Table of Contents

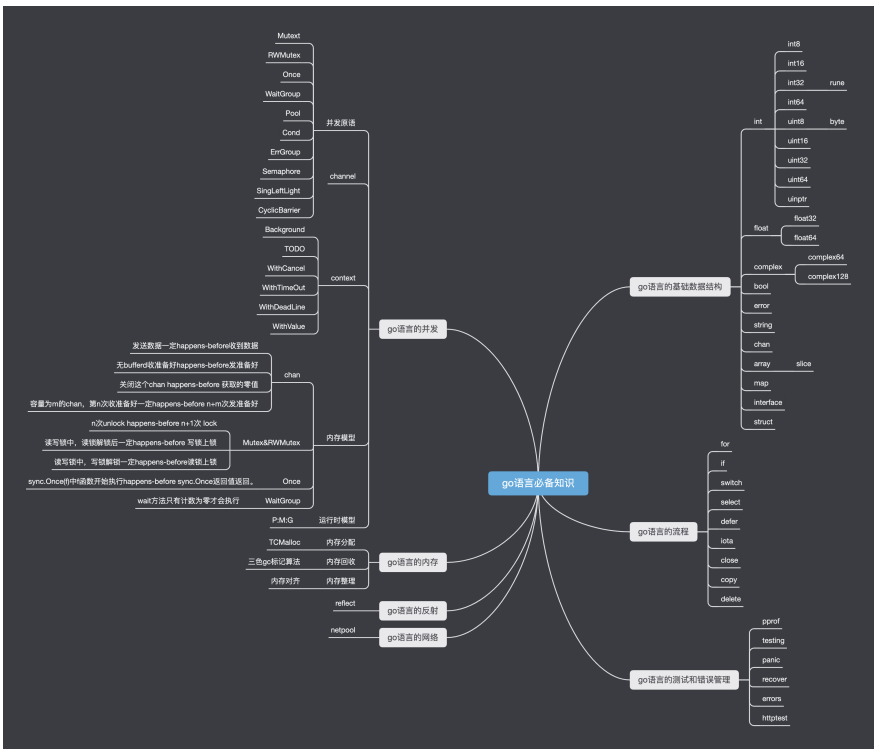
GOFamily

go语言从入门到熟练

作者@科科人神

导图

> 国内访问可能无法访问到图片，可以添加微信公众号：`科科人神`，回复：`go思维导图`获取，另外回复`go-book`可获得本教程。



- go语言的基本语法
  - 介绍go语言
  - go语言的包管理工具
  - go语言的基础组件
  - go语言常用关键字
  - go语言的条件和逻辑语句
  - go语言的面向对象
  - go语言的错误处理
  - go语言的测试
- go语言高级用法
  - 同步原语和锁
  - context
  - channel
  - 定时器
  - 运行时调度器P:M:G
  - 网络轮询器netpool

    - 系统监控
    - go语言的内存模型
    - go语言的并发模型
    - 内存分配
    - 内存回收gc
    - 栈内存管理
    - go语言的动态调试
    - go语言的性能优化
    - cgo，unsafe，reflect非常见的操作能力
    - go语言的反射
    - go语言线上事故排查
- go语言的标准库用法
    - net
    - http
    - rpc
    - time
    - io
    - bufio
    - container
    - sql
    - crypto
    - encoding
    - flag
    - fmt
    - os
    - syscall
    - text
    - strconv
    - sort
    - strings
    - atom
    - log
    - json

关注我

- 头条@科科人神

- 微信公众号@科科人神

- b站@科科人神

作者的计算机知识项目

- foolon : 从零开始学习分布式系统
- GOFamily :go语言从入门到熟练掌握
- 408Family : 算法和数据结构,设计模式，正则表达式，网络，操作系统，计算机组成原理
- DBFamily : 从零开始学习数据库知识
- study-k8s-docker : 从零开始学习k8s和docker等容器化的知识
- hi-test :从零开始学习关于软件测试，性能测试，软件工程的知识
- up : 从初级程序员向高级程序员迈进 --- 系统调优公开课
- repairman :程序员的必修课 --- 线上紧急故障排除
- Refactor : 高级软件工程师的必修课 -- 如何进行代码重构
- coder : 关注程序员的职业发展，会分享比如如何升迁，如何面试，如何赚钱等知识。

go语言介绍

任何语言讲解都是从hello world开始的，下面这个例子就是go语言的hello world。

```
package main

import(
    "fmt"
)

func main(){
    fmt.Println("hello world")
}
```

从这个例子可以看到首先，每一个go文件都需要一个包名，一个包内部可以有若干个文件，比如说一个叫做app的包，里面的文件可以有 `a1.go a2.go a3.go` ，每一个文件的顶部都必须写上 `package app` 。

往下是一个 `import()` ,括号里面是包的名称，这里的fmt是标准库中的包。这个包的作用就是标准输出和标准输入。

最后我们可以看到下面有一个叫做main的函数，在go里面没有“类”这种东西，最大的官儿就是函数，每一个可执行的go包，都必须拥有一个main函数，这个函数是所有函数的入口函数。

看到这里你应该对于go语言有了一个初步的认识，对了你应该发现了吧，go语言里面是没有传统语言里面的分号的，这里要说明一下，go语言严禁分号，这跟js不同，js中可以有也可以没有，c里面是必须有，但是go里面是必须没有。

go语言是起源于美国谷歌的一个服务器语言，目前已经开源，创始人是来自前贝尔实验室的三位大佬，go语言的最大的特点就是将并发，将计算机的多核心利用，融入到了语言层级，我想这也是go语言在云计算以及高并发这些场景受到追捧的原因。

下面介绍以下跟go语言相关的一些网站和工具，各位可以自行下载或者查看：

- https://golang.org ; https://golang.google.cn go语言的官方网站，后面的可国内访问。
- https://awesome-go.com 这是awesome系列，里面是排的上号儿的优秀的go项目。
- goland，jetbrans 推出的go IDE。

go语言包管理工具 go mod 详细介绍

通常来说我们创建某个项目都是直接使用IDE来进行创建，当然了使用vs code这种文本编辑器也可以，所以说使用这些工具来创建一个go的开发环境，是几乎不需要设置的，因为ide已经帮你设置好了，我们就拿goland来说，当我们选择新建一个go项目的时候，go.mod和go.sum都会自动创建完成，说到这里我们就要提一下这两个文件了。

`go.mod`：

通常你会见到这样的格式

```
module github.com/shgopher/short

go 1.16

require(

github.com/gin-contrib/sse v0.0.0-20190124093953-61b50c2ef482
github.com/golang/protobuf v1.2.0

)
```

> go.sum是系统自动管理的文件，用户禁止直接操作，就不谈了。作用就是记录使用的每一个包以及它的版本。

首先呢，每一个go.mod对应了一个项目，每个项目里都有一个go.mod，最上面是一个module字符，后面加上你的包名称，我这里的包名称就是 `github.com/shgopher/short` 这一堆东西就是这个包的路径，注意是路径并不是包的名称，下文会详细说。

下面的 `go1.16` 代表了本项目使用的go的版本。

下面的两行都是指的这个项目中使用到的外部包（除了标准库以外的包），前面是名称，后面是它的版本号，上面这种形式，就是所谓的发布的时候没有采用git version的方式进行发布，所以go自动将发布时候的时间作为了它的版本号，下面这个项目，由于发布的时候使用了 git version，拥有了一个版本号，所以我们使用的时候就直接使用 `v1.2.0` 这个版本号即可。

这里要介绍一下包的命名问题，为什么我们的包名称里面要有github呢？原因也很简单，你没有这个也可以，没有规定一定要有，只是go使用git来管理包的版本，然后呢，大多数的程序员使用GitHub来管理自己的代码，所以说发布到GitHub上以后的包被go引用就是GitHub开头了，举个例子，假设我的仓库在GitHub上，地址是， `github.com/shgopher/short` ，我们在项目中使用这个包（通常在项目中这个包就是被命名为“github.com/shgopher/short”）然后go就会自动下载这个包，使用的原理就是 `git clone https://github.com/shgopher/short` 也就是说go把 `https://` 给你省略了,当然ide是可以自动帮你下载包的。

那么如果ide反应迟钝，或则是因为网络问题，我们该如何下载这个包到本地呢？

通常来说，我们将这个包写入了 `import()` 里面，然后我们在命令行里，输入 `go mod tidy` go就可以自动将远程包下载到本地了。

说到这里，我们还得注意一件事，go是存在子包的，举个例子：

比如这个项目叫做 `github.com/shgopher/short` ，那么这个项目中存在若干子包，比如可以叫做 `github.com/shgopher/short/fast` 这个就是short的子包，通常就是一个子文件夹，里面是另一个包名即可。

我们要注意一点， `github.com/shgopher/short` ，只有最后这个单词才是包的正式名称，这一个整串儿其实只是路径而已，就比方这个项目，package一定是 `short` 而不是那一串儿，这个时候又来了一个问题，那么这个包的名字跟这个路径的名字不一样行不？答案是可以的，比如说这个包的名称是short，那么这个路径完全可以命名为 `github.com/shgopher/babalala` 但是通常我们都是保持一致的，所以说判断一个包的名称具体是啥，不是看module后面的路径，而是看包内部package后面的名称。

关于子包的使用，我们可以给定一个场景，比如说我们的项目，和要使用的外部包都被托管于GitHub，那么我们的包叫做 example，路径是 `github.com/shgopher/example` 那么我们要使用的外部包有short包和short/fast包，那么我们该如何使用呢?

```
package example

import(
    "github.com/shgopher/short"
    "github.com/shgopher/short/fast"
)
```

如何自己的项目中拥有子包，大包要调用子包该如何调用呢？其实这个场景也很常见，因为你搞子包的很大意义就是它自己的功能性自成一体，然后大包要使用这个功能，其实这个也很简单，直接调用即可。

假设这个子包叫做fast

```
package example

import(
    "github.com/shgopher/example/fast"
)
```

大的版本升级，go也是不一样的，通常按照规则，一个项目还比如short吧，它原本的仓库是 `module github.com/shgopher/short` ，如果版本已经上升到了大于等于2的地步，go的推荐是包的命名要改为 `module github.com/shgopher/short/v2` 但是包的名称还是short，并不是v2，也就是说package这里还是short，只是在module这里改成了这种写法，那么别的包使用这个包的时候该如何使用呢

```
import(
"github.com/shgopher/short/v2"
)
```

记得版本一定是v+数字，例如 `v2，v3，v4，v5` 。 在下面使用的时候仍然是用的 `short.xxx` 感觉有点麻烦对吧，就是这么用的，记得就行。

如果你的包引入的外部包不够好，你想重构怎么办？这个时候就靠replace命令出手了

go.mod:

```
module github.com/shgopher/short

replace(

    github.com/shgoper/i v0.1.0 => github.com/shgopher/newShort v0.3.0
)
```

这里就是说我们项目中原本使用的是v0.1.0这个版本的i包，但是我们不想改代码，我们实际上使用的v0.3.0 版本的newshort包。

到这里了，我们再回忆一下重点知识，go的每个项目其实就是以包为单位的，包可以拥有子包，只有main包可以拥有main函数，以及只有main函数才可以执行，才能直接生成可执行文件。module后面跟的其实只是包的路径而已，通常我们命名为 `github.com/xxx` 是因为包部署到github,如果你把包放在了gitlab上，你可以改成 `gitlab/xxx` 也是没有问题的。

