Golang

导包
- import "fmt"
- 别名：ft import "fmt"
- 点操作：import . "fmt"
  - 在你调用这个包的函数时，你可以省略前缀的包名。也就是调用的：fmt.Println可以省略的写成Println
- 只执行init：_ import "fmt"
- init函数
  - 按导入顺序初始化，最先需要导入的依赖包，它的init函数最先执行
  - 每个包只会被初始化一次

数据类型
数字类型
- int
  - int8，8位，一个字节，-127～128
  - int64，8个字节
  - 32位编译器就是4个字节、64位编译器就是8个字节
- unit
  - 0▒255
  - 无符号类型
- float
- bool
- string
派生类型
- struct
- pointer
- 数组
- 切片
- channel
- interface
  - go中interface与具体类型之间的转换，赋值时将实际数据复制了一份进行操作的
- map

值类型与引用类型
- 值类型
  - int、float、bool 和 string ，数组，struct
- 引用类型
  - ptr，函数，映射，slice，map，chan，interface

变量
- 全局变量
  - 全局变量声明但不使用不会报错
- 局部变量
  - 局部变量声明但不使用会报错
- 已经声明的变量不允许重新声明，但可以被重新赋值

数组
- array = [5]int{1,2,3,4,5} //声明并初始化
- var array [5]int //声明
- 根据索引值初始化
  - array := [5]int{1:1,3:4}，只对索引为 1和3的值初始化，其余使用默认初始化值0
- 不支持负数索引
- 多维数组

切片
- 定义的切片与make创建的切片，在cap方面有区别。前者超出cap时会自动扩容，而后者会报错
  - 容量：更像一个警戒值，如果等于或超过警戒值，就会自动将切片进行扩容
  - 自动扩容，每一次都是上一次cap的2倍。当缩减容量时，每次cap减去缩减元素的个数。
- var a []int // 切片(引用类型)声明后没有初始化无法使用。
  b := [3]int{} // 数组(值类型)声明后可以直接使用
- 左闭右开：a[2:5]，实际只有2、3、4三个元素
- append
  - append函数会创建一个新的底层数组，将原数组的值复制到新数组里，再追加新的值，就不会影响原来的底层数组。如果原底层数组够用，则不重新创建新数组。
  - append会重新给变量重新分配空间，所以不会出现index out of range [1] with length 1这种错误
- 可以在range时动态改变

string
- 几种类型
  - 单引号：字符
  - 反引号：原样输出文本，自动转义

tag
用来增强结构体的定义，Tag会带上一些meta信息
- Tags可以由多个键值对来组成，通过空格符来分割键值
  - f string `one:"1" two:"2"`
- omitempty
  - ProductID int64 `json:"product_id,omitempty"`
  - 如果字段具有空值，false、0、零指针、nil接口值以及任何空数组，切片，映射或字符串，则该字段应从编码中省略
- 可以指定序列化、反序列化的后的类型
  - ProductID int64 `json:"product_id,string"`：由int64变为string
- 特殊情况：-
  - 如果字段标签是 "-"，则该字段总是被省略。
- 常与reflect包一起使用

for
- go中只有for一种循环形式
- for有3种使用形式
  - range
- 循环控制语句
  - break
  - continue
  - goto

指针
- 取地址符是 &，但一般使用fmt.printf("%p",xx)
- &:求地址，*取地址对应的值
- 空指针：nil
- 被指针指向的指针
  - 子主题
  - var ptr **int;

常量
- const
  - 显式类型定义： const b string = "abc"
  - 隐式类型定义： const b = "abc"
- 注意，常量、常量表达式(所有数都为常量的运算)在编译时就被确定了值
- iota
  - 特殊常量，可以被编译器修改
  - 子主题
  - iota 在 const 关键字出现时将被重置为0，const中每新增一行常量声明将使 iota 计数一次

```
const (
        c0 = iota
        c1 int = (10*iota) // 10
        c2  // 20
        d
        e  // 40
        f = iota  5
)
```

