Golang随记（单测和channel）

- 测试：尝试去跑一次程序，观察产生的结果，并和预期进行断言，一致表示通过；非一致代表有问题，再去定位具体的错误，再去解决错误。
- 单测的最终目标是通过去写出一些优质的单测代码（合理的，考虑周全的），来帮我们去改进我们业务代码的质量，避免写出更多的bug。然后再更进一点，希望能够在单测过程中，倒逼我们去思考我们在原本的业务代码上有没有什么可以改进的地方。
- 单元测试代码难写的原因？业务逻辑复杂？（如果对业务认知足够清晰，拆分足够到位，通过单测覆盖每个小问题）单元测试难写在于不稳定，比如我们的函数中包含了一些第三方的服务（http，rpc，数据库），这些不幂等请求会有一些不确定情况，从而导致我们的单元测试的不确定性，这对于强一致性的测试来说是不可接受的。如何去解决呢？优化架构，细分好服务，面向对象编程，同时在编写单测时，使用符合相同interface类型的自定义幂等对象进行测试，从而聚焦到需要被测试的代码，隐藏其它第三方实现的细节。
- 面向过程：化整为零，把大问题拆解为一个个小问题，再去逐个解决。
- 面向对象：一切皆对象，每个对象有自己的属性和方法，由他们去交互完成操作。
- 大象装进冰箱：面向过程->打开冰箱门，放入大象，关上冰箱门（都是一些函数）；面向对象->抽象出大象和冰箱两个对象，有各自的方法，由他们进行交互完成。
- interface最好由使用方去实现，而不是定义方去实现（go官方code review），这样我们只需定义使用到的方法，而不需要实现所有方法（不然不能编译）。在使用接口前不要定义接口，如果没有实际的使用示例，很难看到接口是否必要，更别说包含的方法了。
- go的map本身不支持并发操作，如果对其尝试进行并发读写错误。会抛出fatal error，很严重会杀死进程，也不能被recover捕获。
- 读已关闭的channel，如果里面没有数据了，会读到对应类型的零值，因此判断一个channel是否关闭，需要通过bool；而写已关闭的channel则会Panic。此外channel只能关闭一次，多次关闭会Panic。而对于未使用make初始化的channel，读写操作会死锁（这个channel是nil，永远不会往里面去读写）。
- 对于无缓冲的channel，必须要先启动一个goroutine来接收数据，再发送数据，否则会死锁。反之亦然。
- 读写channel默认都是阻塞模式，只有在select语句组成的多路复用分支中，channel的交互会变成非阻塞模式。非阻塞模式下，channel通过一个bool来标识是否读取/写入成功。所有需要当前协程被挂起/死锁的操作，非阻塞模式都会返回false；所有能立即完成读取/写入操作的条件下，非阻塞模式都会返回true。
- select类似io多路复用的select模型，结合channel可以同时监听多个分支，一旦对应的channel可读，会进入对应分支去执行相应的逻辑。
- channel核心数据结构
    1. lock，锁，本身就支持并发
    2. buf，channel中的元素队列（环形数组，有读写指针，用数组寻址更快，并且保证数据地址连续，内存局部性原理，减少io）
    3. recvx，读取元素的index
    4. sendx，写入元素的index
    5. account，channel中存在的元素容量
    6. dataqsize，channel中的元素容量
    7. elemsize，channel中的元素类型大小
    8. elemtype，channel中的元素类型
    9. closed，标识channel是否关闭
    10. recvq，阻塞的读协程队列
    11. sendq，阻塞的写协程队列