最后我把项目中演示的图片放到最后，不过由于网络问题你不一定能看得到。

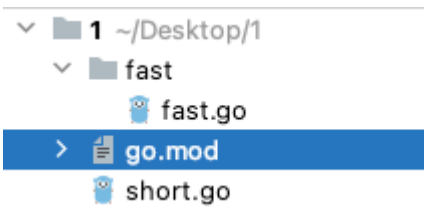
```
module github.com/shgopher/short

go 1.16

require github.com/googege/gotools v0.3.0

replace github.com/googege/gotools v0.3.0 => github.com/shgopher/gotools v0.3.0

```




```
package short

import (
    "fmt"
    "github.com/googege/gotools"
    "github.com/shgopher/short/fast"
)

func Short(){
    fmt.Println( a...: "short")
    fast.Fast()
    gotools.Read( root: "")
}
```

```go
package fast

import "fmt"

func Fast(){
    fmt.Println( a...: "fast")
}
```

go语言的基础组件

go语言自带的基础类型包括

- int ： 有符号的整数类型，具体占几个字节要看操作系统的分配，不过至少分配给32位。
- uint：非负整数类型，具体占几个字节要看操作系统的分配，不过至少分配给32位。
- int8：有符号的整数类型，占8位bit，1个字节。范围从负的2的8次方到正的2的8次方减1。
- int16：有符号的整数类型，占16位bit，2个字节。范围从负的2的16次方到正的2的16次方减1。
- int32：有符号的整数类型，占32位bit，4个字节。范围从负的2的32次方到正的2的32次方减1。
- int64：有符号的整数类型，占64位bit，8个字节。范围从负的2的64次方到正的2的64次方减1。
- uint8:无符号的正整数类型，占8位，从0到2的9次方减1.也就是0到255.
- uint16：无符号的正整数类型，占16位，从0到2的8次方减1.
- uint32：无符号的正整数类型，占32位，从0到2的32次方减1.
- uint64：无符号的正整数类型，占64位，从0到2的64次方减1.
- uintptr：无符号的储存指针位置的类型。也就是所谓的地址类型。
- rune ： 等于int32，这里是经常指文字符。
- byte：等于uint8，这里专门指字节符
- string：字符串，通常是一个切片类型，数组内部使用rune
- float32：浮点型，包括正负小数，IEEE-754 32位的集合
- float64：浮点型，包括正负小数，IEEE-754 64位的集合
- complex64，复数，实部和虚部是float32
- complex128，复数，实部和虚部都是float64
- error，错误类型,真实的类型是一个接口。
- bool，布尔类型

> int8 和uint8 后面的8指的是占得位数，因为有符号的第一位作为符号位置，所以它真的可以计数的位置只有7个了，第一位表示正负，无符号的整数显然没有这个烦恼。

go语言的基础组件分为以下几种：

其中按照是否是引用类型（指针类型）分为引用类型和非引用类型，他们分别是

引用类型	非引用类型
slice，interface，chan，map	array，func，struct，大部分的内置类型

这里要说明以下，所有的非引用类型的初始化值都是一个具体的值，只有引用类型的初始化是`nil`，nil在go里面就是指的是空。是不能直接使用的。

全局变量，引用类型的分配在堆上，值类型的分配在栈上。

局部变量，一般分配在栈上。如果局部变量太大，则分配在堆上。如果函数执行完，仍然有外部引用此局部变量，则分配在堆上。

我们分别来介绍以下他们。

array

数组，我们先看一下数组的初始化。

```
// 给数组进行初始化

// 初始化的方式1
a := [6]string{}

// 初始化的方式2
var a [6]string
```

这里稍微提一下，在go里面的赋值符号有两种：

```
var a

b :=
```

其中var 这种方式不论是局部还是全局变量都可以使用，但是后者也就是 `:=` 只有局部变量可以使用。也就是只有函数内部才能使用。

并且，var后面的变量后面的类型是可以省略的，省略后，go会在编译过程中自动判断。所以如果不省略就是长这样.

```
var a int
```

说到了变量，go里面当然也有定量，使用const来命名，一般都是全局使用。

```
// 根据习惯一般用大写表示定量
const PAI = 12
```

同样的，定量也可以省略后面的类型。

接下来我们看一下数组的赋值

```
a[0] = "0"
a[1] = "1"
a[2] = "2"
a[3] = "3"
a[4] = "4"
a[5] = "5"
`
```

这里要点明一下，值类型，或则说非引用型类型的“声明” 就等于初始化，也就是说，当你给一个变量声明一个值类型的数据时，就自动给定了初始值。

这里谈一下他们的初始值：

array	int	string	bool	float	func	struct
空数组，但不是nil，已经占用了声明的数组长度	0	空字符串，通常我们使用""代指空字符串	false	0	就是一个空的函数	一个空的structure

length <4 的数组，数据是直接存在栈空间上的，如果数据大于4，那么会存放在静态空间，然后才复制到栈空间上，顺便说一下，堆和栈属于动态存储，其中栈是操作系统直接控制，我们的局部变量，不逃逸的情况下都是存在于栈中，逃逸了就去堆里面了，说完了动态，静态存储区域包含了两者，一个是存储的常量，一个是存储的静态变量，常量好理解就是const来声明的常量，静态变量，比如这里大于4的数组。

数组的语法糖 `[...]int` 使用这种方式，必须在声明的时候直接赋值，否则下面进行赋值的时候，系统不知道你的length到底是多少，就会"out of index"

关于go里面的比较问题，只有类型一致的情况下，进行比较，比如struct int,等，接口也可以比较，slice map 以及函数体，都是无法进行比较的。chan 可以比较，但是即使是类型一样，也是false的结果，nil也是可以比较的，因为nil的底层是 `var nil Type` `type Type int` 也就是说nil其实是一个值类型。nil也是有类型的，比如说接口的nil就是接口类型，那么指针类型的nil就是指针类型，slice的nil就是这个slice类型。

数组的初始化的中括号里要么是 `...`，要么就是个常量，不能是变量

slice

切片，是一个内置的引用类型，其实质是一个structure，也就是说是一个结构体，这个结构体内部含有一个指向某个数组的地址，所以说我们可以简单的来理解，slice是某个数组的指针。

```
type SliceHeader struct {
    // 指向底层数组的指针类型
    data uintptr
    // 长度
    len int
    // 容量
    cap int
}
```

切片的初始化：

```
a := make([]string,10)

// 或者

var b [10]string
a := &b

// 或者
a := []string{1,2,3,}
// 或者
a := [3]int{1,2,3,}
// 左闭右开
b := a[0:2]
x := b[0:1]
// 等于c取了数组的全部数据
c := a[:]
```

这里涉及了几个知识点，首先是make()函数，这个函数是go的内置函数，意义就是为了给引用类型初始化，所以能使用make的就是这些引用类型，slice map chan，interface不可以使用，不好意思它make和new都不能使用。

new的含义就是说从一个值类型上取得它的地址，跟 `&` 相似，后者是取地址的符号。并且new的括号里只能是Type类型。 `例如：type A map[string]string`

切片会重新指向新的数组，比如当leng不够需要扩展，然后cap也不够的时候，就会创建一个新的数组底层，然后进行数据的迁移。

切片使用初始化的方式会在编译期间就完成了初始化，但是当使用make关键字创建的时候就会在运行时初始化，在运行时初始化会对速度造成影响。

当切片非常大，或者切片逃逸了，那么就会在堆上创建这个切片。

切片的扩容【考点】：

```
func growslice(et *_type, old slice, cap int) slice {
    newcap := old.cap
    doublecap := newcap + newcap
    if cap > doublecap {
        newcap = cap
    } else {
        if old.len < 1024 {
            newcap = doublecap
        } else {
            for 0 < newcap && newcap < cap {
                newcap += newcap / 4
            }
            if newcap <= 0 {
                newcap = cap
            }
        }
    }
```

给定一个期望扩容数字：

- 如果期望大于两倍的老容量，那么新的容量就是这个期待容量
- 如果期望小于两倍的老容量，并且老的容量个数小于1024，那么新的容量就是老容量的二倍
- 如果期望小于两倍的老容量，并且老的容量个数大于1024，那么这个容量就按照之前老容量的1.25倍开始增加，直到大于了期望容量，开始跳出循环

- 如果老容量是小于等于0的，那么新的容量直接等于期望容量

当然这只是初步确定容量，下面还要进行内容的对齐。

切片的数据拷贝：`copy(newSlice,oldSlice)`，这里直接是值的拷贝。

map

map，也就是所谓的hash map 哈希表，散列表，在go里面的哈希表，使用的避免哈希碰撞的算法是链表法。map的key必须是Type,并且是可以比较的类型，例如int，string，interface{},bool,一般接口类型，不过不可比较的例如 slice map都无法充当key值。

我们来看一下map的初始化

```
a := make(map[string]string)
```

make()后面其实是有三个值的，第一个就是类型，第二个是length长度，第三个是cap 容量，你先了解一下，slice是需要指定length值的，也就是第二个值，但是第三个并不需要指定，map更厉害，它只需要提供类型，后面两个都不需要提供。

> 禁止使用 var a map[string]string来初始化，这种只能是声明一个类型，因为声明后a变量的初始值是nil值，也就是说没有分配内存。

map的使用：

```
a["0"] = "0"
a["1"] = "1"
a["2"] = "2"
// 删除key
delete(map,key)

// ok表示是否含有这个值。
value,ok := a["1"]
```

说到length和cap，我们提一下len()和cap()函数。

这俩函数前者是可以测定array，slice，map的长度，后者是容量，说到长度和容量，听起来很相似，到底啥区别呢?

我们来举个例子：

上文我们谈到，slice的底层是一个数组，那么数组的整个的长度就是这个切片的容量，我们取前三个作为length，那么就是3就是这个slice的长度，所以长度<= 容量，再谈一个点，在go里面的所有关于slice是否 `out of index` 也就是说是否超过下标，指的都是长度而不是容量。

```
func main(){
a := make([]string,10,20)
a[10] ="1"
}
```

```
panic: runtime error: index out of range [10] with length 10
```

在map中，有两个数据非常关键，一个是hash function，一个是hash冲突

其中hash函数，分为两种，一种是加密型hash函数，例如rc4，sha，等，一种是非加密型函数，这种函数就是为了检索而生，例如murmur hash 函数。

hash冲突，简单的来说就是两个key使用hash函数计算出来的hash值是一样的，无法去定位真实的value值，那么我们有两种解决的方法，一种是向后寻找法，一种是链表法。

> 值得注意的是，这里的hash碰撞还不一定就是计算出来的hash值是完全一样，有可能是前某几位是一致的，因为我们取的可能是前几位，不可能取完，

向后寻找意思很简单，我们不论是查找的时候，或者是写的时候都是加入计算出的key，已经有人占据了，那么我们的数据就不放在这个坑了，我们往后找，看是不是有空缺的，如果有就放进去这个key-value数据。

这种方法特别要注意的是装载因子，因为hash表底层存储结构是数组，那么如果数组中的数/长度【这就是装载因子的意义】太大，那么向后寻找法就会很难找到数据，直至时间复杂度变成O(N)

链表法就是key计算出来的hash值一样的放在一个地方，只不过这个地方改成一个链表即可。然后我们查找的时候查找这个链表，看真正要找的key是具体哪个，然后我们取得这个key对应的value即可。这种方法要注意链表过长，过长的意思就是hash函数不行，造成分布不够均匀。或者函数生成的值过于狭窄。

map扩容的两个条件是，一是桶的装载因子超过6.5，二是当溢出桶中的元素数量过多的时候，也需要进行扩容了，如果不扩容就会降低map的性能，其中为了扩容时候的效率，在创建新的数据结构后，因为桶的数量改变了，会重新进行hash计算，并且把旧桶中数据进行分流到新的桶中。

