MASTERING GO FOR UNIX ADMINISTRATORS, UNIX DEVELOPERS AND WEB DEVELOPERS

Day 2

Mastering Go for UNIX administrators, UNIX developers and Web Developers

3h 45m

Topics

1. Go Channels
2. Shared memory
3. Shared variables
4. Concurrent TCP server
5. Go Package versioning

QUESTIONS FROM YESTERDAY?

EXPECT LOTS OF GO CODE TODAY!

Section 4

Go Channels revisited

Types of channels

Channels in Go play a key role and they are not for creating pipelines only.

Zero values: For integers the zero value is 0. For a structure, the zero value is a structure where each one of its fields has the zero value of its type. For the channel type, the zero value is **nil**.

- Nil channels
- Buffered channels
- Channel of channels
- Signal channels
- Using channels for sharing data

The **select** statement is really important!

The select statement

The select statement

The **select** statement in Go looks like a **switch** statement for channels.
In practice, this means that select allows a goroutine to **wait on multiple communications operations**. Therefore, the main benefit that you receive from select is that it gives you the power to work with **multiple channels** using a **single select block**.
As a consequence, you can have nonblocking operations on channels.

A **select** statement is not evaluated sequentially, as all of its channels are examined simultaneously. If none of the channels in a select statement is ready, the select statement will block until one of the channels is ready. If multiple channels of a select statement is ready, then the Go runtime will make a random selection over the set of these ready channels. The Go runtime tries to make this random selection between these ready channels as uniformly and as fairly as possible.

Using select

```
package main

import (
	"fmt"
	"math/rand"
	"os"
	"strconv"
	"time"
)

func gen(min, max int, createNumber chan int, end chan
bool) {
	for {
		select {
		case createNumber <- rand.Intn(max-min) + min:
		case <-end:
			close(end)
			return
		case <-time.After(4 * time.Second):
			fmt.Println("\ntime.After()!")
		}
	}
}
```

```
func main() {
	rand.Seed(time.Now().Unix())
	createNumber := make(chan int)
	end := make(chan bool)

	if len(os.Args) != 2 {
		fmt.Println("Please give me an integer!")
		return
	}

	n, _ := strconv.Atoi(os.Args[1])
	fmt.Printf("Going to create %d random numbers.\n", n)
	go gen(0, 2*n, createNumber, end)

	for i := 0; i < n; i++ {
		fmt.Printf("%d ", <-createNumber)
	}

	time.Sleep(5 * time.Second)
	fmt.Println("Exiting...")
	end <- true
}
```

Time out goroutines

TO: Technique 1

Using select

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    go func() {
        time.Sleep(time.Second * 3)
        c1 <- "c1 OK"
    }()

    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout c1")
    }

    c2 := make(chan string)
    go func() {
        time.Sleep(3 * time.Second)
        c2 <- "c2 OK"
    }()

    select {
    case res := <-c2:
        fmt.Println(res)
    case <-time.After(4 * time.Second):
        fmt.Println("timeout c2")
    }
}
```

TO: Technique 2

Using select

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

func timeout(w *sync.WaitGroup, t time.Duration)
bool {
    temp := make(chan int)
    go func() {
        time.Sleep(5 * time.Second)
        defer close(temp)
        w.Wait()
    }()

    select {
    case <-temp:
        return false
    case <-time.After(t):
        return true
    }
}
```

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)

    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    duration := time.Duration(int32(t)) * time.Millisecond
    fmt.Printf("Timeout period is %s\n", duration)

    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
    w.Done()

    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
}
```

Nil Channels

Nil channels

```
package main

func main() {
    var c chan string
    close(c)
}
```

```
Remember:

- Read from nil channel: blocks forever
- Write to nil channel: blocks forever
- Close a nil channel: panics
```

nilChannel.go

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
        case input := <-c:
            sum = sum + input
        case <-t.C:
            c = nil
            fmt.Println(sum)
            fmt.Println("add() ending...")
            return
        }
    }
}
```

```go
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
    fmt.Println("send() ending...")
}

func main() {
    c := make(chan int)
    go add(c)
    go send(c)

    time.Sleep(3 * time.Second)
}
```

Question: what is wrong with the code?

