第二章

1. a, b := map[key], a代表value，b是一个bool类型，用于判断值是否存在

2. var c sync.WaitGroup c.add(nums) 用waitgroup来管理goroutine

3.匿名函数传值方式 func(A int , B int){  
…

}(a ,b) a和b就是传入的值

11.10

1.字段 tag ‘ key: value ’ 在 定义时，在类型后加一个tag，用于增加标记信息.

> go vet tags.go

可以检查tag是否合理

2.defer用于关闭文件时，放在open后面，可以增加代码可读性（defer file.close( ) ）

3.

type Feed struct {

name type ‘json:name’  
…

}

var feeds \*Feed

json.NewDecoder(file).decode(&feed)

以上代码将file文件中的json数据解析，并存到feed结构体数组中,更确切的说是Feed类型指针的切片

4.接口如果只有一个方法，接口名应当以er结尾

5．函数声明时若带有接收者（及func (name type)funcname(){ }）的形式，则声明了一个方法，使用该类型的值或者指针都可以调用该方法。同理接收者type为指针时也可以

大部分情况下吗，声明为指针可以更好维护接收者状态。

但是如果将接收者传给接口来调用方法的。见下：

> go vet tags.go

var dm defaultType

var inter InterfaceName = &dm

声明接收者为指针时，只能用指针类型来调用。

6.如果需要某一个包的init（）函数而不使用其变量和函数，需要加一个下划线 \_

7. fmt.Errorf（””,param） 用于返回自定义的错误

11.14

1. array := […]int{1,2,3} ; array :=[3]int{1:25 , 2:66}

2.数组作为参数在函数中传递时是值传递，会完整拷贝数组，通过指向数组的指针来替代是好方法。

3.声明空切片的两种方法：slice:=make(int,0) slice:= [ ]int{ }

4.对切片newSlice = slice[i:j ] 长度为j-i ； 容量为 k -i

5. newSlice 和slice共享同一个底层数组，在调用append时，如果没有超过slice的cap，两者共享同一个底层数组，如果超过了，newSlice会指向一个新的底层数组（与slice分开）

6.切片三索引：slice[i:j:k] i代表起始位置，j代表切片长度结束位置（不包含j）,k代表切片容量结束位置（不包含k）j和k都不能超过slice原有的长度及容量

7.调用append函数时，为了防止不同切片共享同一个底层数组，应该限制切片的长度和容量一样。 方法：（1）三索引【i, j , j】(2)开始声明slice时就限制容量和长度一致。

8.append(slice1, slice2…) 可以向slice1追加整个slice2

9. for range的形式迭代slice时，拿到的value和index是slice的副本

10.切片作为函数参数传递时，传递的时切片本身的值，即 地址指针+长度+容量，底层数组不传递，但共享（不调用append并且产生新底层时），返回的也是副本

11.映射使用两个数据结构来存储数据：（1）数组，存储了hash值高位，用于决定键值对存在哪个桶（2）字节数组，依此存放所有键和值

11.15

1.方法分为值接收者和指针接收者

根据定义决定了类型。值接收类型的方法，值和指针都可以调用，但都是值副本

2.go的引用类型包括：映射、函数类型、切片、通道、接口

2.22

go的接口声明：

type interface\_name interface{  
 func\_name ()()

}

go接口的实现方法：在声明函数时指定接收者则是实现接口：

func (name type) func\_name()(){ }

2.23

goroutine: 协程

1.主线程（可以理解为进程），协程（特点：有自己的堆栈、由用户调度、属于轻量级线程）

go协程是逻辑态的，可以理解为虚拟线程，执行时由主线程合理分配到各个cpu,最大的利用cpu

2. MPG模式

M:主线程 ； P:协程上下文； G: 协程

MPG模式下，M0挂有多个协程，如果其中一个协程G0阻塞，则创建新线程M1(或者从线程池中取出)，将M0剩下的协程挂载到G0上。G0阻塞结束后，M0放入空闲线程池。

