

IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean

PROGRAMARE
CONCURENȚA ÎN
GO



[The Go Programming Language](#)

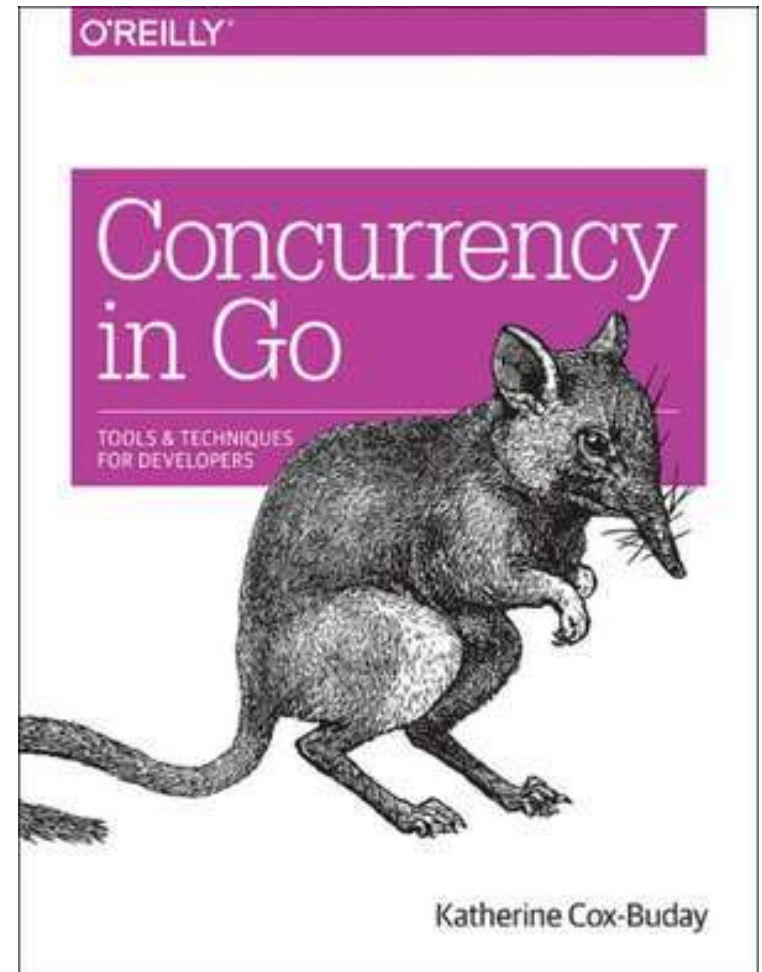
Goroutines and channels

"Before Go was first revealed to the public, this was where the chain of abstraction ended for most of the popular programming languages. If you wanted to write concurrent code, you would model your program in terms of threads and synchronize the access to the memory between them. If you had a lot of things you had to model concurrently and your machine couldn't handle that many threads, you created a *thread pool* and multiplexed your operations onto the thread pool.

Go has added another link in that chain: the *goroutine*. In addition, Go has borrowed several concepts from the work of famed computer scientist Tony Hoare, and introduced new primitives for us to use, namely *channels*.

...

Threads are still there, of course, but we find that we rarely have to think about our problem space in terms of OS threads. Instead, we model things in goroutines and channels, and occasionally shared memory."



➤ Bibliografie

[Katherine Cox-Buday, Concurrency in Go, O'Reilly, 2017](#)

[Concurrency — An Introduction to Programming in Go | Go Resources](#)

[Get Started - The Go Programming Language](#)

[Go Packages - Go Packages](#)

[The Go Programming Language Specification - The Go Programming Language](#)

[Go Wiki: Home - The Go Programming Language](#)



```
package main
```

```
import "fmt"
```

```
var x string = "hello"
```

```
func main() {
```

```
    fmt.Println(x)
```

```
}
```

Programele sunt grupate in pachete (module).

Un pachet poate continue mai multe fisiere.

