Effective concurrency in go

journey in the GO memory model

Who am I ?

Mohamed Badawi
@HashiCorp

@moogacs

@moogacs

https://www.linkedin.com/in/mibrahimcs

Agenda

- What, Why and How Go
- Concurrency Primitives
- Go Memory model
- Happens-Before relationship
- Dataraces
- Q/A

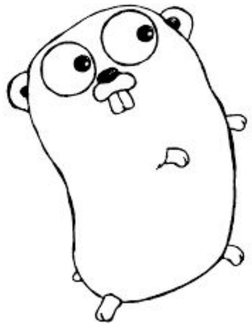
What is GO ?

- Open source (BSD) licence language supported by Google
- Easy to learn
- Built-in **concurrency** and robust std library
- Statically typed
- Tools
- Strong Documentation
- Gophers

Why GO ?

Go was created to address exactly these concurrency needs for scaled applications, microservices, and cloud development. In fact, over 75 percent of projects in the Cloud Native Computing Foundation are written in Go.

https://go.dev/solutions/cloud



How ?

Address challenges with the modern cloud, delivering standard idiomatic APIs

Go addresses many challenges developers face with the modern cloud, delivering standard idiomatic APIs, and built in concurrency to take advantage of multicore processors. Go’s low-latency and “no knob” tuning make Go a great balance between performance and productivity - granting engineering teams the power to choose and the power to move.

https://go.dev/solutions/cloud

Moore’s law

Moore's law is the observation that the number of transistors in an integrated circuit (IC) doubles about every two years.

Concurrency vs Parallelism

Concurrency is dealing multiple things at a single time while parallelism is doing multiple things at single time.

Rob Pike

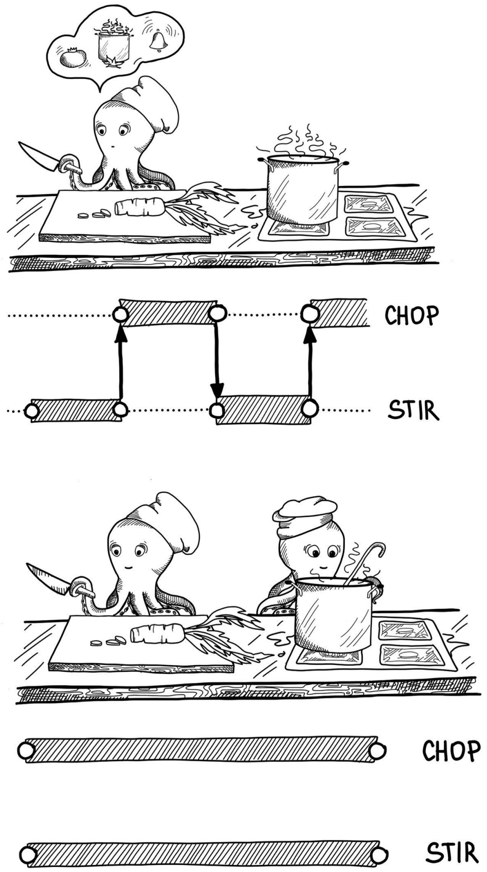
Concurrency vs Parallelism

Concurrency is about dealing with a lot of things at once.

Parallelism is about doing lots of things at once.

Not the same, but related.

One about structure, the other about execution.



Go Concurrency Primitives

- Goroutine
- Channel and the `select` statement
- Mutexes
- Wait Groups
- Condition variables

Goroutines in practice

GO runtime starts several goroutines when a program starts.

- is an execution context that is managed by the Go runtime.
- at least one for the garbage collector
- another for the main goroutine simply calls the main function and terminates the program when it returns.

```
func f (){
      fmt.Println("Hello from goroutine")
}

func main (){
      go f()
      fmt.Println("Hello from main")
      time.Sleep(100)
}
```

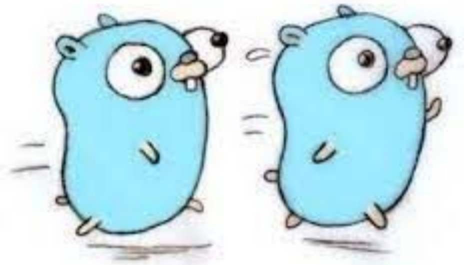
```
Hello from main
Hello from goroutine
```

```
Hello from goroutine
Hello from main
```

```
Hello from main
```

Goroutines are not threads

- They’re a bit like threads, but they’re much cheaper
- Goroutines are multiplexed onto OS threads as required.
- When a goroutine blocks, that thread blocks but other goroutines blocks.
- It’s very common for a Go application to have hundreds and even thousands of goroutines running concurrently.