3. goroutine之间的通讯：

两种方式：全局变量互斥锁、channel

互斥锁：sync.Mutex

mutex.lock() mutex.unlock()

channel : 本质：先进先出队列，含有类型：

channel定义方式： var chan\_name chan type

channel为引用类型，需要先make后才能使用。

使用接口可以让通道接受不同类型的数据：var chan\_name chan interface{ }

要搭配类型断言使用。

使用close(chan\_name)来关闭channel，关闭后只能读不能写，可以使用for range来读取。

**2.24**

1. 如果channel只写不读，会导致阻塞。

2. 可以使用var chan\_name chan <-type 定义只写；用var chan\_name type <- chan 定义只读

4. 用select可以避免channel不关闭导致的阻塞。

for {  
 select{  
 case v:= <-chan\_1 :

case v:= <-chan\_2 :

default:

break

}

}

5. 在协程函数中使用defer + recover来获取协程的error

func func\_name(){  
 defer func{  
 if err:= recover(); err !=nil{

//发生错误，报告错误

}

}()

//函数正常操作

}

6. 反射机制：

变量、接口、reflect.value的相互转换

7.桥接模式：抽象和实现部份分离

在实现系统时，有多个角度多个功能，将这些实现抽离出来，使其互不干扰，能更灵活的增删。

8 . tcp socket 编程

服务器端流程：监听8888端口，与客户端建立tcp链接，创建goroutine,处理来自客户端的请求

客户端流程：与服务器端建立链接，发送请求并从服务器端接受返回结果，关闭链接

**9.buffer / builder / bufio**

io操作涉及两个接口 io.Reader 和 io.Writer

1. byte.Buffer是一个简单字节缓冲池。 用于在高频率IO操作时用作缓冲区。

Buffer实现了io的两个接口，实现方法如下：

(2)strings.builder 用于高效写入或者拼接字符串。拷贝builder,底层共享同一个数组。写入时会判断切片是否为空。

(3)bufio 。定义了两个结构体，Reader和Writer。分别实现了io.reader和io.writer接口，通过先读写文件到缓冲区来提高IO效率。

bufio.read(p [ ]byte)的具体实现如下：

1. 首先判断缓冲区，如果有内容，将缓冲区内容拷贝至p并清空缓冲
2. 如果没有内容，判断p存储大小和缓冲区大小。如果len(p) >len(buffer) ，直接从文件读取数据到p,完成操作。如果小于，先从文件中读取数据到缓冲区（由io.Read实现，应该是一直读取知道缓冲区满），最后将缓冲区内容读取p,并清空缓冲区。

bufio.Write(p []byte)的具体实现如下：

1. 判断缓冲区剩余容量是否够存入p的内容
2. 如果够，将p写入缓冲区
3. 如果不够，且缓冲区为空，直接将p写入文件
4. 如果不够，且缓冲区不为空，p先将缓冲区填满，然后将缓冲区内容写入文件。接着判断p剩余文件大小和缓冲区关系。
5. 如果剩余文件超过缓冲区，直接写入文件。同c
6. 如果小于缓冲区，将p写入缓冲区。同a

字符串拼接：（1）+ （2）fmt.Sprintf （3）buffer.build （4）buffer.writeString

**2.24**

1、go 类型转换：（1）强制类型转换。普通变量使用type(a)，指针变量需要

pa = (\*type)(unsafe.Pointer(pb))（2）类型断言. var a interface{ } = 10 switch a.(type)

或者是 t ,ok := a.(int)

**2. go读取文件有四种方法：**

（1）NewReader()

file := open(path)

r := bufio.NewReader(file)

n, err := r.Read(buf)

n返回的是一次读取的长度，要读取文件全部内容，需要使用for循环读取直到n==0.

**2.26**

1．atomic包操作： atomic.AddInt32( &a, n) 对于Uint类型需要 atomic.AddUint32(&a, ^Uint32(-NN-1)) 假设NN =-43 是一个负数 。如果 NN是一个常量const ,则使用NN&math.MaxUint32来代替^Uint32(-NN-1)

2. watigroup。

var wg sync.WaitGroup 。变量wg有三个方法，分别是Add、Done和Wait。

wg.Add(+/- n)， wg.Done() =wg.Add(-1)

waitgroup 用于控制不同协程之间的控制流程：如G2、G3、G4需要在G1执行后再执行

方案一：使用通道

chan := make(chan int,3)

go func(){ //G2

…  
 channel <- 1

}

for i:0;i<3;i++{ //G1

a:=0  
 channel ->a

}

方案二：使用sync.WatiGroup

wg := sync.WatiGroup

wg.Add(3)

go func(){ //G2

…  
 wg.Done()