Fisierele executabile trebuie sa contina functia **main**.

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\hello> go mod tidy
PS C:\Users\igleu\Documents\DIR\ICLP\GO\hello> go run hello.go
hello
```



- Sistemul tipurilor este static.
- Declaratia tipului nu e obligatorie, variabilele se pot declara cu valoare initiala si tipul e dedus:

```
var x int  
x= 32
```

```
var x int = 32
```

```
x:= 32 (in acest caz nu declaram tipul)
```

- Tipuri:
simple: **int, int16, int32, float32,**
float64, string (immutable), **bool**
compuse: **map, array, struct, slice**

- Pointeri:
x:=1
p:=&x
***p:=2**

- Tipuri definite de utilizator
- Metode asociate tipurilor
- Interfete

```
type Point struct{ X, Y float64 }
```

```
func (p Point) Distance(q Point) float64 {  
    return math.Hypot(q.X-p.X, q.Y-p.Y)}
```

Distance este o metoda asociata Point



```
for i <= 3 {  
    fmt.Println(i)  
    i = i + 1  
}  
  
for j := 0; j < 3; j++  
{  
    fmt.Println(j)  
}  
  
for {  
  
    fmt.Println("loop")  
    break  
}
```

```
if 7%2 == 0 {  
    fmt.Println("7 is even")  
} else {  
    fmt.Println("7 is odd")  
}  
  
if 8%4 == 0 {  
    fmt.Println("8 is divisible by 4")  
}
```



Funcțiile sunt valori

```
import (  
    "fmt")  
  
var salutation string  
var nume string = "Ioana"  
  
func g() func(s string) {  
    var salutation string  
    salutation = "welcome"  
    nume = "Ana"  
    return func(s string) { fmt.Println(salutation, s, nume) }  
}
```

```
func main() {  
    salutation = "hello"  
    fmt.Println(salutation, nume)  
    g()(" you")  
    fmt.Println(salutation, nume)  
}
```

```
> go run fctex.go  
hello Ioana  
welcome you Ana  
hello Ana
```



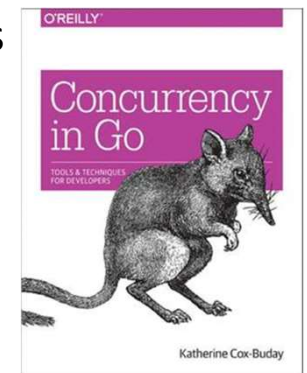
Concurenta in Go

Gorutine si canale



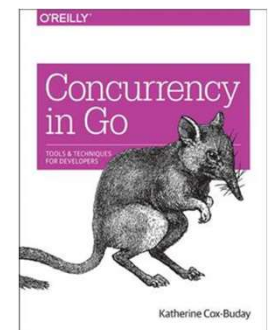
Goroutines are unique to Go (though some other languages have a concurrency primitive that is similar). They're not OS threads, and they're not exactly green threads—threads that are managed by a language's runtime—they're a higher level of abstraction known as *coroutines*. Coroutines are simply concurrent subroutines (functions, closures, or methods in Go) that are *nonpreemptive*—that is, they cannot be interrupted. Instead, coroutines have multiple points throughout which allow for suspension or reentry.

What makes goroutines unique to Go are their deep integration with Go's runtime. Goroutines don't define their own suspension or reentry points; Go's runtime observes the runtime behavior of goroutines and automatically suspends them when they block and then resumes them when they become unblocked. In a way this makes them preemptable, but only at points where the goroutine has become blocked. It is an elegant partnership between the runtime and a goroutine's logic. Thus, goroutines can be considered a special class of coroutine.



Go's mechanism for hosting goroutines is an implementation of what's called an *M:N scheduler*, which means it maps M green threads to N OS threads. Goroutines are then scheduled onto the green threads. When we have more goroutines than green threads available, the scheduler handles the distribution of the goroutines across the available threads and ensures that when these goroutines become blocked, other goroutines can be run.

Go follows a model of concurrency called the *fork-join* model.¹ The word *fork* refers to the fact that at any point in the program, it can split off a *child* branch of execution to be run concurrently with its *parent*. The word *join* refers to the fact that at some point in the future, these concurrent branches of execution will join back together. Where the child rejoins the parent is called a *join point*.



