Channels for Data Protection

Cody Soyland
Austin Go Language User Group
October 17, 2013

How do I share memory among goroutines?

Do not communicate by sharing memory; instead, share memory by communicating.

-Andrew Gerrand

Unsafe programs are a rat’s nest.

Example: sharing a counter variable

```go
package main

import (
    "fmt"
    "sync"
    "runtime"
)

var counter int
var waitgroup sync.WaitGroup

func main() {
    runtime.GOMAXPROCS(4)
    counter = 0
    waitgroup = sync.WaitGroup{}
    waitgroup.Add(1000)
    for i:=0; i<1000; i++ {
        go increment()
    }
    waitgroup.Wait()
    fmt.Println(counter)
}

func increment() {
    counter = counter + 1
    waitgroup.Done()
}
```

Don’t do this.

```
> go run 1-unsafe.go
1000
> go run 1-unsafe.go
1000
> go run 1-unsafe.go
998
> go run 1-unsafe.go
999
> go run 1-unsafe.go
1000
```

```
    package main

    import (
        "fmt"
        "sync"
        "runtime"
    )
    var counter int
    var waitgroup sync.WaitGroup
⟶   var mutex sync.Mutex

    func main() {
        runtime.GOMAXPROCS(4)
        counter = 0
        waitgroup = sync.WaitGroup{}
        waitgroup.Add(1000)
        for i:=0; i<1000; i++ {
            go increment()
        }
        waitgroup.Wait()
        fmt.Println(counter)
    }

    func increment() {
⟶       mutex.Lock()
        counter = counter + 1
⟶       mutex.Unlock()
        waitgroup.Done()
    }
```

Let’s try a mutex.

```
> go run 2-mutex.go
1000
> go run 2-mutex.go
1000
> go run 2-mutex.go
1000
> go run 2-mutex.go
1000
> go run 2-mutex.go
1000
```

How about a token-exchange channel?

```
package main

import (
     "fmt"
     "sync"
     "runtime"
)

var counter int
var waitgroup sync.WaitGroup
var token chan int

func main() {
     runtime.GOMAXPROCS(4)
     counter = 0
     waitgroup = sync.WaitGroup{}
     token = make(chan int)
     waitgroup.Add(1000)
     for i:=0; i<1000; i++ {
          go increment()
     }
     token <- 1
     go func() {<-token}()
     waitgroup.Wait()
     fmt.Println(counter)
}

func increment() {
     <-token
     counter = counter + 1
     token <- 1
     waitgroup.Done()
}
```

```
package main
import (
    "fmt"
    "runtime"
)
var counter int
var inc chan int
var done chan int

func main() {
    runtime.GOMAXPROCS(4)
    counter = 0
    inc = make(chan int)
    done = make(chan int)
    go incrementor()
    for i:=0; i<1000; i++ {
        go increment()
    }
    <-done
    fmt.Println(counter)
}

func increment() {
    inc <- 1
}

func incrementor() {
    for i:=0; i<1000; i++ {
        <-inc
        counter = counter + 1
    }
    done<-1
}
```

Ownership goroutine!

No more sync.WaitGroup

Callbacks! just for lulz

```go
package main

import (
    "fmt"
    "runtime"
)

var counter int
var inc chan func(int) int
var done chan int

func main() {
    runtime.GOMAXPROCS(4)
    counter = 0
    inc = make(chan func(int) int)
    done = make(chan int)

    go incrementor()
    for i:=0; i<1000; i++ {
        inc <- func(count int) int {
            return count + 1
        }
    }

    <-done
    fmt.Println(counter)
}

func incrementor() {
    for i:=0; i<1000; i++ {
        callback := <-inc
        counter = callback(counter)
    }
    done<-1
}
```

```go
package main

import (
    "fmt"
    "runtime"
)

var grant chan int
var reclaim chan int
var done chan int

func main() {
    runtime.GOMAXPROCS(4)
    grant = make(chan int)
    reclaim = make(chan int)
    done = make(chan int)
    go incrementor()

    grant <- 0

    for {
        select {
        case counter := <-reclaim:
            grant <- counter
        case counter := <-done:
            fmt.Println(counter)
            return
        }
    }
```

```go
func incrementor() {
    for i:=0; i<1000; i++ {
        counter := <-grant
        counter = counter + 1
        reclaim <- counter
    }
    done <- <-grant
}
```

No need to have global counter

A more realistic example

- Concurrency-safe map
- Uses commands over channels

```
type Command interface {
  Call(map[string]string)
}
```

```
type GetCommand struct {
   name string
   retchan chan string
}
func (c GetCommand) Call(m map[string]string) {
   c.retchan <- m[c.name]
}

type SetCommand struct {
   name string
   value string
}
func (c SetCommand) Call(m map[string]string) {
   m[c.name] = c.value
}
```

```
func NewSafeMap() *SafeMap {
    m := SafeMap{}
    m.commandchan = make(chan Command)
    return &m
}

func (m *SafeMap) Run() {
    kv := make(map[string]string)
    for {
        command := <-m.commandchan
        command.Call(kv)
    }
}
```

```go
type SafeMap struct {
   commandchan chan Command
}

func (m *SafeMap) Get(name string) string {
   retchan := make(chan string)
   m.commandchan <- GetCommand{name, retchan}
   return <- retchan
}

func (m *SafeMap) Set(name, value string) {
   m.commandchan <- SetCommand{name, value}
}
```

```go
func main() {
   runtime.GOMAXPROCS(4)
   m := NewSafeMap()
   go m.Run()
   go m.Set("itchy", "scratchy")
   go m.Set("herp", "derp")

   time.Sleep(time.Second)
   fmt.Println(m.Get("itchy"))
}
```

Thanks! Questions?