- 在定义常量组时，如果不提供初始值，则表示将使用上行的表达式

占位符
- %v，相应值的默认格式， 一般用在输出结构体数据中
- %T，打印类型
- %p，十六进制表示，前缀 0x。一般用在打印地址
- %s，格式化字符串

编码
- 单引号是字符，双引号是字符串

Map
- 无序的键值对的集合，其中key必须是支持==比较运算符的数据类型
- 用法
  - 判断键值是否存在：value ,ok := name["name"]。
  - 不存在则返回value对应类型的零值
- 遍历
  - 如果是同时遍历key，value。那么Map的迭代顺序是不确定的，每一次遍历的顺序都不相同。
  - 如果只遍历key，则是有顺序的
- 可以通过map实现set类型：map[int]bool{}
- 可以在range时动态改变
- 其他特点
  - 删除：delete(a, "age")
  - 如果key不存在，也不会报错
  - 相同的k-v，会被后面的k-v覆盖

goroutine
用户级线程
- 普通线程开销大约2M～8M
- 特点
  - 占用内存小、最小的栈只有2kb，后面会指数增长。最大是1GB
  - 在程序启动时，Go程序就会为main()函数创建一个默认的goroutine。类似与主线程
  - go fmt.Println(<-ch1)。会从ch1中取数据
  - go 语句后面的函数调用，与defer类似，会先求值
MPG模型
先创建多个内核级线程，然后用自身的用户级线程去对应这些内核级线程，自身的用户级线程需要本身程序去调度，内核级的线程交给操作系统内核去调度，达到M：N的效果。
- M
  - M必须拿到P，才能对G进行调度，让这个G在自己这运行
  - 内核态OS线程，真正干活的人
- P
  - 局部的调度器，是实现从N:1到N:M映射的关键
- G
  - 子主题
  - goroutinue
工作原理
- 1、Go会根据当前机器的逻辑CPU个数来创建相应数量的P，并将它们存放在一张空闲P列表中。和P一样，如果一个M没有工作可做了，该M会被放入空闲M链表中：
- 2、新创建并等待被运行的协程会唤醒一个P来执行这个任务，这个P会创建一个和系统线程相关联的M并与这个M绑定
- 3、Mth行完P维护的局部队列后，它会尝试从G的全局队列寻找G，如果全局队列为空，则从其他的P维护的队列里窃取G到自己的队列
- 如果G0被阻塞，调度器会将P与当前的M0和G解绑，同时去空闲M列表中找新的M1，如果没有空闲M，则创建M1并与P绑定，再顺序执行P下的G。等G0不阻塞了，M0也会被放回M的空闲队列。

Channel
- 控制goroutine通信，比如在可能出现资源竞争的地方建立无缓冲channel
  - 无缓冲通道
  - 有缓冲通道
  - 无缓存channel与size为 1的channel是不一样的。无缓冲通道channel必须在接受方与发送方同时准备好时，通道才 能正常传递数据
  - 缓冲区一满，数据发送端就无法再发送数据了
- 通道关闭后几种情况
  - 1、对一个关闭的通道再发送值就会导致panic
  - 2、对一个关闭的通道进行接收会一直获取值直到通道为空
  - 3、对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值
  - 4、关闭一个已经关闭的通道会导致panic

并发
- 控制并发
  - 全局共享变量
  - channel通信
  - Context包
  - wg sync.WaitGroup
  - 主goroutine：wg.wait()，子goroutine：wg.Add(1) -- > wg.done()
- 并发模型
  - CSP（communicating sequential processes）并发模型，即通过通信来共享内存。
  - Go的CSP并发模型，是通过goroutine和channel来实现的