What will happen here ?

```
var x int
var done bool

func setup() {
   x = 100
   done = true
}

func main() {
    go setup()
    for !done{
    }
    fmt.Println(x)
}
```

no thing

100

Let’s not do that

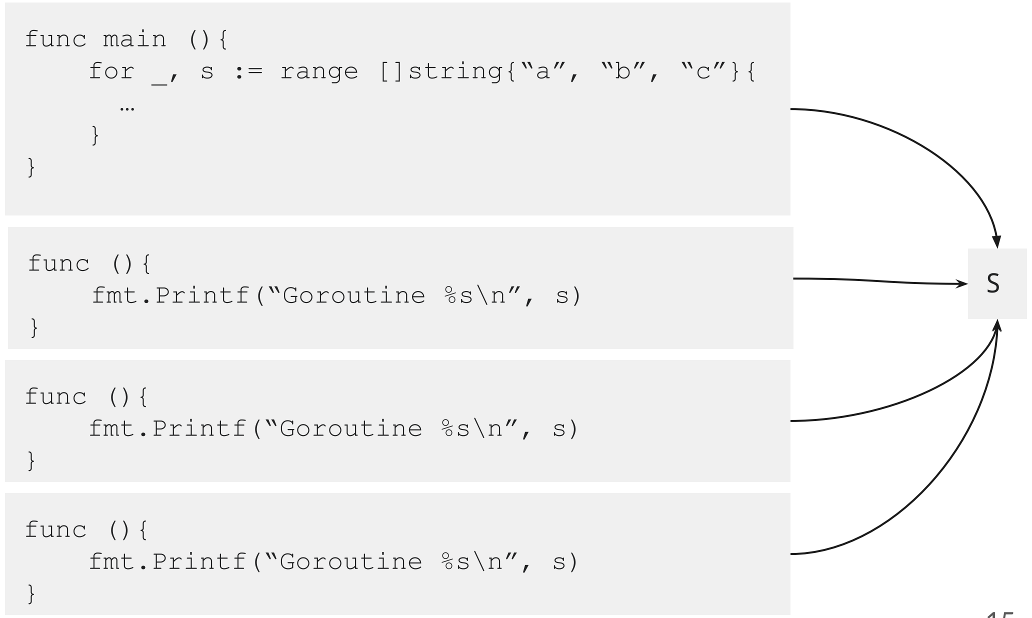
Goroutines in practice

```
func main (){
    for _, s := range []string{"a", "b", "c"} {
      go func(){
          fmt.Printf("Goroutine %s\n", s)
      }()
    }
}
```

```
Goroutine c
Goroutine c
Goroutine c
```

Goroutines in practice (closure)

```
func main (){
    for _, s := range []string{"a", "b", "c"}
{
        go func(){
            fmt.Printf("Goroutine %s\n", s)
        }()
    }
}
```



```
func main (){
    for _, s := range []string{"a", "b", "c"}{
        …
    }
}
```

```
func (){
    fmt.Printf("Goroutine %s\n", s)
}
```

```
func (){
    fmt.Printf("Goroutine %s\n", s)
}
```

```
func (){
    fmt.Printf("Goroutine %s\n", s)
}
```

Goroutines in practice

```
for _, s := range []string{“a”, “b”, “c”} {
    s := s // Redeclare s, create a copy
    // Here, the redeclared s shadows the loop  variable s
    go func() {...}
}
```

```
for _, s := range []string{“a”, “b”, “c”} {
    go func (s string) {
      fmt.Printf(”Goroutine %s\n”, s)
    }(s) // This will pass a copy of s to the function
}
```

```
Goroutine a
Goroutine b
Goroutine c
```