- recvq和sendq都是链表（链表添加和删除元素是O(1)，结构体是对协程g的再封装，有个isselect的bool值标识是否需要被阻塞。
- struct {} 一般仅仅做信号通知传达的意图，编译器会为其分配0内存。
- channel的类型可分为，主要体现在分配内存不同（大小，是否连续）
  - 无缓冲（make时就没分配大小，或者channel元素类型是struct {}）
  - 有缓冲struct型
  - 有缓冲pointer型
- channel中有多个阻塞的读协程（此时缓冲区必然是空的，不然不会出现读协程阻塞），此时有个写协程往里面写会发生什么?
  - lock
  - 将元素直接写入阻塞读协程（memmove），而不是再去写缓冲区了
  - unlock
  - goready（与使用gopark去主动阻塞协程相对应）唤醒阻塞读协程
- channel中无阻塞读协程，并且环形缓冲区仍然有空间
  - lock
  - 将需要写入的数据写入当前的写指针位置，再讲写指针位置+1，由于是环形缓冲区，到队尾会回到队头
  - unlock
- channel中无阻塞读协程，而环形缓冲区无空间
  - lock
  - 加入写协程阻塞队列中
  - gopark () 被阻塞直到被读协程唤醒并且读取该协程的数据后
  - 回收资源（注意这里是读协程主动去读了该协程的数据）
- 协程读时有阻塞的写协程（环形缓冲区此时肯定已满）
  - lock
  - 从环形缓冲区中取出数据（遵循fifo，环形缓冲区的数据肯定是比阻塞写协程更早到达）
  - 将首个阻塞的写协程的数据移入到环形缓冲区的尾部
  - goready唤醒阻塞的写协程
  - unlock
- 协程读时有阻塞的读协程（环形缓冲区此时肯定为空）
  - lock
  - 加入读阻塞队列
  - unlock
  - gopark阻塞直到被写协程唤醒
- 关闭channel
  - 关闭未初始化的channel会panic
  - lock
  - 重复关闭channel会panic
  - 将阻塞读协程队列中的协程节点统一添加到glist（不可能同时出现阻塞的读协程和写协程，两者是交互的）
  - 将阻塞写协程队列中的协程节点统一添加到glist
  - 唤醒glist中的协程
- 在关闭channel时需要保证没有阻塞的写协程存在，否则会Panic。

并发安全的map实现

```
package myconcurrentmap
```

```
import (
    "context"
    "sync"
    "time"
)

// MyConcurrentMap 是一个并发安全的 map 实现
// 支持插入和查询操作，查询时若 key 不存在会阻塞直到 key 被插入或超时
type MyConcurrentMap struct {
    sync.RWMutex
    mp      map[int]int          // 存储 key-value 对
    keyToch map[int]chan struct{} // 存储每个 key 对应的 channel，用于阻塞等待
}

// NewMyConcurrentMap 创建并返回一个新的 MyConcurrentMap 实例
func NewMyConcurrentMap() *MyConcurrentMap {
    return &MyConcurrentMap{
        mp:      make(map[int]int),
        keyToch: make(map[int]chan struct{}),
    }
}

// Put 插入一个 key-value 对到 map 中
// 如果有 goroutine 正在等待这个 key，则唤醒它们
func (m *MyConcurrentMap) Put(k, v int) {
    m.Lock()         // 加锁，确保并发安全
    defer m.Unlock() // 确保函数退出时解锁

    m.mp[k] = v // 插入 key-value 对

    // 检查是否有 goroutine 正在等待这个 key
    if ch, ok := m.keyToch[k]; ok {
        close(ch)            // 关闭 channel，唤醒所有等待的 goroutine
        delete(m.keyToch, k) // 从 map 中删除这个 channel
    }
}

// Get 查询 map 中的 key 对应的 value
// 如果 key 存在，直接返回 value
// 如果 key 不存在，阻塞直到 key 被插入或超时
```

```
func (m *MyConcurrentMap) Get(k int, maxWaitingDuration time.Duration) (int, error) {
    m.RLock()         // 加读锁，允许并发读
    v, ok := m.mp[k]  // 查询 key 是否存在
    if ok {
        m.RUnlock() // 如果 key 存在，解锁并返回 value
        return v, nil
    }

    // 如果 key 不存在，检查是否已有 goroutine 在等待这个 key
    ch, exists := m.keyToch[k]
    if !exists {
        // 如果没有，创建一个新的 channel 并存储到 map 中
        ch = make(chan struct{})
        m.keyToch[k] = ch
    }
    m.RUnlock() // 解锁，允许其他操作

    // 创建一个带超时的上下文
    tCtx, cancel := context.WithTimeout(context.Background(), maxWaitingDuration)
    defer cancel() // 确保超时上下文被取消

    // 等待 key 被插入或超时
    select {
    case <-tCtx.Done():
        // 超时，返回错误
        return -1, tCtx.Err()
    case <-ch:
        // key 被插入，继续执行
    }

    // 被唤醒后，再次查询 key 对应的 value
    m.RLock()
    defer m.RUnlock() // 确保函数退出时解锁
    v, ok = m.mp[k]
    if !ok {
        // 理论上不应该发生，因为 channel 被关闭意味着 key 已被插入
        return -1, context.DeadlineExceeded
    }
    return v, nil
}
```