}

wg.Wait() //G1

3. 切片做参数时是值传递，传递的是切片结构体信息，相当于传递指针

准确的说，是传递的对原切片对应底层数组的再切片，由于共享底层数组，所以传值后修改时会影响原切片。但是如果函数中的切片参数因为调用append且超出容量，会因此创建一个新的底层数组，导致与函数外的原切片割裂。

切片只有copy是深拷贝，[:]和=都是浅拷贝

make([]int,n) n=0创建空切片， n>0创建0切片； slice:=[ ]int{}或者slice:=\*new([ ]int)创建nil切片

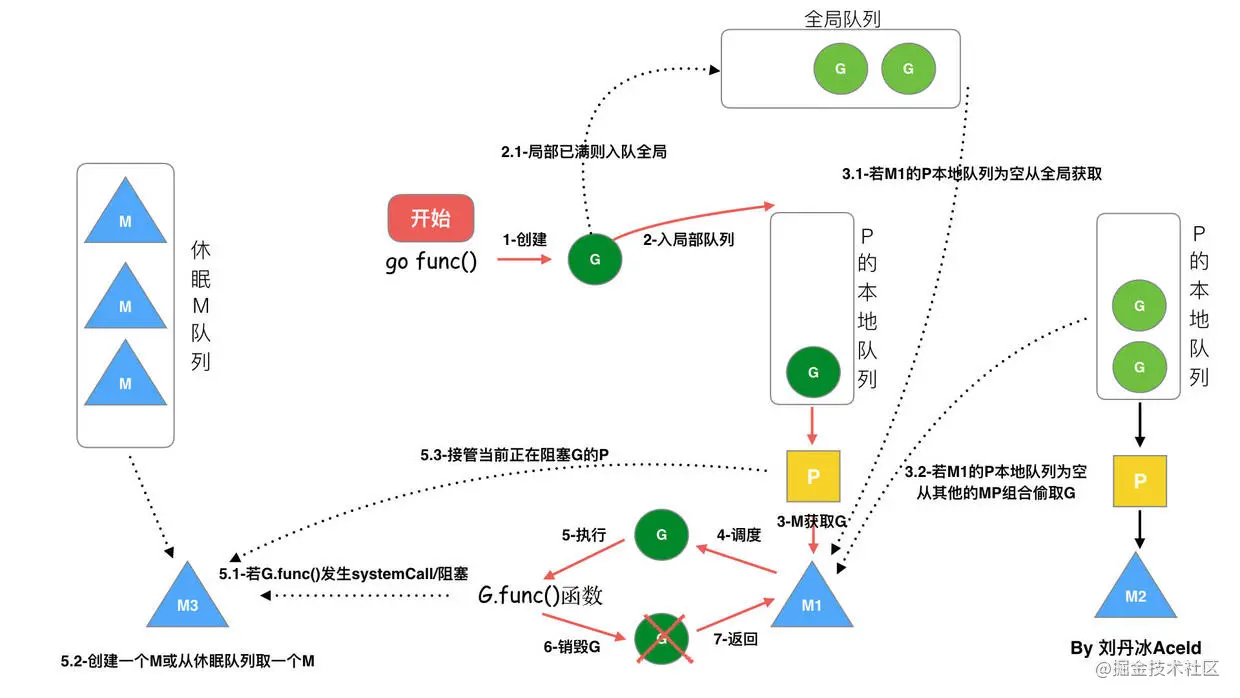
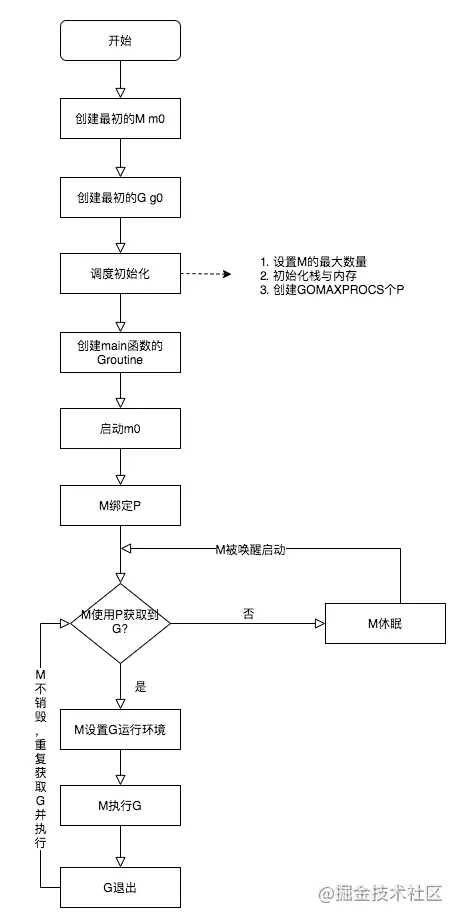
切片扩容策略：growslice. 进行内存对齐，原切片容量<1024，新切片容量变为2倍，大于等于时变为原来的1.25倍

