Select和线程安全

作者：少林之巅

目录

select语义介绍和使用

1. 多channel场景

   A. 多个channel同时需要读取或写入，怎么办?

   B. **串行操作?**　NONONO

select语义介绍和使用

```
package main
import (
    "fmt"
    "time"
)
func server1(ch chan string) {
    time.Sleep(6 * time.Second)
    ch <- "from server1"
}
func server2(ch chan string) {
    time.Sleep(3 * time.Second)
    ch <- "from server2"

}
func main() {
    output1 := make(chan string)
    output2 := make(chan string)
    go server1(output1)
    go server2(output2)
    s1 := <-output1
    fmt.Println(s1)
    s2 := <-output2
    fmt.Println(s2)
}
```

select语义介绍和使用

3. select登场

A. 同时监听一个或多个channel，直到其中一个channel ready

B. 如果其中多个channel同时ready，随机选择一个进行操作。

C. 语法和switch case有点类似，代码可读性更好。

```
select {
    case s1 := <-output1:
        fmt.Println(s1)
    case s2 := <-output2:
        fmt.Println(s2)
}
```

select语义介绍和使用

```
package main

import (
    "fmt"
    "time"
)

func server1(ch chan string) {
    time.Sleep(6 * time.Second)
    ch <- "from server1"
}
func server2(ch chan string) {
    time.Sleep(3 * time.Second)
    ch <- "from server2"

}
func main() {
    output1 := make(chan string)
    output2 := make(chan string)
    go server1(output1)
    go server2(output2)
    select {
    case s1 := <-output1:
        fmt.Println(s1)
    case s2 := <-output2:
        fmt.Println(s2)
    }
}
```

select语义介绍和使用

4. default分支，当case分支的channel都没有ready的话，执行default

   A. 用来判断channel是否满了

   B. 用来判断channel是否是空的

select语义介绍和使用

4. default分支，当case分支的channel都没有ready的话，执行default

   A. 用来判断channel是否满了

   B. 用来判断channel是否是空的

select语义介绍和使用

5. select case分支随机策略验证

```
package main

import (
    "fmt"
    "time"
)

func server1(ch chan string) {
    ch <- "from server1"
}
func server2(ch chan string) {
    ch <- "from server2"

}
func main() {
    output1 := make(chan string)
    output2 := make(chan string)
    go server1(output1)
    go server2(output2)
    time.Sleep(1 * time.Second)
    select {
    case s1 := <-output1:
        fmt.Println(s1)
    case s2 := <-output2:
        fmt.Println(s2)
    }
}
```