Struct
- 一系列具有相同类型或不同类型的数据构成的数据集合。
- 结构体的属性地址都是连续的
- 比较
  - 相同类型的结构体才能比较
  - 子属性类型、顺序完全相同才相同
  - 可以比较的子属性有bool，数值型，字符，指针，数组。所以，当结构体中存在map类型，哪怕它们是完全相同的，也无法进行比较
- 拷贝
  - 如果结构体中有切片，以赋值方式拷贝后，这两个结构体对象的切片会互相影响
  - 空结构体的size是0，所有的空结构体内存分配都是同一个地址
- 结构体嵌套

go mod
- 开启 go mod
  - export GO111MODULE=on
- go init xxx
- go mod download
- go mod tidy
- go mod vendor
- indirect
  - 间接引用
  - 没有明确的出现在某个import语句中
- 子主题
- 代理
  - export GOPROXY=https://goproxy.io
  - export GOPROXY=https://goproxy.cn

go tool
- pprof
  - 1、生成 .out性能检测文件
    - go test -bench=. -benchmem -memprofile memprofile.out -cpuprofile profile.out
  - 2、go tool pprof profile.out进入交互终端
    或者go tool pprof -http=:8080 profile.out在本地查看，里面的flamegraph就是火焰图
    - 方块的大小代表了占用 CPU 使用的长短
- trace
  - 查看gc情况、内存分配使用情况

go build
- go build -n xxx.go 打印编译的所有操作而不执行。
- go build -x xxx.go 打印编译的所有操作并且执行。
- go build -p n xxx.go 使用n个cpu进行编译（速度明显变快）
- go build -race xxx.go 查看是否存在数据竞争
- go build -a xxx.go 强行对所有涉及到的代码包（包括标准库中的代码包）进行重新构建，即使它们已经是最新的了
- go build -ldflags 命令参数 xxx.go 会在每次go工具调用时执行，传递参数
- CGO_ENABLED=1，即默认开始cgo，允许你在Go代码中调用C代码。

go 其他操作
- clean
  - go clean -n 查看会操作的命令但不执行
  - go clean -x 打印操作并执行命令
  - go clean -cache 清楚所有build缓存
- fmt
  - go fmt -n
  - go fmt -x
- get
  - 默认是下载并安装第三方包。可以加-d只下载不安装

编译
- 非静态编译：CGO_ENABLEED=0 GOOS=linux /usr/local/go/bin/go build -a -installsuffix cgo -ldflags '-w' -i -o homeweb-web main.go
- 静态编译：CGO_ENABLED=0 GOOS=linux /usr/local/go/bin/go build -a -ldflags '-extldflags "-static"' -o homeweb-web server.go

测试
- 普通测试
  - 文件名必须以 _test.go为结尾
  - 方法名必须以Test开头，且遵循小驼峰命名规范
  - 参数必须是t *testing.T
  - 对包名不作要求
- 基准测试
  - 文件名必须以 _test.go为结尾
  - 方法名必须以Benchmark开头，且遵循小驼峰命名规范
  - 基准测试函数不能有返回值
  - 被测试代码要放到循环里。b.N是基准测试框架提供的，表示循环的次数，因为需要反复调用测试的代码，才可以评估性能
- 测试覆盖率
- 子测试

类型转换
- 隐式类型转换
  - 如果两个类型的底层类型相同，并且它们之中至少有一个为非定义类型，则其中任一个类型的值可以隐式转换为另一个类型。
- 显式类型转换

自定义类型
- 自定义类型，就是一个真正的类型了，与int不是同一种类型
- 要想使用自定义类型，需先初始化
- 当两个类型的底层基础类型相同时，允许T()这种转型操作。如果T是指针类型，可能会需要用小括弧包装下，比如(*int)(0)
- 类型别名：type MyInt = int，与int完全相同,只存在写代码的过程，编译后不存在

锁
- 互斥锁
  - 使用简单，但性能较差
  - 同时只能有一个goroutine有锁
- 读写锁
  - 读多写少时，用读写锁性能会更好。如果读写都差不多，那与互斥锁性能差不多
  - 多个读操作可以共享
  - 多个写操作是互斥的
  - 写与读操作，写优先。读操作要等写操作结束后再执行
- 死锁
  - 死锁是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去
  - 如果已创建并活动的线程数大于等待工作的线程数，则会出现死锁情况