切片是对数组指针的封装，用结构体来实现。

2.28

GMP调度流程大致如下：

* 线程M想运行任务就需得获取 P，即与P关联。
* 然从 P 的本地队列(LRQ)获取 G
* 若LRQ中没有可运行的G，M 会尝试从全局队列(GRQ)拿一批G放到P的本地队列，
* 若全局队列也未找到可运行的G时候，M会随机从其他 P 的本地队列偷一半放到自己 P 的本地队列。
* 拿到可运行的G之后，M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。



G - Goroutine，Go协程，是参与调度与执行的最小单位

M - Machine，指的是系统级线程

P - Processor，指的是逻辑处理器，P关联了的本地可运行G的队列(也称为LRQ)，最多可存放256个G。

Work Stealing和Hand Off策略保证了M的高效利用

3.15

(1)输入输出的实现：

输入：(1)fmt .(2)建立缓冲 in:= bufio.NewReader(os.stdin) //bufio.NewScaner(os.stdin)

1. tcp socket编程：

服务端：net.Listen(“tcp”,ip+port) ; listen.Accept(); go process(conn);

for {conn.Read()}

客户端：net.Dial(); con.Write([]byte)

序列化过程：Message = type +data ,每种消息单独定义结构体，先marshal（loginMessage）,再marshal（Message）

服务器端结构：

main.go(初始化，监听，创建gorutine)-> processor.go(根据客户端请求选择不同的处理程序) ->userProcessor.go / smsProcessor.go

utils.go 常用工具化函数，如网络read，write等

客户端结构：

main.go( 显示界面/更具用户选择不同处理程序) ->sms.go / user.go -> server.go(和客户端保持连接)

4.3 Go源码类型：命令源文件（main包+main函数，生成可执行于/bin目录下）+库源文件+测试源文件

4.4

(1)Golang channel底层原理：

通过runtime包里的hchan结构体实现，主要包括了：循环链表、指针、锁 构成

刚开始buf为空,写数据时先加锁，然后通过copy向buf传递消息，并修改指针。释放锁

读数据时加锁，然后从buf中copy，修改指针，释放锁

读写阻塞时，创建新的结构体，然后记录协程G和待读写数据的地址，加入到等待队列中

Go channel循环链表实现：sendx = (sendx+1)%size; 再用一个int描述缓冲区已存数据大小

具体底层代码实现：

1. send()，即向通道发送数据（chan <- 100）：首先判断接收队列是否为空，如果已经有人在等了，直接copy到接收者，并唤醒该协程。否则向缓冲区copy数据，sendx和qcount++；如果缓冲区满了，则将该协程加入发送等待队列进行阻塞，等待唤醒。
2. recv()，即从通道读取数据( num :=<-chan):首先判断发送队列是否为空。如果已经有人在等待发送了。如果是无缓存的通道，直接把发送者数据拷到接收者，并唤醒发送者。如果是有缓存的，从缓存区拷贝数据（队列头）到接收者，并将发送者队列第一个人的数据拷贝至缓冲区（队列尾）后唤醒他。

如果没有人等待发送，从缓冲区拷贝数据给接收者后返回。如果缓冲区为空，则将该协程阻塞，加入接收者等待队列。

（2）defer

两个特性：延迟执行，先进先出。（在return的赋值操作后，在return 的ret操作前执行）

应用场景：异常处理，I/O操作（或者各类需要使用完后释放的操作），debug

(3)异常处理

panic发出异常，recover接收异常，异常接口：

type error interface {

Error() string

}

err := errors.New(“abc”)或者 err := fmt. Errorf(“%s”,”abc”)

注意：recover需要在defer中调用且只接受上一级的panic，所以defer recover和嵌 套defer recover都不能捕获。 但是panic会嵌套传递。

