Lecture 5 Go语言内存模型

1. 背景知识

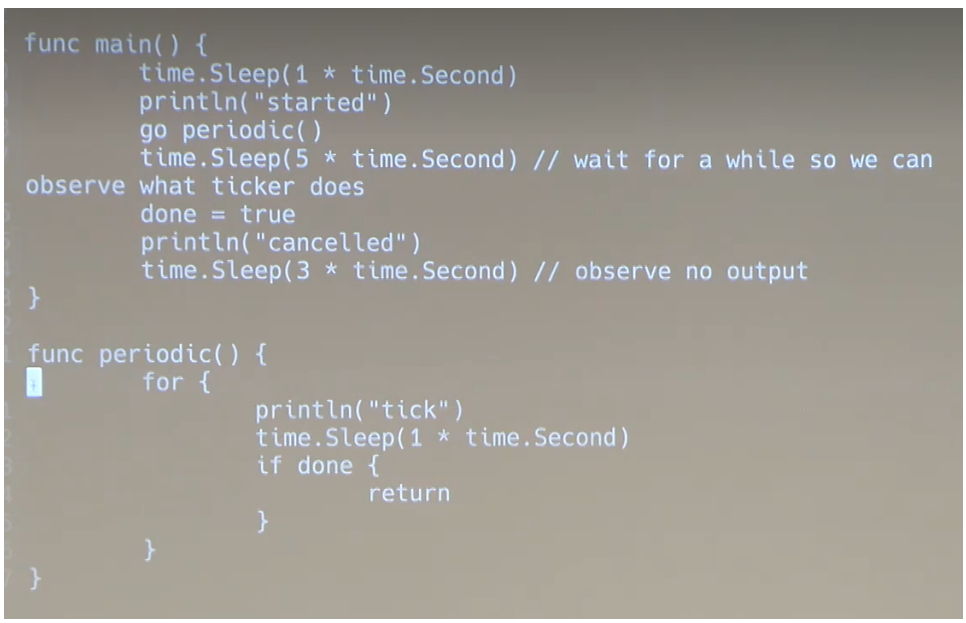
1.1. 乱序执行

- 现代CPU采用多核、多线程技术以提升计算能力；采用乱序执行、流水线、分支预测以及多级缓存等方法来提升程序性能。多核技术在提升程序性能的同时，也带来了执行序列乱序和内存序列访问的乱序问题。与此同时，编译器也会基于自己的规则对代码进行优化，这些优化动作也会导致一些代码的顺序被重排
- 这些重排的代码在单线程程序上可能不会存在问题，但是面对多线程并发，并且线程之间存在共享数据的时候，就有可能发生并发错误
- 可以使用锁机制。但是锁机制需要操作系统的支持，上锁和解锁需要系统调用。锁操作需要在用户态和核心态之间切换，开销大。因此，需要语言层面对多线程数据访问的支持，也就是多线程内存模型

2.

3. 写多线程程序的原则

1. 并发进程访问共享变量时，必须要上锁！以避免编译器进行优化或者CPU的乱序执行。



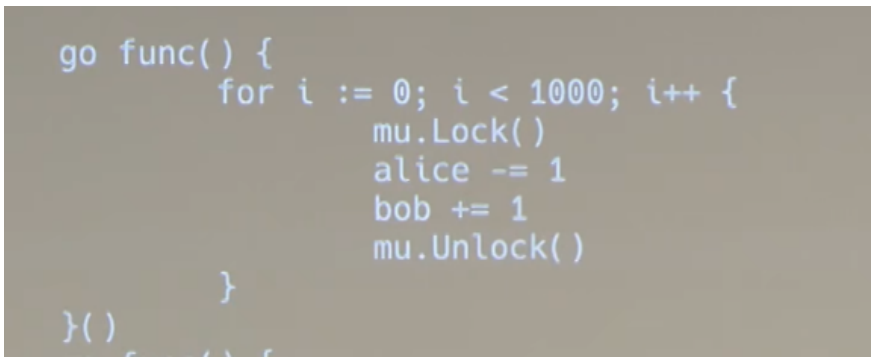
```
func main() {
        time.Sleep(1 * time.Second)
        println("started")
        go periodic()
        time.Sleep(5 * time.Second) // wait for a while so we can
observe what ticker does
        done = true
        println("cancelled")
        time.Sleep(3 * time.Second) // observe no output
}

func periodic() {
        for {
                println("tick")
                time.Sleep(1 * time.Second)
                if done {
                        return
                }
        }
}
```

