## 并发：锁，atomic，once，sync.pool,waitGroup

## waitGroup



waitGroup是等待一组并发操作完成的好方法，将waitGroup视为一个并发-安全的计数器，add增加技术，done减少计数。Wait阻塞，直到计数为0。

## Once

## Cond

## Mutex

实现原理：<https://my.oschina.net/renhc/blog/2876211>



· Mutex 为互斥锁，Lock() 加锁，Unlock() 解锁

· 在一个 goroutine 获得 Mutex 后，其他 goroutine 只能等到这个 goroutine 释放该 Mutex

· 使用 Lock() 加锁后，不能再继续对其加锁，直到利用 Unlock() 解锁后才能再加锁

· **在 Lock() 之前使用 Unlock() 会导致 panic 异常**

· 已经锁定的 Mutex 并不与特定的 goroutine 相关联，这样可以利用一个 goroutine 对其加锁，再利用其他 goroutine 对其解锁

·  **在同一个 goroutine 中的 Mutex 解锁之前再次进行加锁，会导致死锁**

· 适用于读写不确定，并且只有一个读或者写的场景

## RWMutex

实现原理：<https://my.oschina.net/renhc/blog/2878292/print>

· RWMutex 是单写多读锁，该锁可以加多个读锁或者一个写锁

· 读锁占用的情况下会阻止写，不会阻止读，多个 goroutine 可以同时获取读锁

· 写锁会阻止其他 goroutine（无论读和写）进来，整个锁由该 goroutine 独占

· 适用于读多写少的场景

### Lock() 和 Unlock()

· Lock() 加写锁，Unlock() 解写锁

· 如果在加写锁之前已经有其他的读锁和写锁，则 Lock() 会阻塞直到该锁可用，为确保该锁可用，已经阻塞的 Lock() 调用会从获得的锁中排除新的读取器，即写锁权限高于读锁，有写锁时优先进行写锁定

· 在 Lock() 之前使用 Unlock() 会导致 panic 异常

### RLock() 和 RUnlock()

· RLock() 加读锁，RUnlock() 解读锁

· RLock() 加读锁时，如果存在写锁，则无法加读锁；当只有读锁或者没有锁时，可以加读锁，读锁可以加载多个

· RUnlock() 解读锁，RUnlock() 撤销单词 RLock() 调用，对于其他同时存在的读锁则没有效果

· 在没有读锁的情况下调用 RUnlock() 会导致 panic 错误

· RUnlock() 的个数不得多余 RLock()，否则会导致 panic 错误

### 写操作是如何阻止写操作的

写操作依赖互斥锁阻止其他的写操作

### 写操作是如何阻止读操作的

写操作将readerCount变成负值来阻止读操作的

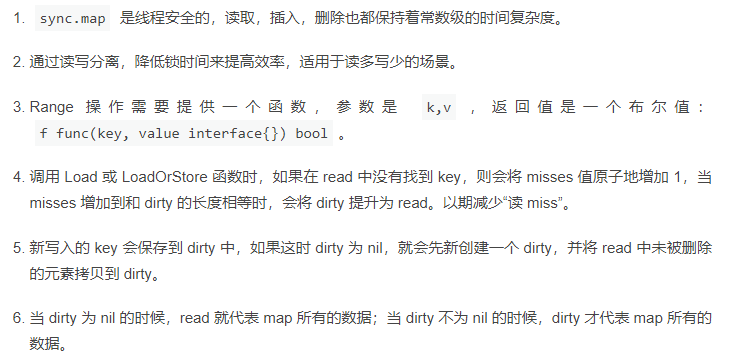
### 写操作将readerCount变成负值来阻止读操作的

读操作通过readerCount来将来阻止写操作的

### 为什么写锁定不会被饿死

写操作到来时，会把RWMutex.readerCount值拷贝到RWMutex.readerWait中，用于标记排在写操作前面的读者个数。前面的读操作结束后，除了会递减RWMutex.readerCount，还会递减RWMutex.readerWait值，当RWMutex.readerWait值变为0时唤醒写操作。

## sync.Map



## Context控制多个goroutine