在扩容期间访问哈希表时会使用旧桶，向哈希表写入数据时会触发旧桶元素的分流。

哈希表的每个桶都只能存储 **8**个键值对，一旦当前哈希的某个桶超出 8 个，新的键值对就会存储到哈希的**溢出桶**中。随着键值对数量的增加，溢出桶的数量和哈希的装载因子也会逐渐升高，超过一定范围就会触发扩容，扩容会将桶的数量翻倍，元素再分配的过程也是在**调用写操作时增量进行的**，不会造成性能的瞬时巨大抖动。

func

函数，以及后文要谈的方法，都是这种形态。

```
//1
func main(){

}
//2
func fast(a string,b int)(float64,func (int,string)){

    return 0,func()(int,string){
        return 1, ""
    }
}
//3
func heigh(a int)(b string){
    return ""
}
//4
var apple = func()(string){}
```

上面的函数显示了一个函数的基本样子，func关键字在最前面，后面是函数名称,函数后面的括号里是两个临时变量，再后面是返回的值，注意，如果返回的值只有一个，可以省略括号，并且go可以直接返回多值。

函数的使用：

```
fast("",0)
```

调用的时候还是比较容易的。

struct

结构体，我们来看一下基本的形态

```
type A struct {
    v1 ,v2 int
    v3 bool

}
使用`type`这个符号，加上变量名称，再加上一个struct，然后结构体内部的变量的声明就如同文中所示

我们看一下结构体的使用

```go
// 这是声明，声明直接初始化。
var a A

/*
下面是赋值
*/

a.v1 = 1
a.v2 = 2
a.v3= true

// 另一种赋值方式

var a = A {
    v1:1,
    v2:1,
    v3:true, // 这里注意一下，逗号一定要有，结尾处有逗号
}

// 也可以省略前面的key，但是这样必须按照顺序
var a = A{
    1,3,true,
}

// 如果想取结构体的地址也是有两种方式的
var a = &A{
    //xxx
}

// 另一种
var a = new(A)
```

interface

go语言的接口，核心就是鸭子🦆 理论，也就是说不像传统语言，java那种必须显示声明出来接口的调用，go语言中只需要实现接口的方法就是实现了这个接口。

声明一个接口

```
type people interface {
    see()
    eat()
}
```

内部的是函数，接口内部只接函数。
```

在此提示一下 接口与众不同，不可使用make和new来声明获取一个接口，接口并没有实际的任何意义，所以它没有任何的底层指向，使用make是没有意义的。同时因为interface的实质也是一个structure 只是内部是记录的一些参数，所以说取这个接口的地址也没有任何的意义，所以go里面不要取接口的地址。

如何实现这个接口?

```
func c (p people){
    p.see()
    p.eat()
}
```

这里，这个函数的变量是接口类型，那么我们要传入的也应该是people这个接口类型，那么什么是符合这个接口类型呢?

等下文的面向对象再详细说一下。

chan

chan，这里先简单介绍一下，后文go并发编程可以详细介绍一下。

因为chan也是引用类型，所以它也必须使用make才可以初始化

```
c := make(chan string,2)
```

chan 后面要加上具体的类型，然后再加上长度即可。

这里你先简单的了解一下chan，中文叫做通道，你可以简单的和unix中的通道类比一下，后面的长度就是指的，通道内可以缓存的数据量，当然这里你把通道当作一个队列。

使用的时候可以这样做。

```
func fast(){
    c := make(chan string,2)
    go b(c)
}

func b(chan string){}
```

这里的go 关键字指的是开辟了一个新的goroutine，然后通道在不同的goroutine中流传传递信息。

其中，有两种特殊的命名方式，就是只读通道和只写通道。

```
// 只读通道
var a <- chan string
// 只写通道
var b chan <- string
```

往通道里读写数据这什么来操作的:

```
// 写入数据
a <- ""

// 读取数据
<- a
```

当然我们一般不会舍弃读取的通道数据，会将数据赋值给一个变量

```
c := <- a
```

iota

特殊常量，可以自增。切记，只能用在常量中。

```
const (
    a =   1+ iota
    b
    c
    d
    e
    f = "12"
    g= iota
    h
    i
)
```

`1 2 3 4 5 12 6 7 8` iota在常量中处于自增的方式，可以看到，iota的初始值是0，所以a等于1，b就是2，b等于1+1，当遇到f的时候，iota自己的自增没有变化，但是f就变了，变成了“12”，然后g又给了iota，那么这个时候的iota就不是从零开始，iota的就是6，意思就是往下数嘛，从0开始，到g就是6了。又因为这个变量的赋值算式不是1+iota，是iota了，那么后面的就变成了直接将iota赋值给变量了。

字符串

go语言中的字符串是一个 `只读字节切片` ，底层数组里存放的是byte类型，如果我们想改变这个字符串，可以将 `字符串 <=> []byte` 互相的转换，进而改变这个字符串，这个转化的过程，先将静态存储区的字符串转到栈或者堆中（数组较大就会转化到堆上）然后string转变为字节数组，更改数组中元素，再转为字符串即可。因为字符串作为只读的类型，我们并不会直接向字符串直接追加元素改变其本身的内存空间，所有在字符串上的写入操作都是通过拷贝实现的。

在两者进行数据转化的时候，是有性能的损失的，所以优化性能的话，可以选择减少字符串和[]byte之间的转换。

```
func main() {
    a := "github.com/shgopher"
    b := []byte(a)
    b[0]= 12
    fmt.Println(string(b))
}
```

go语言常见的关键字

本文章会包含以下内容：

- for和range
- select
- defer
- panic和recover
- make和new

for 和 range

slice

```
func main() {
    a:= []int{1,2,3}
    var ma []*int
    for _,v := range a{
        v+=1
        ma = append(ma,&v)
    }
    fmt.Println(a)
    for i := 0;i < 3;i++ {
        fmt.Println(*ma[i])
    }
}
```

```
[1,2,3]

4,4,4
```

前面a为什么v+1了输出的还是1，2，3呢?

原因是因为这里的v只是切片数据的值复制，所以v的改变不会改变原切片的内容，如果要改变可以更改为

```
for i:= 0;i<len(a);i++ {
    a[i]++
}
```

而遇到这种同时遍历索引和元素的 range 循环时，Go 语言会额外创建一个新的 v2 变量存储切片中的元素，循环中使用的这个变量 v2 会在每一次迭代被重新赋值而覆盖，赋值时也会触发拷贝。

因为在循环中获取返回变量的地址都完全相同，所以会发生神奇的指针一节中的现象。因此当我们想要访问数组中元素所在的地址时，不应该直接获取 range 返回的变量地址 &v2，而应该使用 &a[index] 这种形式。

```
for k,v := range a{
        v+=1
        ma = append(ma,&a[k])
    }
```

那么这种方式为啥就是不一样了呢?

因为经过了复制，也就是说这里的&a[k]每次循环都不是一个&a[k],每次的a[k]都是一次运算，意思就是已经取到值了。

对于所有的 range 循环，Go 语言都会在编译期将原切片或者数组赋值给一个新变量 ha，在赋值的过程中就发生了拷贝。所以这里的range过程中的length就是老的length，新的切片跟这个循环次数一点关系都没有。

map

在map中的遍历是随机的，原理就是寻找桶的时候是随机的，然后我们从某一个桶开始找，找到非零桶的时候把链表里的数据查询出来，一般这个时候是查询出来的数据的地址，当然在扩容期间，就是找到真的k-v，然后进而找到所有的正常桶，然后再寻找溢出桶。这个时候遍历就结束了。

string

当遍历字符串的时候，跟遍历slice差不多，只是会把byte（uint8）类型转化为rune（int32）

chan

range 可以变量chan中的数据，不过如果chan未关闭，但是没有数据了，range会Panic，所以如果没有数据了，chan需要关闭才行。如果不关闭就会阻塞，也因为不关闭，就意味着只有写，才能读，但是不写了，读也读不了，如果读一个关闭的chan，那么只会得到零值而已。

```
func main() {
    a := make(chan int)
    wg := new(sync.WaitGroup)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            a <- i
        }(i)

    }
    go func() {
        wg.Wait()
        close(a)
    }()
    //这里，当range读的通道被close后，就会【自动！】跳出循环。
    for v := range a {
        fmt.Println(v)
    }

}
```

select

select跟io复用中的select比较相似，在select中的case中，只允许出现chan的读或者是写，当然还有default，形如

```
ffunc main() {
    a := make(chan  int)
    wg := new(sync.WaitGroup)
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            a <- i

        }(i)

    }
    go func() {
        wg.Wait()
        close(a)
    }()
    L:
    for  {
        select {
            // 使用这种形式判断通道是否close
        case v,ok := <- a:
            if ok  {
                fmt.Println(v)
            }else {
                break L
            }
        }
    }

}
```

但是请注意，如果两个case都满足，那么执行的时候会随机执行一个。并不会同时执行。随机的目的就是避免饥饿。

select存在这么几种机制

1. 没有case

   这种情况下，select是永远的阻塞状态。

2. 只有一个case

   当case中的chan是nil时，陷入阻塞状态

3. 只有一个case和default

4. 含有多个case

defer

defer执行的位置是在return之前进行执行，并且执行的顺序是栈的顺序。并且defer的后面只能跟函数。

通常来说我们使用defer都会为了关闭某个操作，比如response.body,就需要我们手动关闭，比如数据库的连接，这种操作下，使用defer再合适不过了，并且再在最新的go版本中，defer的性能损耗大大降低 6ns左右，可以说忽略不计了。

```
func createPost(db *gorm.DB) error {
    tx := db.Begin()
    defer tx.Rollback()

    if err := tx.Create(&Post{Author: "Draveness"}).Error; err != nil {
        return err
    }

    return tx.Commit().Error
}
```

这里要注意一件事，tx.Commit().Error先执行，然后这个时候发现有defer，不会立即执行return，而是执行defer，执行完毕后，再把tx.Commit().Error的结果返回出来。也就是说return其实不是一次执行完毕的事情。

这里要注意两个非常重要的事情

- defer执行的顺序
- defer预计算的顺序

我们直到defer后面跟函数，然后后面执行的顺序是栈，也就是defer栈，按照先入后出（后执行）的顺序进行执行，但是有一点要注意，虽然defer是在整个大部队后面执行的，并且是栈顺序，然而，它的预操作是按照整个函数的顺序执行的,defer关键字会立刻拷贝函数中引用的外部参数.

```
func test(i int) int{
    // 这里打印的就是i 而不是i++后的结果，就是因为defer 把外部参数复制了。
    defer fmt.Println(i)
    i++
    return i
}
```

要改变这种方式也很简单，不要外部参数即可。

```
func test(i int) int{

    defer func(){
        fmt.Println(i)
    }()
    i++
    return i
}
```

我们再看两个例子

输入1

```
func test1(i int)(ii int){
    defer func() {
        ii++
    }()
    ii = 1+i
    return
}
// output 3
-----
func test2(i int)int{
    defer func() {
        i++
    }()
    i++
    return i
}
// output:2
```