错误处理
- panic
  - 用于主动抛出错误
- recover
  - 用来捕获 panic 抛出的错误。
  - recover()只能和defer一起使用，并且defer只有在定义在panic之前才能捕获到异常
  - 多个panic只会捕捉最后一个
  - 如果recover被嵌套使用，那么也无法捕获异常

函数
- 所有的函数参数都是值拷贝传入的，函数参数将不再是函数调用时的原始变量。必要时可以添加指针
- go语言中没有默认参数
- func demo2(a ...int) int { // 参数后面加...表示可变参数
- demo(slice...) 如果要直接传切片，只需在后面加...即可
- 如果定义了函数返回值的名称，则在函数体中无需再次声明，并且return时可不写该变量名称
- go函数可以返回多个值
- 匿名函数，f := func() {}

接口
- 是一种抽象的类型
  - 并且可以把所有的具有共性的方法定义在一起，任何其他类型只要实现了接口里的全部方法就是实现了这个接口
- 空接口
  - 空接口，可以放任何类型的数据，因为任何方法，所以任何类型都实现了这个接口
- 接口的值包括type、value两部分
- 类型断言，val, ok := xxx.(type)
- 类型判断，switch v := x.(type)

闭包
- 函数的返回值也是一个函数，并且内函数使用到了外部作用域的变量，就可以认为是一个闭包
- 闭包会使得函数中的变量都被保存在内存中，内存消耗很大

select
- 监听channel IO的的控制语句
  - select没有输入值，只用于信道操作
- case必须是一个通信操作，要么是发送要么是接收
- select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。
- 如果没有可以执行的case。此时如果有default子句，则执行该语句。
- 没有default字句，select将阻塞，直到某个通信可以运行
  - 如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。
  - 其实可以人为的加些条件控制case的优先级

defer
- 当defer被声明时，其参数就会被实时解析
- 在函数return中间执行
- 多个defer，先定义的后执行
- 外部变量的引用是有两种方式的
  - 作为函数参数
    - 在defer定义时就把值传递给defer，并被cache起来
  - 作为闭包引用
    - 会在defer函数真正调用时根据整个上下文确定当前的值
- 如果返回值已经声明了，并且存在defer的话。那么这个值最终由defer决定，而不是return

Make
为 slice、map或 channel 类型(引用类型)申请内存空间并初始化，同时返回一个有初始值的slice、map 或 channel 类型引用

New
new为 int struct(值类型)申请内存空间并初始化，同时返回一个指针

context
作用
- 在不同 Goroutine 之间对信号进行同步。与此同时 Context 还能携带以请求为作用域的键值对信息
- 每一个 Context 都会从最顶层的 Goroutine 一层一层传递到最下层。如果没有 Context，当上层执行的操作出现错误时，下层其实不会收到错误而是会继续执行下去。
种类
- context.Background
- context.TODO()
context的继承与衍生
context包为我们提供的With系列的函数用来衍生。
- WithCancel
  - 传递一个父Context作为参数，返回子Context，以及一个取消函数用来取消Context。
- WithDeadline
  - 和WithCancel差不多，它会多传递一个截止时间参数，意味着到了这个时间点，会自动取消Context，当然我们也可以不等到这个时候，可以提前通过取消函数进行取消
- WithTimeout
  - 和WithDeadline基本上一样，不同之处在于它将持续时间作为参数输入而不是时间对象
- WithValue
  - 和取消Context无关，它是为了生成一个绑定了一个键值对数据的Context，这个绑定的数据可以通过Context.Value方法访问到
- 前三个函数都返回一个取消函数CancelFunc，这是一个函数类型。该函数可以取消一个Context，以及这个节点Context下所有的所有的Context，不管有多少层级
使用原则
- context.Background 只应用在最高等级，作为所有派生 context 的根
- 不要把Context放在结构体中，要以参数的方式传递
- Context作为参数的函数方法，应该把Context作为第一个参数，放在第一位
- Context是线程安全的

