以关注逃逸分析的结果。
- go build -gcflags="-m -m"，多一个-m会展示更详细的分析结果。

**:分割操作符**
:分割操作符有几个特点：

1. :可以对数组或者slice做数据截取，:得到的结果是一个新slice。
2. **新slice结构体里的array指针指向原数组或者原slice的底层数组，新切片的长度是：右边的数值减去左边的数值，新切片的容量是原切片的容量减去:左边的数值。**

append不会改变原切片的值，原切片的长度和容量都不变，除非把append的返回值赋值给原切片。

- 如果原切片的容量足以包含新增加的元素，那append函数返回的切片结构里3个字段的值是：
    - array指针字段的值不变，和原切片的array指针的值相同，也就是append是在原切片的底层数组添加元素，返回的切片还是指向原切片的底层数组
    - len长度字段的值做相应增加，增加了N个元素，长度就增加N

    - cap 容量不变
- 如果原切片的容量不够存储 append 新增加的元素，Go 会先分配一块容量更大的新内存，然后把原切片里的所有元素拷贝过来，最后在新的内存里添加新元素。append 函数返回的切片结构里的 3 个字段的值是：
    - array 指针字段的值变了，不再指向原切片的底层数组了，会指向一块新的内存空间
    - len 长度字段的值做相应增加，增加了 N 个元素，长度就增加 N
    - cap 容量会增加

**slice 扩容机制**

cap 等于原切片的长度 +append 新增的元素个数

**原切片和目标切片的内存空间可能会有重合，copy 后可能会改变原切片的值，参考下例。**

```
package main

import "fmt"

func main() {
	a := []int{1, 2, 3}
	b := a[1:] // [2 3]
	copy(a, b) // a 和 b 内存空间有重叠
	fmt.Println(a, b) // [2 3 3] [3 3]
}
```

slice：https://mp.weixin.qq.com/s?__biz=Mzg2MTcwNjc1Mg==&mid=2247483741&idx=1&sn=486066a3a582faf457f91b8397178f64&chksm=ce124e32f965c72411e2f083c22531aa70bb7fa0946c505dc886fb054b2a644abde3ad7ea6a0&token=1073108956&lang=zh_CN#rd

func makechan(t *chantype, size int) *hchan

从上面的代码可以看出 makechan 返回的是指向 channel 的指针。
因此 channel 作为函数参数时，实参 channel 和形参 channel 都指向同一个 channel 结构体的内存空间，所以在函数内部对 channel 形参的修改对外部 channel 实参是可见的，反之亦然。

```
package main
```

```
import "fmt"

func main() {
	data := make(chan int)
	shutdown := make(chan int)
	close(shutdown)
	close(data)

	select {
	case <-shutdown:
		fmt.Print("CLOSED, ")
	case data <- 1:
		fmt.Print("HAS WRITTEN, ")
	default:
		fmt.Print("DEFAULT, ")
	}
}
```

1. select的运行机制如下：
    - 选取一个可执行不阻塞的case分支，如果多个case分支都不阻塞，会随机选一个case分支执行，和case分支在代码里写的顺序没关系。
    - 如果所有case分支都阻塞，会进入default分支执行。
    - 如果没有default分支，那select会阻塞，直到有一个case分支不阻塞。

- 对于关闭的channel，从channel里接收数据，拿到的是channel的存储的元素类型的零值，因此case <-shutdown这个case分支不会阻塞。
- 对于关闭的channel，向其发送数据会引发panic，因此case data <- 1这个case分支不会阻塞，会引发panic。
- 因此这个select语句执行的时候，2个case分支都不会阻塞，都可能执行到。如果执行的是case <-shutdown这个case分支，会打印"CLOSED, "。如果执行的是case data <- 1这个case分支，会导致程序panic。
-

实际开发过程中，Context的使用流程一般是：

- Step 1: 创建Context，给Context指定超时时间，设置取消信号，或者附加参数(链路跟踪里经常使用Context里的附加参数，传递IP等链路跟踪信息)。
- Step 2: goroutine使用Step 1里的Context作为第一个参数，在该goroutine里就可以做如下事情：
    - 使用Context里的Done函数判断是否达到了Context设置的超时时间或者Context是否被主动取消了。

- 使用 Context 里的 Value 函数获取该 Context 里的附加参数信息。
- 使用 Context 里的 Err 函数获取错误原因，目前原因就 2 个，要么是超时，要么是主动取消。

context.WithTimeout 函数和 context.WithDeadline 函数可以创建一个有超时时间的 Context。通过 Context 的 Done 函数可以判断是否超时了。

ctx, cancel := context.WithTimeout(parent, 100 * time.Millisecond)
response, err := grpcClient.Send(ctx, request)

拷贝数组

Good

```go
// SetTrips sets the driver's trips.
// Note that the slice is captured by reference, the
// caller should take care of prevening unwanted aliasing.
func (d *Driver) SetTrips(trips []Trip) { d.trips = trips }

// or:

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = make([]Trip, len(trips))
  copy(d.trips, trips)
}

trips := ...
d1.SetTrips(trips)

// We can now modify trips[0] without affecting d1.trips.
trips[0] = ...
```

Bad

```go
type Stats struct {
  sync.Mutex

  counters map[string]int
}

// Snapshot returns the current stats.
func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  return s.counters
}

// snapshot is no longer protected by the lock!
snapshot := stats.Snapshot()
```

Good

```go
type Stats struct {
  sync.Mutex

  counters map[string]int
}

func (s *Stats) Snapshot() map[string]int {
  s.Lock()
  defer s.Unlock()

  result := make(map[string]int, len(s.counters))
  for k, v := range s.counters {
    result[k] = v
  }
  return result
}

// Snapshot is now a copy.
snapshot := stats.Snapshot()
```

枚举

Import Aliasing

```
import (
  "net/http"

  client "example.com/client-go"
  trace "example.com/trace/v2"
)
```

wantError := `unknown error:"test"`

Go supports raw string literals, which can span multiple lines and include quotes. Use these to avoid hand-escaped strings which are much harder to read.

| Bad | Good |
| --- | --- |
| `wantError := "unknown name:\"test\""` | `wantError := `unknown error:"test"`` |

Initializing Struct References