go进阶

channel

1.底层实现

- 数据结构
  - 1.buf，指定大小的环形队列，环形队列的读索引和写索引
  - 2.sendq、recvq，读消息的和写消息的协程队列，表示协程的结构体，*sudog
  - 3.closed，通道是否关闭的标记
  - 4.mutex，保护所有数据结构
- 数据流程
  - 发送操作概要
    - 1、锁定整个通道结构。
    - 2、确定写入。尝试从协程等待队列recvq中获取一个等待goroutine，然后将元素直接写入goroutine，结束。（zero copy）
    - 3、如果recvq为Empty，则确定缓冲区是否可用。如果可用，从当前goroutine复制数据到缓冲区。
    - 4、如果缓冲区已满，则要写入的元素将保存在当前正在执行的goroutine的结构中，并且当前goroutine将在sendq中排队并从运行时挂起。
    - 5、写入完成释放锁。
  - 读取操作概要
    - 1、先获取channel全局锁；
    - 2、尝试从等待队列sendq中获取等待的goroutine；
    - 3、如有等待的goroutine，没有缓冲区，取出goroutine并读取数据，然后唤醒当前goroutine，结束读取释放锁；
    - 4、如有等待的goroutine，且有缓冲区（此时缓冲区已满），从缓冲区队首取出数据，再从sendq队列中取出一个goroutine，将goroutine中的数据存入buf队尾，结束读取释放锁；（不可能出现有等待的goroutine，但是缓冲区未满的情况，这种情况下等待的goroutine会直接把数据写入缓冲区）
    - 5、如没有等待的goroutine，且缓冲区有数据，直接读取缓冲区数据，结束读取释放锁。
    - 6、如没有等待的goroutine，且没有缓冲区或缓冲区为空，将当前的goroutine加入recvq排队，进入睡眠，等待被写goroutine唤醒，结束读取释放锁。

2.可能出现的误用

- 协程泄漏
  - 启动的新协程中从一个无缓冲通道中获取数据，但是该通道一直没有其他协程写入数据，导致协程未能退出
  - 从一个已关闭的带缓冲的协程中读取数据，未判断通道状态，导致一直读取空值，未退出
- 读取已close的chan，导致panic

3.作用

- 信号同步（finish/close？shutdown）
- 消息传递(queue/stream)
- 互斥(mutex)

defer

- 误用
  - defer 中改变某个返回变量的值；改变无效，因为defer在return之后执行。

```
func main() {
 i := 1
 j := add(&i)
 fmt.Println("i: ", i,"j: ", j)
}

func add(i *int) int {
 defer func() {
  *i = *i + 1
 }()
 return *i
}
```

1.defer 将匿名函数压栈

2. return语句将*n放置在番薯返回值内存区

3. 函数退出，j使用返回值赋值给自己

4.defer调用执行，*n被增加（即i）

5.打印i/j

  - defer后如果是单独语句，会对表达式提前求值。不能正确满足某些逻辑。defer fmt.Printf("cost: %v", time.Since(start))
- 使用注意事项
  - 1.defer声明时刻即参数解析时刻
  - 2.defer执行结果为先进后出
  - 3.defer中可以访问程序内部变量，但是不可修改返回值

Mutex/RWMutex

- 误用
  - 使用defer释放锁，但是在调用的函数中嵌套使用了锁
- 最佳实践
  - 锁的粒度越小越好，加锁后尽快释放
  - 没有特殊原因，尽量不用defer释放锁
  - RWMutex的读锁不要嵌套使用

map/sync.Map

- 误用
  - 使用sync.Map时只写不读，然后删除。结果导致内存泄漏。https://github.com/golang/go/issues/33762
    - 1. 删除时只将key对应的value设置为nil，以期GC回收；
    - 2.但是有些复杂对象，GC难以回收
- 适用场景
  - map+RWMutex
    - 1. 读写相对均衡，机器内核数少于4
    - 2.高并发读写
  - sync.Map
    - 1. 读多写少，机器内核数大于4
    - 2. 并发读写的key无冲突
- syns.Map实现

```
type Map struct {
 mu Mutex
 read atomic.Value // readOnly
 dirty map[interface{}]*entry
 misses int
}
```

  - 特性
    - 1. 通过read和dirty两个字段将读写分离，读取时会先查询read，不存在再查询dirty，写入时则只写入dirty；
    - 2.读取read并不需要加锁，而读或写dirty都需要加锁；
    - 3.另外有misses字段来统计read被穿透的次数，超过一定测试则将dirty数据同步到read上；
    - 4.对于删除数据则直接通过标记来延迟删除。
  - 缺点
    - 1. 内存占用是标准map的两倍；
    - 2. 对于少于4核的机器，性能并不比map好

变量遮蔽

- 定义：同名变量使用短声明导致的覆盖
  常发生于短声明 :=
- 避免名字shadow的建议
  - 1. 不要再嵌套代码块内使用短变量声明；一定要用，则尽早退出；
  - 2.至少有一个新变量时，外部定义；
  - 3.关注imort . 时的名字遮蔽问题
  - 4.谨慎使用if foo,err := do(); err != nil{}这类语句
- 子主题 3