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate,

and allows to share memory be communicating

```
c := make(chan int)
```

```
c <- 1 // sending to channel
```

```
x = <-c // reading from channel
```

```
var c = make(chan int, 10)
var a string

func f() {
	a = "Hello, DevFestHH"
	c <- 0
}

func mai
Hello, DevFestHH

	print(a)
}
```

Select statement

The select statement lets a goroutine wait on multiple communication operations.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

```
select {
    case x := <- ch1:
    // Received x from ch1
    case y := <- ch2:
    // Received x from ch2
    default:
    // optional default, if none of the other
    operation can proceed
}
```

```
select {} // blocking indefinitely
```

```
select { // non blocking receive
    case x := <- ch:
    // Received x from ch
    default:
}
```

```
select { // non blocking send
    case ch <-x:
    // send to ch
    default:
}
```

Wait Groups

To wait for multiple goroutines to finish, we can use a wait group.

```
func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)
}

func main() {
    var wg sync.WaitGroup
    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            worker(i)
        }(i)
    }
    wg.Wait()
}
```

Remember? Do this instead

```go
var x int
var wg sync.WaitGroup

func setup() {
   x = 100
   wg.Done()
}

func main() {
    wg.Add(1)
    go setup()
    wg.Wait()

    fmt.Println(x)
}
```

```
100
```

The Go memory model

Advice

Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access.

To serialize access, protect the data with channel operations or other synchronization primitives such as those in the sync and sync/atomic packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

Don't be clever.

https://go.dev/ref/mem

Synchronization characteristics of GO concurrency

```
package B

import “fmt”

func init() {
    fmt.Println(“B initializing”)
 }
```

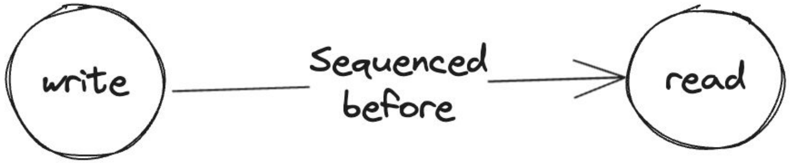
```
package A

import (
    “fmt”
    “B”
)

func init() {
    fmt.Println(“A initializing”)
 }
```

```
B initializing
A initializing
```

Sequenced before



The memory operations in each goroutine must correspond to a correct sequential execution of that goroutine, given the values read from and written to memory

That execution must be consistent with the sequenced before relation

Compiler

```
x = 1
y = 2
z = x
y++
w=y
```

Compiler Reorder

```
x = 1
z = x
y = 2
y++
w=y
```

Synchronized before

Synchronizing memory operations can be used to define the synchronized-before relationship when multiple goroutines are involved.

- if a synchronizing memory read operation of a variable observes the last synchronized write operation to that variable.
- then that synchronized write operation is synchronized before the synchronized read operation.

Synchronized before example

```
package main
import (
	"fmt"
	"sync"
)
func main() {
	var v int
	var wg sync.WaitGroup
	wg.Add(2)
	go func() {
		v = 1
		wg.Done()
	}()
	go func() {
		fmt.Println(v)
		wg.Done()
	}()
	wg.Wait()
}
```

Data race; let’s not do that

Synchronized before

```
var v int
var wg sync.WaitGroup
wg.Add(2)
ch := make(chan int)
go func() {
  v = 1
  ch <- v
  wg.Done()
}()
go func() {
  <-ch
  fmt.Println(v)
  wg.Done()
}()
wg.Wait()
```

```
1
```