select语义介绍和使用

6. empty select

```
package main

func main() {
    select {
    }
}
```

线程安全介绍

7. 现实例子

   A. 多个goroutine同时操作一个资源，这个资源又叫临界区

   B. 现实生活中的十字路口，通过红路灯实现线程安全

   C. 火车上的厕所，通过互斥锁来实现线程安全

线程安全介绍

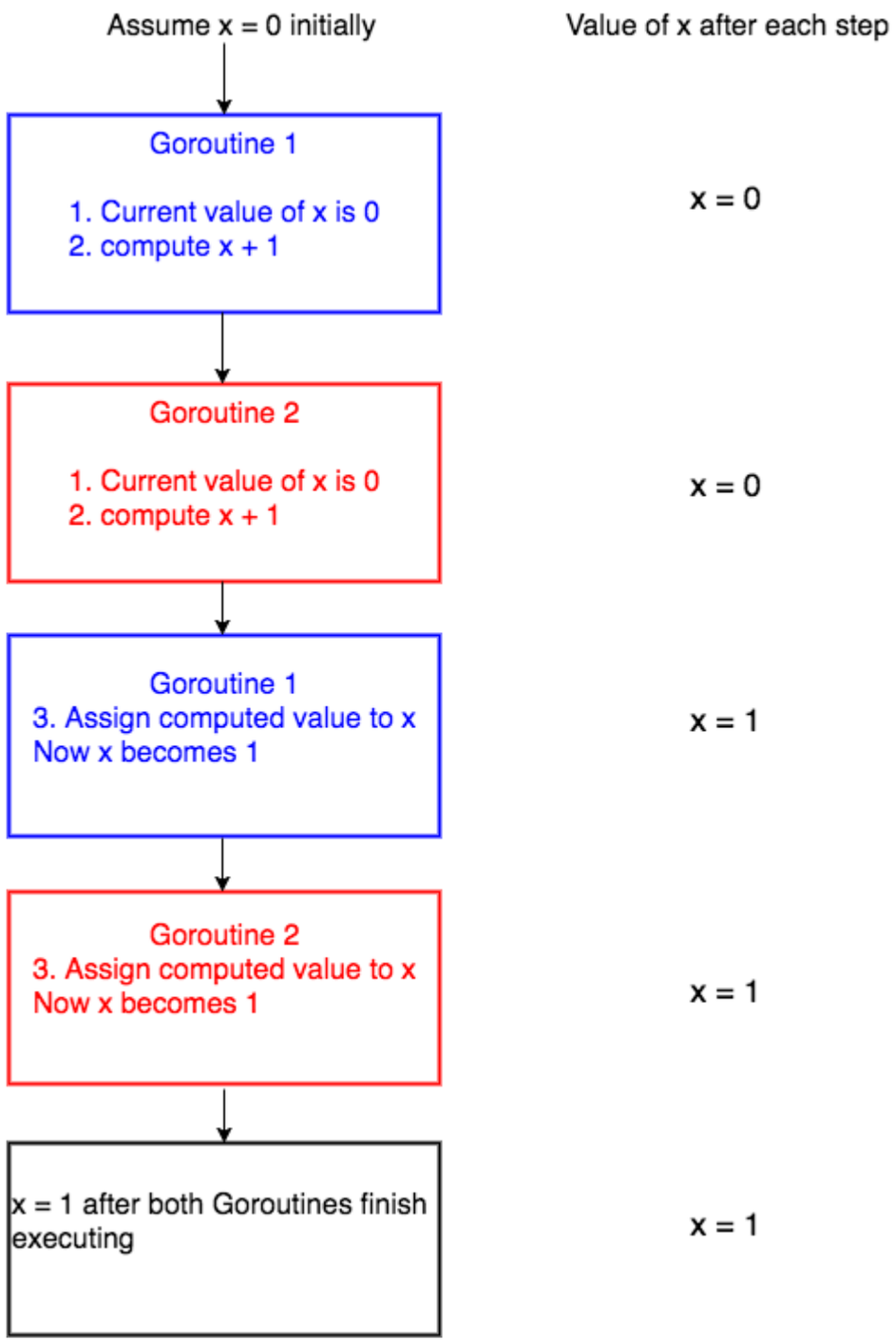
8. 实际例子， x = x +1

   A. 先从内存中取出x的值

   B. CPU进行计算，x+1

   C. 然后把x+1的结果存储在内存中

线程安全介绍


Assume x = 0 initially
Value of x after each step
Goroutine 1
1. Current value of x is 0
2. compute x + 1
x = 0
Goroutine 2
1. Current value of x is 0
2. compute x + 1
x = 0
Goroutine 1
3. Assign computed value to x
Now x becomes 1
x = 1
Goroutine 2
3. Assign computed value to x
Now x becomes 1
x = 1
x = 1 after both Goroutines finish executing
x = 1

线程安全介绍

9. 互斥锁介绍

A. 同时有且只有一个线程进入临界区，其他的线程则在等待锁

B. 当互斥锁释放之后，等待锁的线程才可以获取锁进入临界区

C. 多个线程同时等待同一个锁，唤醒的策略是随机的

线程安全介绍

有问题的代码!

```
package main
import (
    "fmt"
    "sync"
    )
var x  = 0
func increment(wg *sync.WaitGroup) {
    x = x + 1
    wg.Done()
}
func main() {
    var w sync.WaitGroup
    for i := 0; i < 1000; i++ {
        w.Add(1)
        go increment(&w)
    }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

线程安全介绍

使用互斥锁fix

```
package main
import (
    "fmt"
    "sync"
    )
var x  = 0
func increment(wg *sync.WaitGroup, m *sync.Mutex) {
    m.Lock()
    x = x + 1
    m.Unlock()
    wg.Done()
}
func main() {
    var w sync.WaitGroup
    var m sync.Mutex
    for i := 0; i < 1000; i++ {
        w.Add(1)
        go increment(&w, &m)
    }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

读写锁介绍

10. 读写锁使用场景

    A. 读多写少的场景

    B. 分为两种角色，读锁和写锁

    C. 当一个goroutine获取写锁之后，其他的goroutine获取写锁或读锁都会等待

    D. 当一个goroutine获取读锁之后，其他的goroutine获取写锁都会等待, 但其他goroutine获取读锁时，都会继续获得锁.

读写锁介绍

11.读写锁案例演示

读写锁介绍

12. 读写锁和互斥锁性能比较

原子操作

13. 原子操作

A. 加锁代价比较耗时，需要上下文切换

B. 针对基本数据类型，可以使用原子操作保证线程安全

C. 原子操作在用户态就可以完成，因此性能比互斥锁要高

原子操作

14. 原子操作介绍

```
func AddInt32(addr *int32, delta int32) (new int32)
func AddInt64(addr *int64, delta int64) (new int64)
func AddUint32(addr *uint32, delta uint32) (new uint32)
func AddUint64(addr *uint64, delta uint64) (new uint64)
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
```

加减操作

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)
```

比较并交换

```
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
```

读取操作

```
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
```

写入操作

```
func SwapInt32(addr *int32, new int32) (old int32)
func SwapInt64(addr *int64, new int64) (old int64)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
func SwapUint32(addr *uint32, new uint32) (old uint32)
func SwapUint64(addr *uint64, new uint64) (old uint64)
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

交换操作