这就是return的区别了，前面那种执行的过程是，return的结果在defer后面执行，后面那种是return的结果在defer前面就复制了，然后再执行defer，然后return结果，这个结果不受到defer的影响。

实际执行的是等于

```
func test2(i int)int{
    defer func() {
        i++
    }()
    i++
    //
    x: = i
    //
    return x // x的值不受到i的影响
}
```

再看一下让人更震惊的：

```
func test1(i int)(ii int){
    defer func() {
        ii++
    }()
    ii = 1+i
    return 1 // 这里等于 ii = 1
}
// 结果不是1，而是2 ，因为这里的return后面的值还是ii，ii被重置为1，然后再++ ，然后再返回。
```

```
func test1(i int)int{
    defer func() {
        i++
        fmt.Println(i)
    }()
    i++
    return 5
}

// 返回5，打印3 ，证明return还是等到defer执行完再返回，不过return后面的变量是新变量了。
```

也就是说，return后面的结果就是ii，这种情况就是ii被1+i重置为了1，然后还是要再执行defer，得到的最终的值，才是返回值。

defer很容易出问题，所以这种技巧尽量不要采用。defer老老实实的使用的场景就是关闭某某。

我们来总结以下：

- 如果是带值返回的那种，返回的就是那个显式的变量
- 如果不带值的那种，返回的其实是return后面那个结果的复制值。所以这个之前的参数再怎么在defer里反应，即便return等到defer执行完再返回，也不会改变这个复制值。

panic和recover

panic 就是恐慌，使得程序在这个goroutine中立刻停下，当然整个程序也会停止，并且其它的在这个goroutine后面的任何代码都不会执行了(不过Panic不会影响defer函数的执行，但是只是这个goroutine中的defer，其它goroutine中如果没有执行到就不会执行了)recover就是恢复或者忽略Panic，并且recover函数，必须在defer函数中使用。而且在一个goroutine中的潜逃的defer也不会收到影响

```
func main() {
    defer fmt.Println("in main")
    defer func() {
        defer func() {
            panic("panic again and again")
        }()
        panic("panic again")
    }()

    panic("panic once")
}
// defer 都可以正常执行
```

正确的recover的执行

```
func main(){
    defer func(){
        if err := recover();err != nil {
            fmt.Println("panic:",err)
        }
    }()
    get()
}

func get(){
    panic("get is panic.")
}
```

注意一下

```
func main() {
    defer recover()
    get()
}
```

虽然recover也是函数，但是这种方式，recover不起作用，提示的错误是 不应该在defer后面直接跟recover，应该使用一个匿名函数。

make和new

- make 的作用是初始化内置的数据结构，也就是切片、哈希表和 Channel，值得注意的是切片可以给定len和cap，后面的哈希表和chan只有len，并且slice可以省略cap
- new 的作用是根据传入的类型分配一片内存空间并返回指向这片内存空间的指针

go语言的条件和逻辑语句

条件语句

`if else` 是go里面的第一个条件语句，看一个例子就能一目了然：

```
func fast(a int){

    if a > 0 {
        fmt.Println("YES")
    }else if a == 0 {
        fmt.Println("YES!")
    }else {
        fmt.Println("no")
    }

}
```

可以看出，if后面跟一个判断的语句，并且，不加括号。

switch是第二个判断语句。

```
func fast(a int){
    switch a {
        case 1:
            fmt.Println("yes")
        case 2 :
            fmt.Println("yes!")
            fallthrough
        default:
             fmt.Println("no")
    }
}
```

这里面，switch后面是a，a是int类型，那么case后面也得是int类型,这里有个 fallthrouth 语句，意思是，走进这个case后不是直接出去，而是直接往下继续走。这跟其它语言不同，其它语言是默认继续走，然后使用了break才能出来，go是默认就是break了。

switch还有另一种跟断言结合的使用方法

```
func fast(a interface{}){
    switch a.(type){
        case int:
        case float64:
    }
}
```

断言的意义就是说搞清楚这个空接口具体的类型，所以叫做断言，断言的语句就是1. `一个空接口类型变量.(type)` 【只能在switch中使用】，另一种是 `b,ok := 一个空接口类型变量.(int)` 后面必须加具体类型，然后前面是第一个参数是返回的具体类型，第二个参数是布尔类型，判断是否断言成功。

逻辑语句

for 是go里面唯一的逻辑语句，在go里没有while和do while。这两个都是不合法的。

```
// 第一种表示一直for循环
for {

}
// 第二种表示，只要a>12就循环
for a > 12 {

}
// 第三种就比较传统了。
for i := 0;i < 12; i++ {

}
```

遍历语句

range 是go唯一的遍历语句

```
    b := make([]int,10)

    for a := range b {
    fmt.Println(a)
    }
```

这里要特别的指出来，a是对于b遍历的数据的值复制，也就是说下文中对于a的任何处理都不会影响原本的b里面的数据。

go语言中的面向对象

其实go语言并没有完整的面向对象，go语言可以说是面向接口编程的一个语言，go里面最重要的一个组件就是接口，接口让go语言变得非常灵活。

函数

go语言中的多返回值，使用的方式是栈，这不同于c的寄存器模式，优点是结构简单，可以从右向左压入栈中，计算的时候从栈中取出就会从左往右的排列了，而且可以天生支持多重返回，缺点就是速度比从寄存器模式慢了很多。

go里面的继承，使用的是结构体的内嵌

```
type People struct {
    face int

}
type Student struct {
    People
    year int
}
// 当然也可以选择不内嵌，将结构体当成一个类型也行

type Student struct {
    People People
    year int
}
```

在struct中也可以将接口当作一个类型加进去

```
type Talk interface {
    Say()
}
type People struct {
     Talk
}
```

看到这里你应该发现了把，在go里面变量的首字母的大小写是有不同的含义的，如果首字母是小写的，那么这个变量的使用权就是这个包的内部，如果是大写那么这个变量就可以被其它包使用。所以它的界限是以出包不出包为界线的。

接下来我们谈一个非常重要的东西 --- 方法

方法

```
type Student struct {
    year int
}
func(s *Student)Show(){
    fmt.Println(s.year)
}
```

这里我们讲解一下知识点，首先是方法的使用方式，如图所示，方法使用就是 `func(变量 对象)方法名称(){}` ,这里注意一下，并不是只有structure才能拥有方法，任何只要是以下方法的对象都可以拥有

```
type Student int
func(s *Student)get(){

}
type People map[string]string
```

只要是这种形式的都可以拥有方法，哦对了接口不行。因为接口没人任何的实际意义，它也就不可能拥有方法。

这里还有一个知识点，就是go语言中的指针，

```
// 这是指针类型的声明，我声明了，a是int类型，并且是一个int类型的指针类型。也就是说a装的不是i
var a *int

// 赋值，这里注意哈，new后面只能是类型 type ，所以这里可以使用new(int)因为int是一个类型。
a = new(int)
// 这样也是OK的。这个时候类型就是b了，b和int不是一个类型。
var a *b

type b int
a = new(b)
```

这里稍微谈一下 类型和底层类型的使用。

```
type b int
func fast()b{
    return 1
}
```

可以非常明显的看出，b类型的底层是int，但是b不是int，int只能是b的底层数据，就跟rune和int32，byte和uint8一样。 但是这个时候重点来了，这里有个返回值，返回的是b，那么具体要返回什么值才能满足b呢？也就是返回它的具体值即可。

更多例子：

```
type b map[string]string

func fast()b {
    return map[string]string{}
}
```

这里还要说一下，在go里面，值类型上的方法和指针类型上的方法是可以直接调用的。

举个例子：

```
type a sturct {

}

func(a1 a)get(){

}
// 这样百分之一百OK
func main(){
    var aa a
    aa.get()

}

// 这样也是OK的

func main(){
    var aa  = new(a)
    aa.get()
}

func(a1 *a)set(){}

// 这样一定OK

func main(){
    var aa = new(a)
    aa.set()
}

// 但是这样也OK

func main(){

   var aa a
   aa.set()
}
```

接口

接口的本质实际上是代码的解耦，接口的使用，跟接口的实现在两个部分。

go实现了隐式接口，只要实现了接口的方法就等于实现了这个接口，为此，有些特立独行的接口为了防止被实现就内置包内方法，这样外部包就无法访问了。

任何变量都实现了空接口 `interface{}` 所以说，只要是空接口的地方，所有的变量类型都可以放进去。当然了，内部需要断言一下。

接口的实现

上面说到了定义在值和指针类型上的方法可以互换，或者说可以语法糖式的调用，但是在接口的实现上不能打马虎眼。一定不行就是不行；

```
type Tallk interface {
    say()
}
type Student struct {

}
func(s *Student)say(){

}

func main(){
    s := new(Student)
    // 这里必须是指针类型才可以，值类型就不行， var s Student，就是错的。
    // 但是，如果是值类型实现的方法，那么指针类型的初始化是可以正常使用的，也就是说指针类型拥
    Td(s)
}
func Td(t talk) {

}
```

接口的继承

```
func main() {
    var c b
    c.set()
    c.get()
}

type a interface {
    get()
}

type b interface {
    a
    set()
}
```

go错误处理

我们知道传统的编程语言喜欢使用try模式，将很多错误一并处理，go语言不允许这么做，go的哲学就是，只要是错误，你就得处理，处理的越细致越好，看一下传统的go错误处理形如下文：

```
func fast()error {
    return fmt.Errorf()
}

func main(){
    if err := fast();err != nil {
        //xxx
    }
}
```

这就是一个非常常见，以及非常传统的go生成错误，以及处理错误的方式，使用 `fmt.Errorf` 生成标准错误，然后处理错误的时候，使用一个if语句，如果错误不等于nil（可以回忆以下这里为啥是不等于nil，我上文说过）那么就对错误进行处理，其中这里的 `fmt.Errorf` 可以更换以下，`errors.New("")` 也是生成了一个新的error对象。

接下来我们谈一下，最新的go版本里新添加的几个函数。

- `errors.Unwrap`

  ```
  func Unwrap(err error) error {
      // 先断言，
  u, ok := err.(interface {
  Unwrap() error
  })
  // 没有就返回nil，有就返回对象实例上面自己的Uwrap()方法
  if !ok {
  return nil
  }
  return u.Unwrap()
  }
  ```

  断言的时候，看看这个实现了error接口的对象实例，究竟有没有实现了Unwrap()error方法，这里的括号里面就是说要断言的具体的类型。这里是简略了。

  这里的省略，如果不省略就是下面这个样子

  ```
  type A interface {
      Unwrap()error
  }

  func Unwrap(err error) error {
  u, ok := err.(A)
  if !ok {
  return nil
  }
  return u.Unwrap()
  }
  ```

这是个小的知识点，可以留意以下。

- `errors.Is(err,target error)` 这个语句的意思是说，err是否等于后面的结果，如果等于就是返回的true，如果不等于就返回false

```
// 本代码来自go标准库源码

func Is(err, target error) bool {
if target == nil {
    return err == target
}
// 这句话的意思就是说可以比较吗
isComparable := reflectlite.TypeOf(target).Comparable()
for {
    // 如果可以比较，那么以及err等于target，就返回true
    if isComparable && err == target {
        return true
    }
    // 下文就是不匹配了，如果不匹配的话，就调用err的IS()方法。意思就是如果这个实现了
    if x, ok := err.(interface{ Is(error) bool }); ok && x.Is(target) {
        return true
    }
    // TODO: consider supporting target.Is(err). This would allow
    // user-definable predicates, but also may allow for coping with slo
    // APIs, thereby making it easier to get away with them.
    if err = Unwrap(err); err == nil {
        return false
        }
    }
}
```

