# 深入理解并发安全的sync.Map

## Map和sync.Map的比较

Golang中内置map关键字，但是是非线程安全的，故加入sync.Map，提供并发安全的map。







但是当map的数据量非常大的时候，会引发大量goroutine争夺同一把锁，这种现象将直接导致性能的急剧下降。有点类似于java中的hashMap和concurrentHashMap。







## 2、load



对dirty加锁，对read不加锁，减少锁的争用。原理就是只将数据锁一段时间，也就是只锁dirty，当dirty满了，那么就拷贝read，read是不加锁的。而满没满是看misses值是否小于les(m.dirty)。而且因为不管有没有从dirty中拿到数据，misses都会自加1，所以保证了数据不会长时间加锁。对数据加锁的方式就是，锁掉读数据的代码，这样就不会存在多个goroutine同时读数据了。而且会对read，进行两次读，防止在对读dirty数据的代码加锁的时候，dirty中的数据已经拷贝到read了。



## 3、Store



Read中是否有key，

有且未标记删除，直接修改指针的val值。

有但标记删除，插入dirty，修改read的key的指针指向即可。

Read中没有key

dirty中有该键，更新val

dirty中也没有，将read中赋值未删除的数据到dirty（保持dirty中数据是最新的），然后再在dirty中用newEntry新增entry。



## 4、Range



## **5、Sync.Map的使用例子**



## **适用场景**

Sync.Map并不是为了代替锁+map的组合。

两种情况应该选择sync.Map

1. key值一次写入，多次读取，因为总是会double check read
2. 多个goroutine的读取、写入和覆盖在不相交的key集。

# 原子操作和互斥锁的区别

原子操作即是进行过程中不能被终端的操作，针对某个值的原子操作在被进行的过程中，cpu绝不会再去进行其他的针对该值的操作。为了实现这样的严谨行，原子操作仅会由一个独立的cpu指令代表和完成。原子操作是无锁的，常常通过cpu指令直接实现，事实上，其他同步及时的实现常常依赖与原子操作。

## 1、Go对原子操作的支持：

Go语言的sync/atomic包提供了对原子操作的支持，用于同步访问整数和指针。

五种：增减、比较并交换、载入、存储、交换。

原子操作支持的类型包括int32，int64、uint32、uint64、uintptr、unsafe.Pointer

下面演示两种原子操作的实例：

AddInt32



CompareAndSwarp



## **2、原子操作和互斥锁区别：**

首先二者均不可被其他线程打断，

Atomic操作的优势是更轻量，比如CAS可以在不行成临界区和创建互斥量的情况下完成并发安全的值替换操作。这可以大大的减少同步对程序性能的损耗。

当然也有劣势，趋于乐观，总是假设被操作值未被改变，并一段确认这个假设的真实性就立即进行值替换，那么在被操作值被频繁变更的情况下，CAS操作并不容易成功。相反，互斥锁的做法趋于悲观，总假设会有并发的操作要修改被操作的值，并用锁将相关操作放入临界区进行保护。

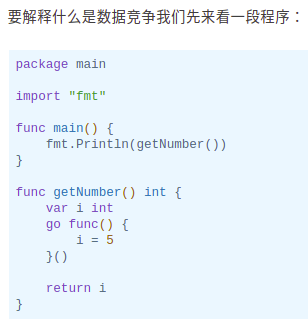
总结：

1. 互斥锁是一种数据结构，用来让一个线程执行程序的关键不分，完成互斥的多个操作。
2. 原子操作是针对某个值的单个互斥操作。
3. 互斥锁可以理解为悲观锁，共享资源每次只给一个线程使用，其他线程阻塞，用完后再把资源转让给其他线程。