在上图的例子中，编译器可能会将if done判断移动到for循环外面（从单线程程序的角度来看）。因此很有可能会使得periodic陷入死循环

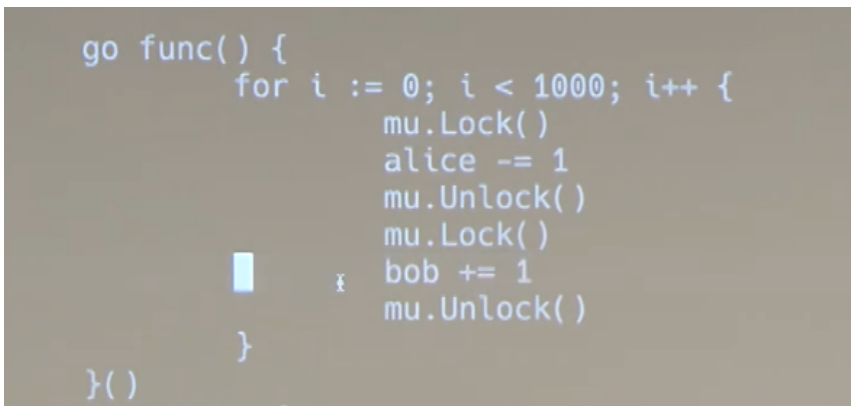
2. 编写多线程程序应当尽量减少使用channel。除非在类似于生产者-消费者的情景之下。channel可以用来实现信号量

3. 不仅要在使用共享变量时需要使用同步工具，还需要考虑线程并发执行过程中的不变量。例如银行账户总额。

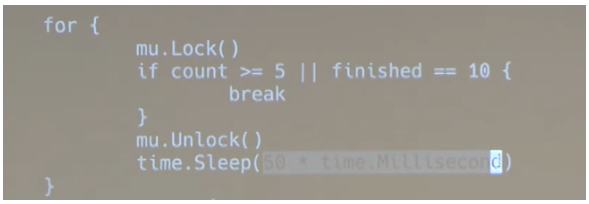
   下面的情况可以维护不变量，保证银行账户总额始终不变：



   下面的情况则不能维护不变量，不变量在并发过程中会有错误，但是能保证在所有线程执行完毕之后是正确的：



4. 在有master 线程的情况下，master线程可能需要忙等待其他线程的结果，如果死循环轮询，可能会占用过高的CPU，解决这一问题只需要在轮询期间加入sleep即可：



   但这里用到了sleep时间，是一个magic number/arbitrary number，可以用条件变量来更好的实现

4. 线程同步工具

- WaitGroup
- Channel
- mutex lock
- 条件变量
  - 固定写法如下：

```
mu.Lock()
// do something that might affect the condition
cond.Broadcast()
mu.Unlock()

----

mu.Lock()
while condition == false {
        cond.Wait()
}
// now condition is true, and we have the lock
mu.Unlock()
```

```
package main

import "sync"
import "time"
import "math/rand"

func main() {
        rand.Seed(time.Now().UnixNano())

        count := 0
        finished := 0
        var mu sync.Mutex
        cond := sync.NewCond(&mu)

        for i := 0; i < 10; i++ {
                go func() {
                        vote := requestVote()
                        mu.Lock()
                        defer mu.Unlock()
                        if vote {
                                count++
                        }
                        finished++
                        cond.Broadcast()
```

  - 广播必须要在释放锁的内部
  - 条件变量需要与锁绑定
  - master线程首先获取锁，而后在不满足条件时会挂起。条件变量会维护一个等待队列，当条件变量运行了广播方法时，所有被挂起在该条件变量上的线程都会激活，重新判断

条件（会自动获得锁），挂起时又会自动释放锁