- `errors.As(err error,target interface{})` As在err链中找到与目标实例匹配（意思是实现了共同的方法）的第一个错误，如果匹配，则将target设置为该错误值并返回true。否则，它返回false。

  通常来说，target本身是一个指针类型，然后传入的是这个指针类型的地址。下文中的代码里有相应的显示。

  这里稍微解释一下，err是一个调用的链条，只要其中一个跟target类型一样了，然后就把target赋值成err的值。而且这个target必须是指针类型。

  我们来看一下官方的案例

```
    func main() {
        if _, err := os.Open("non-existing"); err != nil {
            var pathError *fs.PathError
            //
            if errors.As(err, &pathError) {
                fmt.Println("Failed at path:", pathError.Path)
            } else {
                fmt.Println(err)
            }
        }

    }
```

go语言中的测试

- 错误测试
- 基准测试
- 例子输出
- main测试
- 子测试
- 跳过测试
- 文件系统测试
- io测试
- 黑盒测试

- http测试
- 性能分析
- http请求跟踪测试

测试文件的命名是有一套规则的，通常是某个文件相对应的测试文件，比如 `app.go` 的测试文件就是 `app_test.go`

错误测试

错误测试，也是测试中最基础的一种,test首字母要大写，后面的函数（测试谁写谁）首字母也要大写。使用 `go test` 命令进行启动。

```
func TestXxx(t *testing.T){
    if xxx {
        t.Errorf("xxx")
    }
}
```

基准测试

所谓基准测试，指的是go语言提供的某个算法或者程序执行一定的次数，然后输出平均的执行时间这个就叫做基准测试

跟test一样 `B` 大写， `Benchmark` 后面的函数首字母也要大写。

```
func BenchmarkXxx(*testing.B){
    // 这里的b.N go会自动提供，次数不一定。
     for i := 0; i < b.N; i++ {
        // 这里就是要测试的内容
        rand.Int()
    }
}
```

如果想在多线程的环境中测试，go给出了一个例子：

```
func BenchmarkTemplateParallel(b *testing.B) {
    templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
    // 这里的 RunParallel函数是关键。
    b.RunParallel(func(pb *testing.PB) {
        var buf bytes.Buffer
        for pb.Next() {
            buf.Reset()
            templ.Execute(&buf, "World")
        }
    })
}
```

范例测试

范例测试的意思就是说，运行的结果要跟你提供的例子保持一致

```
func ExampleHello() {
    fmt.Println("hello")
    // Output: hello
}

func ExampleSalutations() {
    fmt.Println("hello, and")
    fmt.Println("goodbye")
    // Output:
    // hello, and
    // goodbye
}
```

这里是有固定形态的， `//Output:` 是固定的用法，后面跟例子输出的结果。

main测试

测试来控制哪些代码在主线程上运行。

```
func TestMain(m *testing.M){
os.Exit(m.Run())
}
```

子测试

```
使用`t.Run()`可以进行子测试。
func TestTeardownParallel(t *testing.T) {
    // This Run will not return until the parallel tests finish.
    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })
    // <tear-down code>
}
```

关于子测试，命令行里的命令是不一样的：

```
go test -run ''      # Run all tests.

go test -run Foo     # Run top-level tests matching "Foo", such as "TestFooBar

go test -run Foo/A=  # For top-level tests matching "Foo", run subtests matchi

go test -run /A=1    # For all top-level tests, run subtests matching "A=1".
```

跳过测试

如果想跳过某些条件，可以使用 `t` 或者 `b` 的 `.Skip()` 方法。

```
func TestTimeConsuming(t *testing.T) {
    if testing.Short() {
        t.Skip("skipping test in short mode.")
    }
    ...
}
```

文件系统测试

这个包是啥意思呢？其实就是帮你模拟了一个文件系统，因为你比如要打开xx文件吧，你不需要单独真的去新建一个，使用这个文件系统的测试，就可以达到这个目的，这个包是 `testing/fstest`

```
    // 声明一个fstest.MapFs 对象，因为这个对象实现了fs.Fs接口
    //【func (fsys MapFS) Open(name string) (fs.File, error)】
    var ms fstest.MapFS
    // 如果这里不声明ms是这个对象，而是直接就初始化底层类型给ms，
    //下面函数要使用的ms就不是fstest.Mapfs对象，而是map[string]*fstest.MapFile对象
    // 当然你也可以直接不初始化，下面使用的时候显示转化一下即可
    // fstest.MapFs(ms) 即可。


    ms = make(map[string]*fstest.MapFile)
    mf1 := &fstest.MapFile{
        Data: []byte("test"),
        Mode: 30,
        ModTime: time.Now(),
        Sys: "12",
    }
    mf2 := &fstest.MapFile{
        Data: []byte("test1"),
    }
    // 前面是路径，后面是文件。这是一个模拟。
    ms["a/1"] = mf1
    ms["a/2"] = mf2
    fmt.Println(fstest.TestFS(ms,"a/1","a/2","a/3"))
```

这里多说一点，go里面的显示类型转化，刚才讲的一般使用的时候，type A int，A类型并不是int，只是它的底层是int，虽然它的一切操作都可以按照int来做，比如

```
type A int

 var a A
//a+1 就等于1,

// 但是它仍然是A类型不是int，要注意类型转化，只有一个地方go会自动的语法糖，就是return的
func fast()A{
        return 1
}
// 这里 return的时候进行自动类型判断了，没有把1判断为int，而是判断为A类型了。这属于语法
```

io测试

io测试包 `testing/iotest` 主要是实现了readers和writers，具体我们可以理解为，它实现了很多读取和写入。

黑盒测试

黑盒测试，使用的包是 `testing/quick`

比如一个可以快速得到是否正确的函数

```
func main(){
    f := func() bool{
        return 1 == 2
    }
    quick.Check(f,nil)
}
```

首先，对于我们来说，qucik.Check()是这个盒子的外壳，我们只能看到它，f我们是看不到的，所以我们可以通过check得到f的返回bool结果。

下面这个函数就是可以黑盒看f f1 是否是一致的。

```
func main(){
    f := func() bool{
        return true
    }

    f1 := func()bool {
        return false
    }

    quick.CheckEqual(f,f1,nil)
}
```

http测试

http测试，意思就是当你需要一个服务的时候，不需要自己再写一个http服务，你只需要 `net/http/httptest` 包即可。

这个包大致可以分为三个内容

1. request，请求 注意此包并不是客户端的请求，这是服务端的请求。

```
func NewRequest(method, target string, body io.Reader) *http.Request
```

目标是RFC 7230“请求目标”：它可以是路径或绝对URL。如果目标是绝对URL，则使用URL中的主机名。否则，将使用“ example.com”。

method 空是get

1. response，响应 这个包就是生成一个响应。

```
func NewRecorder() *ResponseRecorder
```

2. server，服务 服务器是侦听本地接口上系统选择的端口的HTTP服务器，用于端到端HTTP测试。

```
// 在这段代码中，第一段是一个server，下面是一个客户端get请求。所以上面哪个server监听了本地
func main() {
    ts := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *h
        fmt.Fprintln(w, "Hello, client")
    }))
    defer ts.Close()

    res, err := http.Get(ts.URL)
    if err != nil {
        log.Fatal(err)
    }
    greeting, err := io.ReadAll(res.Body)
    res.Body.Close()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s", greeting)
}
```

性能分析

`net/http/pprof` 包提供了例如gc，内存，cpu等数据的性能分析包。

如果要使用这个功能，需要写入这个 `import _ "net/http/pprof"`

以及将下面代码加入

```
go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

新开一个go的groutine，然后来进行性能分析。

> https://golang.org/pkg/net/http/pprof/

```
go tool pprof -http=:6062 http://localhost:6060/debug/pprof/block

go tool pprof -http=:6062 http://localhost:6060/debug/pprof/goroutine

go tool pprof -http=:6062 http://localhost:6060/debug/pprof/cpus
```

使用这个命令，可以把数据的分析，使用浏览器打开，http跟的端口，是自己设定的，后面的是分析的具体参数，比如 `/block` `/heap` 等。下面有个列表，最前面就是这些命令。

> runtime/pprof pprof的具体实现，所有类型的代码都可以使用。如果不是Web应用程序，建议使用该包。

类型	描述	备注
allocs	内存分配情况的采样信息	可以用浏览器打开，但可读性不高
blocks	阻塞操作情况的采样信息	可以用浏览器打开，但可读性不高
cmdline	显示程序启动命令及参数	可以用浏览器打开，但可读性不高
goroutine	当前所有协程的堆栈信息	可以用浏览器打开，但可读性不高
heap	堆上内存使用情况的采样信息	可以用浏览器打开，但可读性不高
mutex	锁争用情况的采样信息	可以用浏览器打开，但可读性不高
profile	CPU 占用情况的采样信息	浏览器打开会下载文件
threadcreate	系统线程创建情况的采样信息	可以用浏览器打开，但可读性不高
trace	程序运行跟踪信息	浏览器打开会下载文件

http请求跟踪测试

`net/http/trace` 包提供了监听http请求的各个过程的功能，我们来看一个例子

```
func main() {
    // 这里有一个新的request
    req, _ := http.NewRequest("GET", "http://example.com", nil)
    // 这里，有两个参数被监听
    trace := &httptrace.ClientTrace{
        GotConn: func(connInfo httptrace.GotConnInfo) {
            fmt.Printf("Got Conn: %+v\n", connInfo)
        },
        DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
            fmt.Printf("DNS Info: %+v\n", dnsInfo)
        },
    }
    // 将钩子放入这个http的请求之内，实现监听的效果。
    req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
    _, err := http.DefaultTransport.RoundTrip(req)
    if err != nil {
        log.Fatal(err)
    }
}
```

同步原语和锁

本章会含有以下内容：

- sync.Mutex
- sync.RWMutex
- sync.WaitGroup
- sync.Once
- sync.Cond
- sync.Pool
- golang/sync/errgroup.Group
- golang/sync/semaphore.Weighted
- golang/sync/singleftlight.Group
- github.com/marusama/cyclicbarrier

导读： mutex是互斥锁，也就是最基础的锁，RWMutex是读写锁，读写分离，写是强互斥，保持串行，读也有锁，但是降低锁的颗粒度，可以并发执行，但是读还是要让位于写，遇到写读就会强制等待，等写完成，总的来说还是提高了读的性能，waitgroup是让单goroutine等待多goroutine执行完毕，再去做某事，cond的应用场景就是等待以及通知，once就是只允许执行一次，pool就是线程池原理，这里传入的是临时变量，errgroup是将单任务变成多子任务的功能，semaphore是信号量，所谓信号量就是通过信号的增加和减少来控制某个操作的行为，可以理解为多个人用一个网吧会员，singleftlight叫做合并请求，意思是多个一模一样的请求合并为一个，cyclicbarrier叫做循环栅栏，先说栅栏意思就是多个goroutine达到某个条件后然后再去执行某件事，循环的意思就是它可以循环使用，并且循环的适合使用简单，不会Panic

sync.Mutex

这是go语言最基础的锁，也是很多原语都要内置的锁，比如chan中就使用到了这个锁，通常来说我们对锁的优化方法就是降低锁的颗粒度，举个简单的例子，现在1000个人公用一把锁，后面我们1000个人用10把锁，每100个人用一把，这就是降低了锁的颗粒度。

其中操作的时候可以这么做