# Go并发变成里的数据竞争以及解决之道

Go语言以容易进行并发变成而闻名。

## 数据竞争



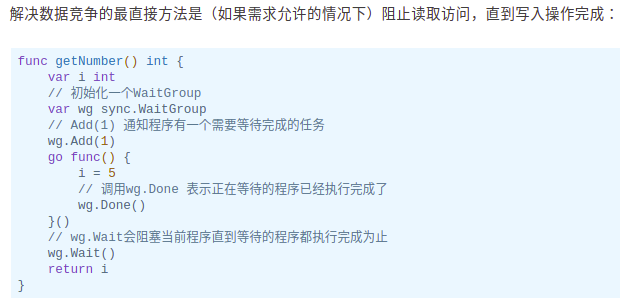
这段代码有两个携程：

1. 主携程：返回i
2. 自定义携程：设置i

那么可能返回的i值是0或5，那么这就产生了数据竞争。

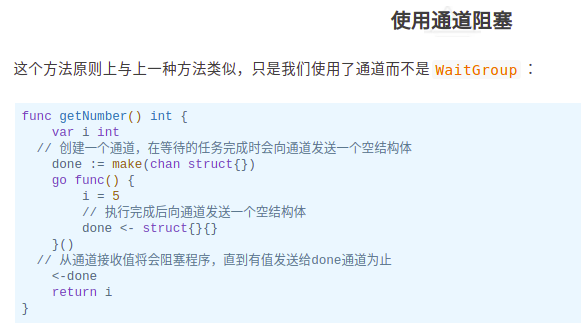
## 2、解决数据竞争

### 1、使用WaitGroup



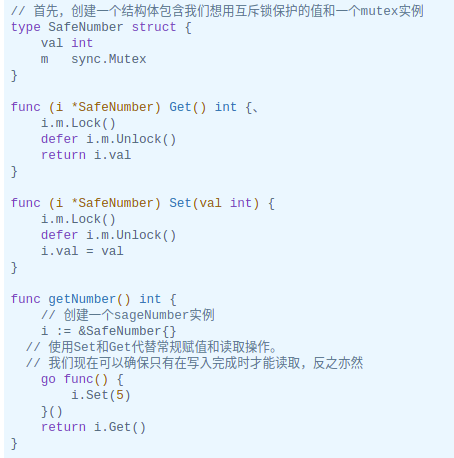
## 2、使用通道阻塞

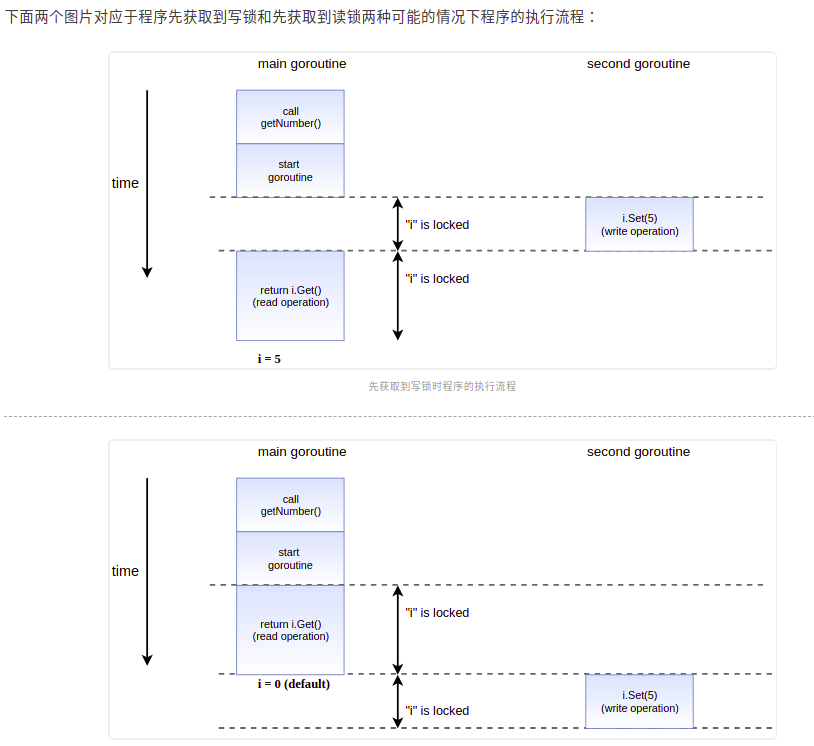
通道中如果没有值，但如果你要从通道里取值，那么会阻塞程序，直到有值取出，我们可以在我们期待的优先执行的goroutine中给通道存值，后运行的goroutine中取值来实现避免数据竞争。



## 使用Mutex

上诉两种方法，保证的是先写入后读取，但假如不需要保证顺序，只需要保证二者是互斥的，不能同时发生即可，那么应该考虑使用Mutex互斥锁。





## mutex Vs channel

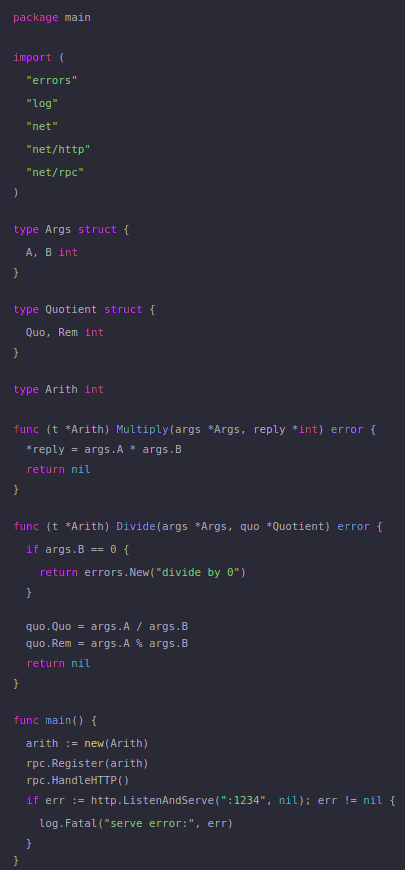
大多数新手都视图用通道解决所有并发问题，这是go语言的一个很酷的特性，这是不对的，二者都各有好处。