1. 闭包

外部函数可访问内部函数变量，且内部函数结束后变量存于内存而不是立刻销毁。通过内存逃逸，从原栈逃逸到堆中。

4.5

1. map底层实现

go使用hash table实现map ,使用链式结构解决hash冲突。以bmap作为最小结构进行冲突处理。

hmap结构体，关键字段：buckets, count, hash0等

buckets作为桶，即bmap[ ]结构体数组； bmap结构体中有topbits[] , keys[] ,elements[ ]等关键字。一个bmap中最多存八对键值对。

读取map的过程：（有两种方法，接受一个返回参数和接受两个参数）

1. 判断map是否为空，以及判断map是否是有竞争。
2. 计算key的hash值，高8位作为tophash, 低B位作为桶序号
3. 判断hmap的oldoverflow字段，判断是否有扩容发生，如果old不为空先在old中找。
4. 进行内外层循环遍历，先比较tophash，再寻找key与对应value。外层循环桶到溢出桶，内层循环桶内bmap的8个key。

写map过程：

1. 判断map是否为空，以及判断map是否有竞争
2. 计算key的hash,低B位作为桶序号
3. 判断是否有扩容正在进行，有的话先搬迁
4. 遍历桶，比较tophash。如果不等，记录第一个空出来的位置。相等时，需要再判断完整hash是否相等。
5. 如果没有找到，根据空位写入。如果需要增加溢出桶，先判断装载因子是否>6.5以及是否溢出桶数量过多。若是，需要先进行扩容后再写入。若不满足扩容条件，则增加溢出桶。

扩容条件两个：

1.装载因子：count/2^B ,即平均每个桶中元素，当factor > 6.5时需要进行扩容。

2. overflow 过多 >2^B

扩容方式：

如果是因为条件1, 则使用增量扩容；如果是因为溢出桶过多，则使用等量扩容。

map的扩容流程：

1. 首先根据扩容条件选择扩容模式
2. 然后分配新桶，并将老桶挂载到oldbuket，然后每次操作时进行两个桶的搬迁
3. 每次搬迁主要是（1）重新计算桶序号。同一个桶内的key进行一个rehash。新的桶序号= 新加位==0，桶号不变。 || 新家为==1，桶号=低key+2^B

简单来讲，扩容后，变为了前桶和后桶。确定了区间，就可以用buketx +n\*oldorder来快速计算。

**冲突处理**

当两个不同key落在同一个桶时，就发生了hash冲突。首先寻找桶及溢出桶内是否有该key，没有则寻找第一个空位。如果空位不够，则添加溢出桶。

注意事项：

map是线程不安全的，map如果value是结构体时，需要用结构体指针才能对结构体内元素进行修改。

* 1. Golang GC

1. 常用垃圾回收（2）标记-清除（3）分代回收
2. Golang 三色标记法。四步：1.栈扫描，标白。2.root灰。3.灰引加灰，灰变黑4.循环至灰空，回收白
3. 写屏障。保证黑不引白
4. 减少STW时间。标记和程序并行。
5. 触发方式：1.主动 runtime.GC 2.被动 系统监控时间+步调算法内存

go内存分配到堆的情况：函数返回指针，make,slice，变量很大时，

4.12 方法：

（1）值接收和指针接收

指针可以直接调用方法，但是仍然是指针指向值的副本。但是定义方法时接收者为指针则可以直接改变对应的值。同时可以使用值调用指针接收者的方法，同样可以改变原值。（编译器自动转换）

（2）分为三种类型：1.内置类型（数值，字符串，布尔）2.引用类型（切片、通道、函数等）通过传引用类型的副本实现传指针的效果。3.结构类型

（3）多态：针对不同类型提供一个统一的接口，以此实现某一类操作应用于不同的类型。

4.13 接口：

接口= 类型+数据；所以接口==nil 等价于类型==nil和数据==nil

（1）底层结构iface{itab,data};itab用于描述实体类型+接口类型+方法集指针； data记录具体的值。

（2）断言的使用：A.(type) (1)val,ok :=A.(type1);if ok{}; (2)switch A.(type) case type1:

（3）接口的静态和动态特性：静态特性（1）静态类型，赋值时进行类型检查（2）动态实体数据：存有运行时具体的值。

1 2 3 4

4 3 2 1

2 1 3 4

GO语言面向对象的实现：

1. 封装：基于包，通过大小写来实现封装，等价于public和private的效果
2. 继承：通过组合的方式，引用别的包的变量。不允许相互引用。
3. 多态：通过接口实现，指定方法接收者。