内存模型
栈
- 操作系统控制，不需要gc回收
堆
- 堆内存最初会是一个完整的大块，即未分配内存，当来申请的时候，就会从未分配内存，分割出一个小内存块(block)，然后用链表把所有内存块连接起来
  - 释放内存实质是把使用的内存块在链表中标记为未使用，当分配内存块的时候，可以从未使用内存块中有先查找大小相近的内存块，如果找不到，再从未分配的内存中分配内存。
  - 当然，还没考虑内存碎片的问题。随着内存不断的申请和释放，内存上会存在大量的碎片，降低内存的使用率
内存块
- 一个内存块包含了3类信息：元数据、用户数据和对齐字段
gc
- off状态：GC总是从 Off，如果不是 Off 状态，则代表上一轮GC还未完成
- 1、Stack scan：收集全局变量和 goroutine 栈上的变量。遍历全局变量的话需要STW开启写屏障，G stack只需要堵塞目标goroutine。（因为全局变量可能会多个goroutine在争夺，所以需要STW来暂停）（1.8版本之前之后该阶段不开启STW）
  - 写屏障：可以理解为为了防止在gc时，这个阶段原来的对象引用被更改，而短暂的挂起go进程。只对堆空间启动，栈空间不启动写屏障
- 2、mark：标记对象，直到标记完所有根对象和根对象可达对象。此时写屏障会记录所有指针的更改。
  - 采用三色标记法，三色标记将对象分为黑色、白色、灰色三种
    - 黑色：已标记的对象，表示对象是根对象可达的
    - 白色：未标记对象，gc开始时所有对象为白色，当gc结束时，如果仍为白色，说明对象不可达，在 sweep 阶段会被清除。
    - 灰色：被黑色对象引用到的对象，但其引用的自对象还未被扫描，灰色为标记过程的中间状态，当灰色对象全部被标记完成代表本次标记阶段结束。
    - 过程：
      1. 开始时所有对象为白色
      2. 将所有根对象标记为灰色，放入队列（只放灰色）
      3. 遍历灰色对象，将其标记为黑色，并将他们引用的对象标记为灰色，放入队列
      4. 重复步骤 3 持续遍历灰色对象，直至队列为空
      5. 此时只剩下黑色对象和白色对象，白色对象即为下一步需要清除的对象
    - 由于引入了灰色对象这一中间状态，标记过程和用户的 golang 代码中可以并发执行，不需要STW，这极大的减少了应用的停顿时间。
    - 三色标记使用写屏障(Write Barrier)避免在标记过程中对象应用的改变
- 3、Mark Termination：重新扫描上个阶段发生更改的栈变量，为保持引用关系一致性，该阶段会STW(Stop The World)，也是 gc 时造成 go 程序停顿的主要阶段。
  - 重新扫描，是为了防止在标记过程中，对象引用发生变化，导致清除仍在使用的对象
- 4、Sweep: 并发的清除未标记的对象，如果此次gc持续到下次gc开始，那么跟下次gc一期执行，gc周期会有叠加的风险。
gc触发
- 主动触发，用户代码中调用 runtime.GC 会主动触发 GC
- 默认每 2min 未产生 GC 时，golang 的守护协程 sysmon 会强制触发 GC
- 当 go 程序分配的内存增长超过阈值时，会触发 GC

go运行时
- go程序并没有像java一样自带虚拟机(java虚拟机负责将上层代码与底层系统交互)，而是和c类似，。每一个go程序都带有一个runtime，runtime负责和底层操作系统交互
  - 协程调度，内存分配，GC；
  - 操作系统及CPU相关的操作的封装(信号处理，系统调用，寄存器操作，原子操作等)，CGO；
  - pprof，trace，race检测的支持；
  - 等几乎所有操作
- go没有虚拟机概念，go的runtime与用户代码一起打包在可执行文件中。
  用户代码与runtime代码在执行的时候没有明显的界限，都是函数调用
  Go对系统调用的指令进行了封装，可不依赖glibc（纯静态编译）