```
l := new(sync.Mutex)
// 加锁
l.Lock()
// 解锁
l.Unlock()
```

我们看一下sync.Mutex的底层数据结构：

```
type Mutex struct {
    state int32
    sema  uint32
}
```

可以看出，state+sema，也就是状态+信号量一共64位，8个字节，就表示了整个go语言的互斥锁.

state按照位运算，后三位表示三种模式，即：mutexlock，mutexwwoken，mutexstarving，分别的意思是锁定，醒来，和饥饿，然后剩下的位都用来表示当前等待的goroutine数量。

sync.Mutex 分为正常模式和饥饿模式。正常模式下，按照先入先得的模式来获取锁，但是当一个被唤醒的goroutine跟一个新建的同时去获得锁的时候，就会发现，刚被唤醒的竞争不过这个新建的，体现就是这个g超过了1ms都获取不到锁，这个时候防止它被饿死，锁立马切换成饥饿模式，率先让队头的g获得锁，把这个新建的g放到队尾部，然后，当获得锁的g在队尾，以及g获取锁的时间小于1ms的时候，那么这个锁就会重新切换成正常模式。饥饿模式可以防止某个g一直无法获取锁，而被饿死的情况发生，引入了公平机制。

加锁和解锁过程，主要是控制的mutexlock字段，当锁的位置是0的时候，那么就使用atom包的原子操作，将值改成1，当互斥锁的状态不是0的时候，就会把这个g改成自旋的方式，继续去等待这个锁，当然了这个进入自旋是有条件的，比如必须是多核心cpu，比如自旋的次数不能大于4，比如必须存在至少一个正在运行的p，且队列是空。处理完自旋以后，系统就会更新mutexwoken mutexstarving这几个字段的状态。如果这个时候没有获得锁，那么系统就会通过信号量的方式，不允许两个goroutine获取到锁，并且将此g改成休眠状态，然后等待被唤醒。在正常模式下，这段代码会设置唤醒和饥饿标记、重置迭代次数并重新执行获取锁的循环；在饥饿模式下，当前 g 会获得互斥锁，如果等待队列中只存在当前 g，那么互斥锁就会从饥饿模式退回正常模式。

解锁的过程就是通过原子操作将mutexlock字段从1改为0，如果不等于0，那么就会尝试慢速解锁。首先解锁会查看是否已经解锁，如果已经解锁再调用unlock就会发生Panic，正模式下，如果发现没有等待者，或者mutexlock这几个字段不都为0，那么就会直接退出，不需要唤醒其它g，如果发现有其它g，那么就会唤醒其它g，然后移交控制权。饥饿模式下，这个锁会直接移交给下一个等待的g，并且这个时候饥饿模式不会被取消掉。

四种会发生错误的操作一定要避免

- 解锁已经解锁的锁，会Panic
- 复制这个锁，因为底层来看都是有状态的，复制以后就不是初始值了，肯定会出错
- mutex不支持重入，意思就是某个持有的goroutine可劲再调用锁，go不支持，因为go的锁不记录goroutine的信息。

几种可以提高锁的性能的方法

1. 降低锁的颗粒度
2. 引入排它锁，就是某个g持有了锁，其他锁就不再争夺了。

sync.RWMutex

读写锁，这是细颗粒度的互斥锁，写锁跟互斥锁一样，但是读锁的颗粒度就会降低，所以说不是说读就不加锁，而且锁的颗粒度降低。所以说读锁在读的时候会并发执行，来加快读的速度，读锁如果发现这个变量被写锁占用，那么就会等待，也就是说读会让位写，让写优先。读的时候并发执行来代替串行自然就提高了速度。

sync.WaitGroup

- sync.WaitGroup 必须在 sync.WaitGroup.Wait 方法返回之后才能被重新使用；
- sync.WaitGroup.Done 只是对 sync.WaitGroup.Add 方法的简单封装，我们可以向 sync.WaitGroup.Add 方法传入任意负数（需要保证计数器非负）快速将计数器归零以唤醒等待的 Goroutine；
- 可以同时有多个 Goroutine 等待当前 sync.WaitGroup 计数器的归零，这些 Goroutine 会被同时唤醒；

sync.Once

可以保证某段代码只执行一次。sync.Once.Do 方法中传入的函数只会被执行一次，哪怕函数中发生了 panic；两次调用 sync.Once.Do 方法传入不同的函数只会执行第一次调传入的函数；

sync.cond

Go 标准库提供 Cond 原语的目的是，为等待 / 通知场景下的并发问题提供支持。Cond 通常应用于等待某个条件的一组 goroutine，等条件变为 true 的时候，其中一个 goroutine 或者所有的 goroutine 都会被唤醒执行。 使用的机会较少。一般我们都是使用channel来代替这个东西

sync.Pool

go实现的临时线程池，可以往池里加东西，然后取东西，但是无法清楚的获取到底是取得到哪个东西

errgroup.Group

处理一组子任务，就该使用errgroup等这种控制分组的操作，它适合于将一个通用的父任务拆成几个小任务并发执行的场景。

它有三个方法

- WithContext 传入一个context，并返回一个group
- go 传入子任务
- wait 等待子任务的完成

```
package main
import (
    "errors"
    "fmt"
    "time"
    "golang.org/x/sync/errgroup"
)
func main() {
    var g errgroup.Group
    // 启动第一个子任务,它执行成功
    g.Go(func() error {
        time.Sleep(5 * time.Second)
        fmt.Println("exec #1")
        return nil
    })
    // 启动第二个子任务，它执行失败
    g.Go(func() error {
        time.Sleep(10 * time.Second)
        fmt.Println("exec #2")
        return errors.New("failed to exec #2")
    })
    // 启动第三个子任务，它执行成功
    g.Go(func() error {
        time.Sleep(15 * time.Second)
        fmt.Println("exec #3")
        return nil
    })
    // 等待三个任务都完成
    if err := g.Wait(); err == nil {
        fmt.Println("Successfully exec all")
    } else {
        fmt.Println("failed:", err)
    }
}
```

semaphore.Weighted

信号量顾名思义，就是使用某个变量比如 0 1的状态不同来去表示不同的状态，这就是信号量。 在信号量里有两个操作p和v，p就是减少信号量，v就是增加信号量。

golang.org/x/sync/semaphore 中是以下内容：

```
type Weighted struct{}
// 返沪一个 weighted
func NewWeighted
// p  操作
func Acquire
// v 操作
func Release
// 尝试获取 n 个资源，但是它不会阻塞，要么成功获取 n 个资源，返回 true，要么一个也不获取，返
func TryAcquire
```

所以说信号量用非常容易的话来说就是，假如你在网吧办了一张会员，你和你的朋友都能用，你们每用一次就是增加来一次使用次数，假设一共能用10次，越接近十次就越少机会继续使用，任何充钱每次充钱就把这个信号减去1，代表你们可以用的次数又多了。简单的信号量就是这么容易，使用某个信号充当某个作用。

所以说如果按照这种逻辑，使用channel来实现一个信号量也很容易，初始化就是给定一个chan多少容量，每用一次就往chan中添加几个次数，然后每次充钱就从chan中取回来多少信号。

使用chan有个缺点，就是一次只能实现取得一个资源但是golang.org的这种就可以一次获取n个资源，这也是golang.org实现这种方式的信号量的原因把。

singleftlight.Group

SingleFlight 的作用是将并发请求合并成一个请求，以减少对下层服务的压力，在处理多个 goroutine 同时调用同一个函数的时候，只让一个 goroutine 去调用这个函数，等到这个 goroutine 返回结果的时候，再把结果返回给这几个同时调用的 goroutine，这样可以减少并发调用的数量。在面对**秒杀**等大并发请求的场景，而且这些请求都是读请求时，你就可以把这些请求合并为一个请求，这样，你就可以将后端服务的压力从 n 降到 1

在高并发秒杀服务中，这种合并请求简直就是无敌的存在啊～

cyclicbarrier

CyclicBarrier 是一个可重用的栅栏并发原语，用来控制一组请求同时执行的数据结构。它常常应用于重复进行一组 goroutine 同时执行的场景中。举个例子，有一种场景，需要在多个goroutine同时达到某个条件时，然后让这些goroutine共同发生某种操作，并且这种等待然后操作是循环的，这个时候就可以使用循环栅栏，这个waitgroup也可以，不过因为waigroup重复使用要又注意事项，就是在wait结束后才能add所以容易出现bug，这个时候使用cyclicbarrier更好。

WaitGroup 更适合用在“一个 goroutine 等待一组 goroutine 到达同一个执行点”的场景中，或者是不需要重用的场景中。CyclicBarrier 更适合用在“固定数量的 goroutine 等待同一个执行点”的场景中，而且在放行 goroutine 之后，CyclicBarrier 可以重复利用

接下来我们来看一个使用了循环栅栏和信号量的一段代码，这段代码的意义是生成水分子。

```go
// github.com/marusama/cyclicbarrier
// golang/x/sync/semaphore
package main

import (
    "context"
    "fmt"
    cyc "github.com/marusama/cyclicbarrier"
    sema "golang.org/x/sync/semaphore"
    "math/rand"
    "sort"
    "sync"
    "time"
)
func main(){
    CreatH2O()
}
type H2O struct {
    H *sema.Weighted
    O *sema.Weighted
    b cyc.CyclicBarrier
}
func NewH2O()*H2O {
    return &H2O{
        // 这里配置好配方，一o二h，共计3个才能下一轮
        H: sema.NewWeighted(2),
        O: sema.NewWeighted(1),
        b: cyc.New(3),
    }
}
func (h *H2O)CreatH(r func()){
    h.H.Acquire(context.Background(),1)
    r()
    // 这里的wait的意思就是如果达不到三个，那么就 一直wait，这里的信号量的意义就跟我说的网
    h.b.Await(context.Background())
    h.H.Release(1)

}
func (o *H2O)Creat0(r func()){
    o.O.Acquire(context.Background(),1)
    r()
    o.b.Await(context.Background())
    o.O.Release(1)
}

func CreatH2O(){
    var N = 100
    wg := new(sync.WaitGroup)
    h2o := NewH2O()
    ch := make(chan string,N*3)
    r1 := func() {
        ch <- "H"
    }
    r2 := func() {
        ch <- "o"
    }
    wg.Add(N*3)
    for i:= 0;i <N*2;i++ {
        go func() {
            defer wg.Done()
            // 假设创造h不需要时间等待
            h2o.CreatH(r1)
        }()
    }
```

```
    for i:= 0;i <N;i++ {
        go func() {
            defer wg.Done()
            time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
            h2o.Creat0(r2)
        }()
    }
    wg.Wait()
    // 这里 chan初始化给定的是容量，但是len给不了，len是值的队列中实际存在的数据。所以初始
    if len(ch) != 3*N {
        fmt.Println("error,n is wrong",len(ch))
    }
    t := make([]string,3)
    for i:= 0;i< N;i++{ // 这里不要使用range，因为需要关闭chan才能跳出，使用select也可
        t[0]=  <- ch
        t[1] = <- ch
        t[2] = <- ch
        sort.Strings(t)
        water := t[0] + t[1]+t[2]
        if water != "HHo" {
            fmt.Println("error",water)
        }

    }
    fmt.Println("yes ,im ok ")
}
```

context包的使用

我们先看一下context最核心的 context interface的定义：

```
type Context interface {
    // 返回done的时间。
    Deadline() (deadline time.Time, ok bool)