happened-before relationship

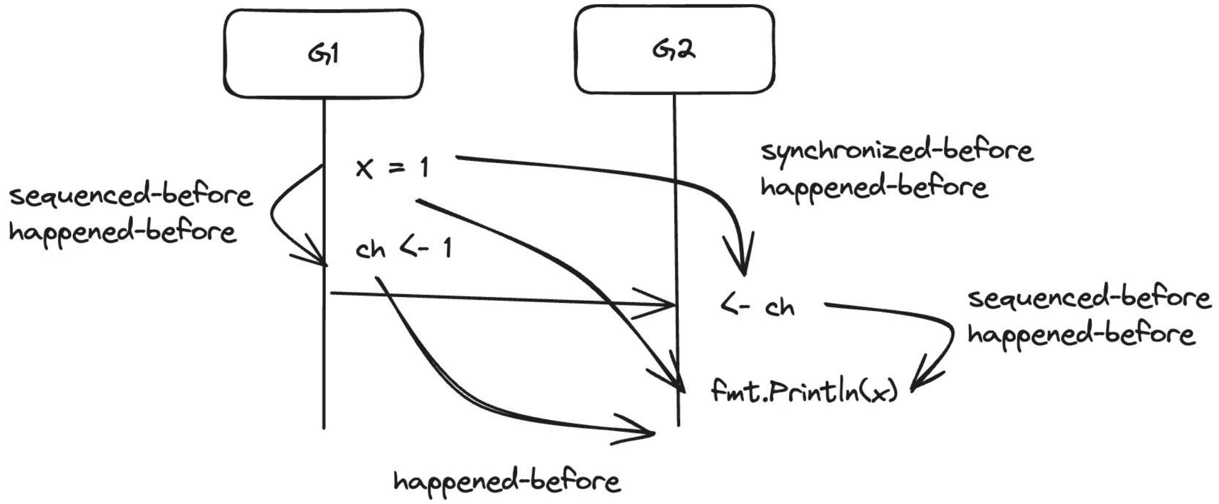
combination of a synchronized-before and sequenced-before relationships

- If memory write operation w , is synchronized before a memory read operation r, then w happened before r.
- If a memory write operation x, is sequenced before w and a memory read operation y is sequenced before r , then x happened before y,

happened-before

```
go func() {
    x = 1
    ch <- 1
}()

go func() {
    <-ch
    fmt.Println(x)
}
```



What creates happens-before relationship ?

- Goroutine creation
- Channel communication
- Locks
- sync.Once
- Import initialization
- Functions in sync package
    - Wait Groups
    - Atomics

Why data races aren’t okay

- Memory corruption
- Panics:
  - unexpected fault address 0x0, fatal error: fault
  - fatal error : concurrent map read map write
- May be appear fine now, but will cause problems at the worst possible time.

Data race detector

Go includes a built-in data race detector. To use it, add the -race flag to the go command

```
$ go test -race mypkg    // to test the package
$ go run -race mysrc.go  // to run the source file
$ go build -race mycmd   // to build the command
$ go install -race mypkg // to install the package
```

Data race detector

```go
var x int
var done bool

func setup() {
	x = 100
	done = true
}

func main() {
	go setup()
	for !done {
	}
	fmt.Println(x)
}
```

```
==================
WARNING: DATA RACE
Write at 0x0000011fbd08 by goroutine 6:
  main.setup()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:10 +0x3b

Previous read at 0x0000011fbd08 by main goroutine:
  main.main()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:15 +0x33

Goroutine 6 (running) created at:
  main.main()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:14 +0x27
==================
==================
WARNING: DATA RACE
Read at 0x0000011fbd90 by main goroutine:
  main.main()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:17 +0x4e

Previous write at 0x0000011fbd90 by goroutine 6:
  main.setup()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:9 +0x24

Goroutine 6 (running) created at:
  main.main()
      /Users/mohamedbadawi/go/src/github.com/atlas/main.go:14 +0x27
==================
100
Found 2 data race(s)
exit status 66
```

Recap

- Concurrently is not parallelism.
- Goroutines are cheaper than threads.
- Dataraces are evil.
- Go proverb Don’t communicate by sharing memory, share memory by communicating.
- Keep in mind **happens-before** relationship when impl. concurrent app.
- Channels orchestrate.

Resources

- https://go.dev/ref/mem
- https://go-proverbs.github.io [videos]
- https://go.dev/blog/pipelines
- https://freecontent.manning.com/concurrency-vs-parallelism

Questions

Thank you

@moogacs

@moogacs

https://www.linkedin.com/in/mibrahimcs