通常，当goroutine需要相互通信时使用通道，当确保只有一个goroutin能访问代码的关键部分时使用互斥锁。比如这个问题，互斥锁更为适合，你在通道里面传入的空结构体并没有任何通信的作用。

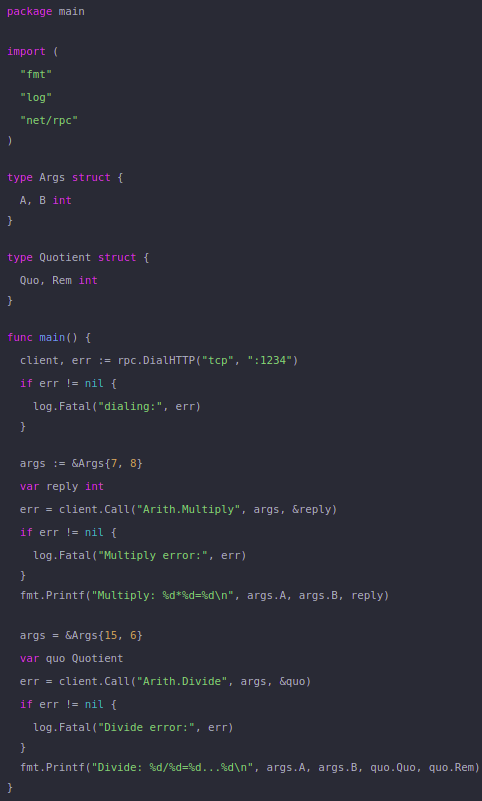
# GO标准库net/rpc

## 基于HTTP RPC同步调用：

服务端：

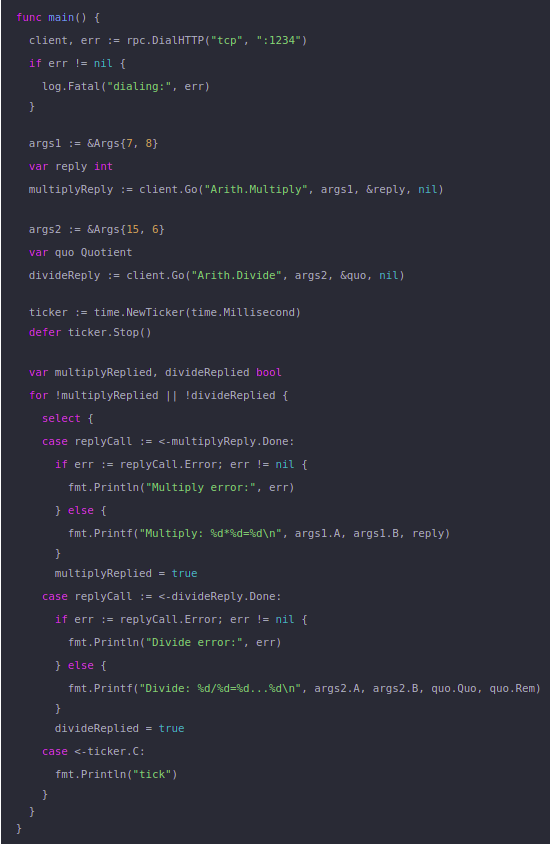


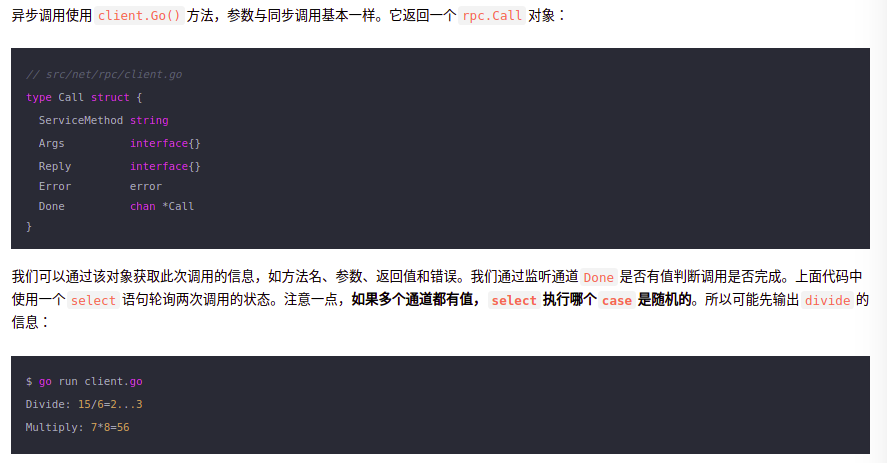
客户端：



## 基于HTTP RPC异步调用：

客户端：





## TCP



## 定制方法名