    // done返回一个chan，也就是说，当取消运行的时候，这个done就会返回这么个chan。其实就是
    Done() <-chan struct{}

    // 如果done没有正常的关闭，那么就会返回错误。它只会在 Done 方法对应的 Channel 关闭时
    Err() error

    // value会返回一个key - value
    Value(key interface{}) interface{}
}
```

在context包中有以下几个函数返回的type实现了context接口：

- context.TODO()
- context.Background()
- context.WithDeadline()
- context.WithValue()
- context.WithCancle()
- context.WithTimeout()

这些函数返回的类型都实现了context接口。当然你也可以自己实现以下。毕竟这些都是可外漏的方法。

不过，像todo和background返回的都是没有任何意义的context，也可以把他们叫做root 上下文，也就是上下文的最顶端。

context，中文名就是上下文，它的主要目的就是作为树枝把goroutine串成一棵树，然后统一调度，也就是说，让这些个goroutine统一取消就一起取消，所以它才被翻译为上下文，因为真的是控制上下文的作用。

下面这个例子就是一个context使用的案例

```
func main() {
    ctx,cancel := context.WithCancel(context.TODO())
    ctx = context.WithValue(ctx,"0","0")
    ctx = context.WithValue(ctx,"1","1")
    go get1(ctx)
    go get2(ctx)
    cancel()
    time.Sleep(time.Second)
}

func get1(ctx context.Context) {
L:
    for {
        select {
        case <-ctx.Done():
            fmt.Println("该退出了get1",ctx.Value("0"))
            break L
        }
    }
    go get2(ctx)

}
func get2(ctx context.Context) {
L:
    for {
        select {
        case <-ctx.Done():
            fmt.Println("退出 get2",ctx.Value("1"))
            break L
        }
    }

}
// output :
//退出 get2 1
//该退出了get1 0
//退出 get2 1
```

使用context比并发原语或者扩展语句更加的简单。主要是操作的时候减少了deadlock的机会，不过也会引入狗皮膏药一样的ctx，不过总体来看context包确实能解决好了控制goroutine树这个问题。另外不要使用withvalue来轻易传递值，因为它寻址的时候是递归的往父树上去寻找，很慢的。

通道 channel

我们使用make来初始化这个chan，后面还有一个参数（这里没有len和cap的概念）这个参数是容量，拥有容量就是可以拥有buffered，没有容量就是必须收了发的数据，才能传递下一个。

通道的基本使用

```
// 初始化:
ch := make(chan Type,Cap)
// 例如:
ch := make(chan int,10)
// 写入数据
ch <- 1
// 或者可以使用select写入数据
select {
    case ch <- 1:
}
// 读取数据
<- ch // 这种方式即是读取数据，然后舍弃数据
f(<-ch) // 这种方式是读取数据，传入参数
a := <-ch // 读取数据，并且赋值给a

//当然读取也可以使用select

select{
    case <-ch :
}
```

另外chan中存在着几种内置函数，close关闭这个chan，len返回队列中的数据，cap是队列的容量，make后面的是cap不是len。make chan中后面只有容量着一个参数。

chan拥有下面三种形式：

- chan 可发可收的channel
- chan<- 只能发的chan
- <- chan 只能收的chan

注意一点，selec中的case中，发chan和收chan是平等的，都可以存在在case中，以及chan也是可以是chan存储的对象的，比如 `chan<- <-chan` 这个意思就是只发chan中存储了一个只收chan，<- 尽量跟左侧的chan结合，比如说 `chan<- chan` 意思就是只发chan中存储的是收发chan，如果想改变以下意思，变成chan中装只收chan可以加小括号来进行约束 `chan (<-chan)`

清空一个chan是有快捷方式的：

```
for range ch {

}
```

通道的实现原理

chan在go语言中的具体实现来自runtime.hchan,具体的数据结构如下：

```
type hchan struct {
    qcount   uint           //  循环队列中的元素个数，调用len，返回的就是这个变量。
    dataqsiz uint           // 循环队列的大小，就是cap返回时盗用的变量。
    buf      unsafe.Pointer // 循环队列的指针
    elemsize uint16 // chan中数据的大小
    closed   uint32 // 是否关闭通道
    elemtype *_type // chan中元素的类型
    sendx    uint   // send在buffered中的索引，意思就是发送数据的时候，这个队列中的ind
    recvx    uint   // receive 在buffered中的索引，同样的，取数据的时候，index所在这个
    recvq    waitq  // recv的等待队列，如果接收者，因为没办法接收了，那么它就会被加入到这
    sendq    waitq  // send中的等待队列，如果不能发送了，那么就会加入到这个队列中等待。
    lock mutex // 互斥锁
}
```

初始化chan的底层实现：

```
func makechan(t *chantype, size int) *hchan {
    elem := t.elem

    // 对于size进行检查
    if elem.size >= 1<<16 {
        throw("makechan: invalid channel element type")
    }
    if hchanSize%maxAlign != 0 || elem.align > maxAlign {
        throw("makechan: bad alignment")
    }


    mem, overflow := math.MulUintptr(elem.size, uintptr(size))
    if overflow || mem > maxAlloc-hchanSize || size < 0 {
        panic(plainError("makechan: size out of range"))
    }

    var c *hchan
    switch {
    case mem == 0:
        // 这里是指的是make后面没有指定cap的情况。
        c = (*hchan)(mallocgc(hchanSize, nil, true))
        // 这种情况下，就不必创建buffered了。
        c.buf = c.raceaddr()
    case elem.ptrdata == 0:
        // 如果chan中存储的不是指针类型，那么就给hcan数据结构分配空间。
        c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
        // 并且给buf分配空间
        c.buf = add(unsafe.Pointer(c), hchanSize)
    default:

        元素存在指针，单独分配buf
        c = new(hchan)
        c.buf = mallocgc(mem, elem, true)
    }
    // 记录元素的大小类型和容量
    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)
    lockInit(&c.lock, lockRankHchan)

    if debugChan {
        print("makechan: chan=", c, "; elemsize=", elem.size, "; dataqsiz=", s
    }
    return c
}
```

这段代码对于这三种方式给出了不同的分配内存的方式。

- 有buffered
- 无buffered
  - 值类型
  - 指针类型

发送：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
    // 第一部分先做判断，如果chan没有初始化（所以才是nil），
    if c == nil {
        if !block {
            return false
        }
        // 那么就无法继续往下走，gopark的意义就是挂起这个goroutine
        gopark(nil, nil, waitReasonChanSendNilChan, traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    }

    if raceenabled {
        racereadpc(c.raceaddr(), callerpc, funcPC(chansend))
    }

    // 这部分的意思是，如果队列中满了，但是不想阻塞，那么就会直接返回false
    if !block && c.closed == 0 && full(c) {
        return false
    }

    var t0 int64
    if blockprofilerate > 0 {
        t0 = cputicks()
    }

// 这部分的意思就是当一个已经关闭的chan，继续往里面send数据，那么就会Panic
    lock(&c.lock) // 添加 互斥锁
// 这里的直接意义就是，close关闭了，然后程序走到这一步了（因为是send函数，所以意思就是我关闭
    if c.closed != 0 {
        unlock(&c.lock) // 在这种情况下的解锁。
        panic(plainError("send on closed channel"))
    }
// 这一部分的意思是从接收者的等待队列中，拿出来一个接收者，然后把发送的数据交给它。很明显啊，
    if sg := c.recvq.dequeue(); sg != nil {
        // 可以看到 这个if中也解锁了。
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }
// 这种情况就是说buf还没满，需要把数据放入buf，然后返回即可。
    if c.qcount < c.dataqsiz {
        qp := chanbuf(c, c.sendx)
        if raceenabled {
            racenotify(c, c.sendx, nil)
        }
        typedmemmove(c.elemtype, qp, ep)
        c.sendx++
        if c.sendx == c.dataqsiz {
            c.sendx = 0
        }
        c.qcount++
        // 解锁了
        unlock(&c.lock)
        return true
    }
// 如果buf满了，就返回false
    if !block {
        unlock(&c.lock)
        return false
    }
```

```
    ....
}
```

接收：

```
//如果没有第二个参数的时候就直接调用chanrev
func chanrecv1(c *hchan, elem unsafe.Pointer) {
    chanrecv(c, elem, true)
}

//如果拥有第二个参数的时候，那么就按照这种方式调用 v,ok := <- chan 的形式
func chanrecv2(c *hchan, elem unsafe.Pointer) (received bool) {
    _, received = chanrecv(c, elem, true)
    return
}
```

接下来我们探究一下 这个chanrecv函数(片段，有删节)：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received boo
    ...
    // 这个的意思是说，如果这个chan没有初始化，那么接收的时候调用者就会被阻塞。
    if c == nil {
        if !block {
            return
        }
        // 这个函数的意思就是阻塞挂起
        gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)
        throw("unreachable")
    }
    ...

// 这部分的意思是说，首先先加锁
    lock(&c.lock)
//如果chan被关闭了，然后队列中没有元素了，那么返回true和false并且解锁
    if c.closed != 0 && c.qcount == 0 {
        if raceenabled {
            raceacquire(c.raceaddr())
        }
        unlock(&c.lock)
        if ep != nil {
            typedmemclr(c.elemtype, ep)
        }
        return true, false
    }
    // 这部分的意思是 sendq中有等待者（意思就是等着收的接收者队列中有等待者）那么如果buf中有
if sg := c.sendq.dequeue(); sg != nil {

        recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true, true
    }
    //没有等待的发送者，buf中还存在数据，取出一个数据给reciver
    if c.qcount > 0 {
        // Receive directly from queue
        qp := chanbuf(c, c.recvx)
        if raceenabled {
            racenotify(c, c.recvx, nil)
        }
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }
        typedmemclr(c.elemtype, qp)
        c.recvx++
        if c.recvx == c.dataqsiz {
            c.recvx = 0
        }
        c.qcount--
        unlock(&c.lock)
        return true, true
    }

    if !block {
        unlock(&c.lock)
        return false, false
    }

    ...