Buffered channels

Buffered Channels

Buffered channels are channels that allow the Go scheduler to put jobs in the queue quickly in order to be able to deal with more requests.
Moreover, you can use buffered channels as semaphores in order to limit the throughput of your application.
You can create a buffered channel as follows:

```
numbers := make(chan int, 5)
```

What is the size of that channel?

bufferedChannels.go

```go
package main

import (
    "fmt"
)

func main() {
    numbers := make(chan int, 5)
    numbers <- 0
    numbers <- 1
    numbers <- 2
    numbers <- 3
    numbers <- 4

    // This will not work
    // numbers <- 5

    // This will work
    select {
    case numbers <- 5:
    default:
        fmt.Println("Not enough space in the channel!")
    }

    // You can read a buffered channel as usual
    for i := 0; i < 10; i++ {
        select {
        case num := <-numbers:
            fmt.Println(num)
        default:
            break
        }
    }
}
```

Order of execution

Order

You will now learn a technique for specifying the **order of execution** for the goroutines of a program.

defineOrder.go

```go
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}

func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
    close(b)
}

func C(a chan struct{}) {
    <-a
    fmt.Println("C()!")
}

func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})

    go C(z)
    go A(x, y)
    go C(z)
    go B(y, z)
    go C(z)

    close(x)
    time.Sleep(3 * time.Second)
}
```

Worker pools

Worker pools

A **Worker Pool** is a set of threads that are about to process jobs assigned to them. More or less, the Apache web server works that way: the main process accepts all incoming requests that are forwarded to the worker processes for getting served. Once a worker process has finished its job, it is ready to serve a new client. Nevertheless, there is a central difference here because our worker pool will use goroutines instead of threads. Additionally, threads do not usually die after serving a request because the cost of ending a thread and creating a new one is too high, whereas goroutines do die after finishing their job.

As you will see shortly, worker pools in Go are implemented with the help of buffered channels, because they allow you to limit the number of goroutines running at the same time.

The code!

```go
func worker(w *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Data{c, square}
        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}

func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

```go
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    nWorkers, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }

    go create(nJobs)
    finished := make(chan interface{})
    go func() {
        for d := range data {
            fmt.Printf("Client ID: %d\tint: ", d.job.id)
            fmt.Printf("%d\tsquare: %d\n", d.job.integer, d.square)
        }
        finished <- true
    }()

    makeWP(nWorkers)
    fmt.Printf(": %v\n", <-finished)
}
```

Sharing data with a channel

monitor.go

```go
package main
import (
   "fmt"
   "math/rand"
   "os"
   "strconv"
   "sync"
   "time"
)
var readValue = make(chan int)
var writeValue = make(chan int)

func set(newValue int) {
   writeValue <- newValue
}

func read() int {
   return <-readValue
}

func monitor() {
   var value int
   for {
      select {
      case newValue := <-writeValue:
         value = newValue
         fmt.Printf("%d ", value)
      case readValue <- value:
      }
   }
}
```

```go
func main() {
   if len(os.Args) != 2 {
      fmt.Println("Please give an integer!")
      return
   }
   n, err := strconv.Atoi(os.Args[1])
   if err != nil {
      fmt.Println(err)
      return
   }

   fmt.Printf("Going to create %d random numbers.\n", n)
   rand.Seed(time.Now().Unix())
   go monitor()
   var w sync.WaitGroup

   for r := 0; r < n; r++ {
      w.Add(1)
      go func() {
         defer w.Done()
         set(rand.Intn(10 * n))
      }()
   }
   w.Wait()
   fmt.Printf("\nLast value: %d\n", read())
}
```

QUESTIONS SO FAR?

Lab 4

Go Channels in action

Channels

We are now going to see Go programs that use channels in action to better understand how they work.

Please be extra careful when working with channel because, when used wrongly, channels can panic your Go programs!

First, execute the next two programs on your computer:
- lab4/defineOrder.go
- lab4/closeNilChannel.go

Worker pools

workerPool.go

Execute **lab4/workerPool.go** on your computer.

Let me share my Terminal application in order to execute **lab4/workerPool.go**.

Buffered channels

bufferedChannels.go

Let me share my Terminal application in order to execute **lab4/ bufferedChannels.go** once more.
But first, let me give you some time to execute it on your UNIX shells.

Sharing data using channels

monitor.go

We will execute **monitor.go** now.

Section 5

Shared memory + variables

The sync.Mutex type

Mutex

Shared memory and **shared** variables are the most common ways for UNIX threads to communicate with each other.
A **mutex** works like a buffered channel of capacity one, which allows at most one goroutine to access a shared variable at any given time. This means that there is no way for two or more goroutines to try to update that variable simultaneously.

The definition of the **sync.Mutex** type, used in Go, is nothing extraordinary. All of the interesting work is being done by the **sync.Lock()** and **sync.Unlock()** functions that can lock and unlock a sync.Mutex mutex, respectively. Locking a mutex means that nobody else can lock it until it has been released using the **sync.Unlock()** function.

mutex.go

```
func change(i int) {
      m.Lock()
      time.Sleep(time.Second)
      v1 = v1 + 1
      if v1%10 == 0 {
            v1 = v1 - 10*i
      }
      m.Unlock()
}
```

The reason for protecting a write operation is that you do not want someone else to try to change the shared variable at the same time.

```
func read() int {
      m.Lock()
      a := v1
      m.Unlock()
      return a
}
```

The reason for protecting a read operation is that you do not want the value of the shared variable to change while you are reading it.

The sync.Mutex type is the Go implementation of a mutex. Its definition, which can be found in the mutex.go file of the sync directory, is as follows:

```
   // A Mutex is a mutual exclusion lock.
   // The zero value for a Mutex is an unlocked mutex.
   //
   // A Mutex must not be copied after first use.
type Mutex struct {
    state int32
    sema uint32
}
```

A critical section cannot be *embedded* in another critical section when both critical sections use the **same** *sync.Mutex* or *sync.RWMutex* variable. Put simply, avoid, at almost any cost, spreading mutexes across functions because that makes it really hard to see whether you are embedding or not!

The sync.RWMutex type

sync.RWMutex

The sync.RWMutex type is another kind of mutex that is based on **sync.Mutex** with the necessary additions and improvements. In reality, it is an improved version of sync.Mutex, which is defined in the **rwmutex.go** file of the **sync** directory.

Now let's talk about how sync.RWMutex improves sync.Mutex. Although only one function is allowed to perform write operations using a sync.RWMutex mutex, you can have **multiple readers** owning a sync.RWMutex mutex. However, there is one thing of which you should be aware off: until all of the readers of a sync.RWMutex mutex unlock that mutex, you cannot lock it for writing, which is the price you have to pay for allowing multiple readers.

sync.RWMutex

The functions that can help you work with a sync.RWMutex mutex are **RLock()** and **RUnlock()**, which are used for locking and unlocking the mutex, respectively, for reading purposes. The **Lock()** and **Unlock()** functions used in a **sync.Mutex** mutex should still be used when you want to lock and unlock a **sync.RWMutex** mutex for writing purposes. Thus, an **RLock()** function call that locks for reading purposes should be paired with an **RUnlock()** function call.

It should be apparent that you should not make changes to any shared variables inside the **RLock()** and **RUnlock()** blocks of code.

Things will become clearer in the Lab section that follows.

The name of the Go program that illustrates sync.RWMutex is **s4/rwMutex.go**.

QUESTIONS SO FAR?

Lab 5

Shared variables: sync.Mutex

Action!

Let me share my Terminal window and execute lab5/mutex.go.

We might try to make some changes to it so see how it behaves!

Forgetting to unlock

Unlock!

Forgetting to unlock a mutex is a bad thing - look at the Go code in the next Go slide.

forgetMutex.go