    // 下面的处理内容就是 buf中没有数据，reciver阻塞，直到它从sender中取出来了数据。
```

close:

```
func closechan(c *hchan) {
    if c == nil { // chan是nil的时候panic，所以不能close一个没有初始化的chan，结局就是
        panic(plainError("close of nil channel"))
    }

    lock(&c.lock)
    if c.closed != 0 {
        // 如果chan的close已经被标记为0了（意思就是这个chan已经被关闭了）也是panic
        unlock(&c.lock)
        panic(plainError("close of closed channel"))
    }

...

    c.closed = 1

    var glist gList

    // 释放所有的reader
    for {
        sg := c.recvq.dequeue()
        ...
        gp := sg.g
        ...
        glist.push(gp)
    }

    // 释放所有的writer，他们会panic
    for {
        sg := c.sendq.dequeue()
        ...
        gp := sg.g
        ...
        glist.push(gp)
    }
    unlock(&c.lock)

    ...
```

使用chan最容易犯的错误

- panic
- 协程泄漏

panic

下面这些，上面源码分析中都有提示

- close chan等于nil的chan
- send已经close的chan
- close已经close的chan

协程泄漏

举一个简单的例子

```
func Test(){
    for i:= 0;i < 100;i++ {
        go func(){
               select {

               }
        }()
    }

}
```

这种情况下，这些个协程就永远被阻塞了，也就是永远不会被gc掉，也就是所谓的goroutine泄漏，下面给出一个稍微生产环境一点的例子:

```
func A(timeout time.Duration)int{
    ch := make(chan int)
    go func(){
        // 模拟耗时处理
        time.Sleep(time.Second + timeout)
        ch <-1
        fmt.Println("test ")
    }()
    select {
        case v := <-ch :
            return v
        case time.After(timeout):

            return -1
    }
}
```

这种情况下 ch绝对不会被接收，因为select中走 time.After了，然后上面那个协程，ch就用于啊不会发送1，因为当select运行的时候，收已经准备好，这个时候按道理是可以接收发传来的data的，然后发并没有准备好，所以收的case只能阻塞了，那么当time.After搞定以后，就直接走这个case了，那么结局就是收关闭了，上面的goroutine就会一直阻塞到ch <- 1这个地方，原因就是内存模型：无buf的chan只有收准备好，才能发，收不可能准备好了，所以发的操作就挂起阻塞了。

解决这个问题也很简单，就是给这个buf设置1，这个时候发就可以开始了，当然没有收它的reader，这个chan很快就会被gc掉。

```
func A(timeout time.Duration)int{
    // 设置一个buf
    ch := make(chan int,1)
    go func(){
        // 模拟耗时处理
        time.Sleep(time.Second + timeout)
        // 这里将不再阻塞
        ch <-1
        fmt.Println("test ")
    }()
    select {
        case v := <-ch :
            return v
        case time.After(timeout):

            return -1
    }
}
```

什么时候用chan，什么时候用并发原语

- 共享资源的并发，使用并发原语
- 复杂的任务编排和消息的传递使用chan
- 消息通知机制使用chan
- 等待所有任务的完成使用并发原语（waitgroup），当然了如果是可循环的那种，直接使用 cyclicbarrier，【也就是说符合并发原语典型场景的用并发原语就对了，比如合并请求用singleleftlight，分组的使用errgroup】
- 需要跟select结合的用chan
- 需要和超时结合的，使用chan和context

值得注意的事情

一个通道中如果含有数据，这个时候数据并没有被接收读取，然后它被close了，那么这个值仍然可以被读取，然后再读才是零值，当然了，被close掉的chan是不能再往里面send数据了

	nil	空	满	不空也不满	closeed
接收者	阻塞	阻塞	正常读取	正常读取	读取队列中未读取的值，然后再读取到零值
发送者	阻塞	正常发送	阻塞	正常发送	panic
close内置函数	panic	正常关闭	正常关闭，数据可以正常被读取完	正常关闭，数据可以被正常读取完	panic

通道的使用场景

通道是一个带有锁的队列，包括含有buffered的队列和没有buffered的队列，如果 chan 中还有数据，那么，从这个 chan 接收数据的时候就不会阻塞，如果 chan 还未满（“满”指达到其容量），给它发送数据也不会阻塞，否则就会阻塞。
unbuffered chan 只有读写都准备好之后才不会阻塞，这也是很多使用 unbuffered chan 时的常见 Bug。

chan拥有以下五种使用场景：

- 数据传递
- 数据交流
- 信号通知
- 任务编排
- 锁

数据传递，简单的理解就是从一个g传递到另一个g，击鼓传花

数据交流，channel基于生产者消费者模型的无锁队列

信号通知，使用chan传递信号，最典型的就是传递退出信号，跟context结合的时候

任务编排，如果按照某种逻辑，使用chan来使得这些g按照我们设想的逻辑顺序去执行

实现锁，锁不只有互斥锁，还有例如排外锁等

计时器

时间对于系统来说尤其重要，比如分布式系统中，相对的时间非常重要，不过目前最佳的分布式系统也是存在误差的，go计时器的目的是为了获取相对的时间。标准库中还提供了定时器、休眠等接口能够我们在 Go 语言程序中更好地处理过期和超时等问题。

go语言的计时器发展经历了三个阶段分别是：

- 全局四叉堆
- 分片四叉堆
- 单独管理四叉堆，网络轮询器触发

全局四叉堆

```
var timers struct {
    lock         mutex
    gp           *g
    created      bool
    sleeping     bool
    rescheduling bool
    sleepUntil   int64
    waitnote     note
    // 这个字段存储的就是四叉堆
    t            []*timer
}
```

其中这个字段中的t，后面跟的[]*timer就是表示的四叉堆，这个期间，全局维护一个timers，使用一个lock进行互斥加锁，当然了，明显颗粒度较大。

在go的运行时期间，有两种情况会唤醒计时器，1四叉堆中的计时器到期了，2是四叉堆中加入了更早触发的计时器

分片四叉堆

由于全局的计时器，因为互斥锁的颗粒度实在是太大，所以系统改成了64片的互斥锁，这个64基本上是根据内核核心数设置的，如果p（pmg模型中的p）的个数大于了64，那么就将节点改成桶，往桶里叠加即可。不过这种行为就会造成cpu和线程的上下文切换时间增加。切换造成的时间浪费又成为了新的麻烦。

```
const timersLen = 64

// 这是一个含有64个timerbucket的全局切片。
var timers [timersLen]struct {
    timersBucket
}

type timersBucket struct {
    lock         mutex
    gp           *g
    created      bool
    sleeping     bool
    rescheduling bool
    sleepUntil   int64
    waitnote     note
    t            []*timer
}
```

这种情况下可以将锁的颗粒度从1改为64

单独管理四叉堆

最新的四叉堆，取消了计时桶，所有的四叉堆都存放在处理器p（P:M:G中的p，属于运行时的上下文调度处理器）中，

```
type p struct {
    // 锁
    timersLock mutex
    // 四叉堆
    timers []*timer
// 存放处理器中的四叉堆的数量
    numTimers     uint32
    // adjust状态的计时器的数量
    adjustTimers  uint32
    // deleted状态的计时器的数量
    deletedTimers uint32
}
```

目前计时器都交由处理器的网络轮询器（netpool）和调度器（P:M:G）触发。

计时器的数据结构

内部的计时器数据结构下面所示，这种结构存放在各自处理器的四叉堆中，所以下面的数据结构其实就是四叉堆中存放的元素的数据结构。

```
type timer struct {
    pp puintptr

    when     int64
    period   int64
    f        func(interface{}, uintptr)
    arg      interface{}
    seq      uintptr
    nextwhen int64
    status   uint32
}
```

暴漏出来的计时器：

```
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
```

time.Timer 计时器必须通过 time.NewTimer、time.AfterFunc 或者 time.After 函数创建。 当计时器失效时，订阅计时器 Channel 的 Goroutine 会收到计时器失效的时间

go语言运行时的调度器

P:M:Ggo运行时调度模型

go语言runtime调度器

runtime.gosched()基于阻塞的协程调度

io.rader/writer unix 文件哲学

go语言的网络io模型

net/http 基于goroutine的http服务器

go语言的内存模型 --- happens-before

此处讨论的内存模型跟内存分配和内存gc一点关系都没有，是属于并发的概念。它描述的是在并发环境中，读取相同变量时，变量的可见性，比如某个g在什么时候可以读取到另一个g修改过的变量。

指令重排和可见行的问题

go编译器为了实现优化，有的时候你写的代码的顺序不一定按照你写的顺序执行，

```
package main

var a, b int

func main() {
    go b1()
    a1()

}

func a1() {
    print(a)
    print(b)
}
func b1() {
    a = 1
    b = 2
}
```

这里，print的b的值不一定是2，有可能是0，因为a1（）不一定能看得到b1中的a b的先后关系。因为跨goroutine了。

go内存模型中有个非常重要的概念 happens-before，如果严格按照这个概念去执行，那么程序代码在并发的多goroutine中对于变量的读取，便不会出现上述的问题。

happens- before

happens-before的定义就是xx一定发生在xx之前，并且其它干扰项不会同一时间出现进行干扰，也就是说happens-before发生的时候就是单指这两者，不能有第三方去干扰。

在一个goroutine内部，程序的执行和代码的顺序严格保持一致，即使指令重排了也不会改变这个原则。这不过另一个goroutine看不到这种规则，因为这个规则只限于单个的goroutine内部。那么对于外部的goroutine我们要记住happens-before就可以避免这些问题。

go内存模型通过 happens-before和happens-after定义了三个状态，即：发生在我之前，发生在我之后，还有就是同时发生。

go的导包过程中的happens-before的执行过程是这样的，比如main包里面有init和main以及var和const，main导入的包是a1，a1同样拥有 const var init a1又有a2导入，a2同样有这么一套东西，那么执行的顺序就是 a2的cosnt var init（如果不止一个，一个包内可以有很多个，但是一个文件只能有一个，按照文件名称顺序执行init）然后再来就是a1的const var init 再来是 main的cosnt var init然后最后是main函数。

启动goroutine的go语句这个执行的过程，一定是happens-before 这个goroutine内部的执行的。不过值得注意的是，goroutine退出的时候并没有任何的happens-before保证。

channel中的happens-before：

- 往这个chan中发送数据一定happens-before 从这个chan中取出这个数据，即第n个发一定比第n个收早。
- 关闭一个chan一定happens-before从这个关闭的通道住读取它的一个零值。
- 对于一个没有buffered的chan，从这个chan中读取数据这个操作一定happens-before往这个chan中发送数据这个操作。
- 对于一个容量为m（>0）的的chan，第n个recive，一定是happens-before n+m个send。意思就是这些缓存可以在没有收的情况下即使是充满了也不会去看send有没有准备好。所以收的这个happens-bere就是跟上面的一样，before n个发，只不过这里有缓存所以就是n+m个发。

mutex和rwmutex中的happens-before：

- 第n次的unlocok一定happens-before 第n+1次的lock
- 对于读写锁，只有释放了写锁，才能去调用读锁
- 对于读写锁，lock必须等待读锁释放了才能获取 waitgroup中的happens-before：
- wait方法只有等到计数值归零了才会返回 once中的happens-before：
- sync.Once(f)，f函数一定是在once的返回值返回之前执行，意思是说，这个once执行的时候，f函数一定是最早执行的那个，但是不一定能执行完啊，只是说它一定最早执行。

atomic中的happens-before：

目前 atomic 无法保证。

go语言并发模型 --- csp 通信顺序进程

csp在go中的实现就是goroutine之间顺序执行，chan在之间交流通信。

在cspgo的实现里，有着非常流行的一句话，不要通过共享内存来通信，而是应该通过通信来共享内存。通过通信就是通过chan。

通常来说，不同的goroutine之间是没有任何的关联的，然后一个g通过chan往另一个g传递了信号，这个时候他们就有了关联

```
func main(){
    ch := make(chan int)
    // 1 g
    go func(){
        ch <- 1
        close(ch)
    }()
    for v := range ch {
        fmt.Println(v)
    }
}
```

这段代码中，1g就往主g中发送了一个信息1

go语言的逃逸分析

go语言的反射

反射的意义是可以在go代码执行的过程中，动态操作对象，其中最重要的两个函数就是

- reflect.TypeOf

```
func main() {
var a interface{}
a = 1
fmt.Println(reflect.TypeOf(a))
a = "1"
fmt.Println(reflect.TypeOf(a))
a = true
fmt.Println(reflect.TypeOf(a))
a= map[int]int{1:1}
fmt.Println(reflect.TypeOf(a))
}
```

- reflect.ValueOf

  获得类型的value值，进而对值在运行时进行操作。

前者是为了动态的获取目标的类型，后者是为了获得目标的value值。

go语言项目线上事故排查

使用go的pprof

内存泄漏

协程泄漏

[syscall基于操作系统的原生syscall能力