```go
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

```go
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

Shared variables: sync.RWMutex

Let me share my Terminal window and execute **lab5/rwMutex.go**.

We might try to make some changes to it so see how it behaves!

Two versions!

```
func showWithLock(c *secret) string {
	c.M.Lock()
	fmt.Println("showWithLock")
	time.Sleep(3 * time.Second)
	defer c.M.Unlock()
	return c.password
}
```

```
func show(c *secret) string {
	c.RWM.RLock()
	fmt.Print("show")
	time.Sleep(3 * time.Second)
	defer c.RWM.RUnlock()
	return c.password
}
```

Some benchmarking

Now, let me do some benchmarking, to really understand how using **sync.RWMutex** can improve the performance of a program.

We will use the time(1) utility. As I am using zsh(1) that has its own version of time(1), I will have to return to **bash(1)** before invoking the default time(1) command.

As you will see, the version that uses **sync.RWMutex** is much faster when multiple read operations take place. If you have no read operations, then both versions should take the same time – let me try that now…

Lab 6

This is going to be quick. You are going to see the Go code of the next two programs:

- A concurrent TCP server
- A version of the **find(1)** utility written in go

A concurrent TCP server

Show me the code!

```
func handleConnection(c net.Conn) {
    fmt.Printf("Serving %s\n",
c.RemoteAddr().String())
    for {
        netData, err :=
bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        result := strconv.Itoa(random()) + "\n"
        c.Write([]byte(string(result)))
    }
    c.Close()
}
```

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
    rand.Seed(time.Now().Unix())

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
    }
}
```

Using concTCP.go

Using fiboTCP.go

QUESTIONS SO FAR?

Implementing find(1)

Show me the code!

```
	walkFunction := func(path string, info os.FileInfo, err error) error {
		fileInfo, err := os.Stat(path)
		if err != nil {
			return err
		}

		if printAll {
			fmt.Println(path)
			return nil
		}

		mode := fileInfo.Mode()
		if mode.IsRegular() && *minusF {
			fmt.Println(path)
			return nil
		}

		if mode.IsDir() && *minusD {
			fmt.Println(path)
			return nil
		}

		fileInfo, _ = os.Lstat(path)
		if fileInfo.Mode()&os.ModeSymlink != 0 {
			if *minusSL {
				fmt.Println(path)
				return nil
			}
		}

		if fileInfo.Mode()&os.ModeNamedPipe != 0 {
			if *minusP {
				fmt.Println(path)
				return nil
			}
		}

		if fileInfo.Mode()&os.ModeSocket != 0 {
			if *minusS {
				fmt.Println(path)
				return nil
			}
		}

		return nil
	}
```

Using improvedFind.go

Section 6

Go Package versioning

Package Versioning

The subject of the last section of this course will be *Package Versioning*, which is a Go feature that came with Go version *1.11*.

First, let us start by presenting the Go code of the next slide, which is the initial version of a package that will be used in the last section of the Live Course.

PS. This section is more like a Lab :)

aGoPack

Live coding section!

Action!

Now, let me share my Terminal window and illustrate Package versioning.

aGoPack - final version

```
Live coding section!
```

Package versioning in action

Package Versioning

Now, let me share my Terminal window to learn more about Package Versioning in Go as this is a practical subject.

Last: How I code

About coding

There exist many approaches to coding. I will now share my screen to illustrate the way I code.

Please remember that there is no right or wrong way of coding – do what feels natural to you and generates great code!

QUESTIONS?

THANK YOU!