```
package main
```

```
import (  
    "fmt"  
)
```

```
func f(s string) {  
    for i := 0; i < 30; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {
```

```
    go f("A")  
    go f("B")
```

```
    fmt.Println("gata")
```

```
}
```

main este o gorutina

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go mod tidy  
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go  
gata
```



```
import (
    "fmt")
```

```
func main() {
    go f("A")
    go f("B")
}
```

```
var input string
fmt.Scanln(&input)
fmt.Println("gata")
```

[illegible]

```
package main
```

```
import (  
    "fmt")
```

```
func f(s string) {  
    for i := 0; i < 30; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {  
    go f("A")  
    go func(s string) {  
        for i := 0; i < 30; i++ {  
            fmt.Print(s)  
        }  
    }("B")
```

apelul unei functii anonime

```
var input string  
fmt.Scanln(&input)  
fmt.Println("gata") }
```

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go  
AAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAe  
gata
```



```
package main
```

```
import (
```

```
    "fmt"
```

```
    "math/rand"
```

```
    "time")
```

```
func f(s string) {
```

```
    for i := 0; i < 30; i++ {
```

```
        fmt.Print(s)
```

```
        amt := time.Duration(rand.Intn(250))
```

```
        time.Sleep(time.Millisecond * amt)
```

```
    }}
```

Sleep opreste executia corutinei

[time package - time - Go Packages](#)

```
func main() {
```

```
    go f("A")
```

```
    go f("B")
```

```
    var input string
```

```
    fmt.Scanln(&input)
```

```
    fmt.Println("gata")
```

```
}
```

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
BABBBAAAABBABAABBABBBAAABABAABBABABBBABAABAABAABAABBBABABBe
gata
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
ABBAAABABAABAAABABABBABBBBAABBAAABBABBAABABABBBABBBABABe
gata
```



```
package main
```

sincronizare folosind **sync.WaitGroup**

```
import (  
    "fmt"  
    "sync"  
)
```

```
var wg sync.WaitGroup
```

```
func f(s string) {  
    defer wg.Done()  
    for i := 0; i < 5; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {  
    wg.Add(1)  
    go f("A")
```

```
    wg.Wait()  
    fmt.Println("gata")  
}
```

```
> go run concABW.go  
AAAAAgata
```

defer amana executia unei functii pana cand functia parinte isi termina executia



```
package main
```

```
import (  
    "fmt"  
    "sync")
```

```
var wg sync.WaitGroup
```

```
func f(s string) {  
    defer wg.Done()  
    for i := 0; i < 5; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {  
    wg.Add(1)  
    go f("A")  
  
    wg.Wait()  
    fmt.Println("gata")  
}
```

sincronizare folosind **sync.WaitGroup**

A WaitGroup waits for a collection of goroutines to finish.

The main goroutine calls [WaitGroup.Add] to set the number of goroutines to wait for. Typically the calls to Add should execute before the statement creating the goroutine or other event to be waited for.

Then each of the goroutines runs and calls [WaitGroup.Done] when finished.

At the same time, [WaitGroup.Wait] can be used to block until all goroutines have finished.

[- The Go Programming Language](#)




```
package main
```

sincronizare folosind **sync.WaitGroup**

```
import (  
    "fmt"  
    "sync"  
)
```

```
var wg sync.WaitGroup
```

```
func f(s string) {  
    defer wg.Done()  
    for i := 0; i < 5; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {  
    wg.Add(1)  
    go f("A")  
    wg.Add(1)  
    go f("B")  
  
    wg.Wait()  
    fmt.Println("gata")  
}
```

```
> go run concABW.go  
BBBBBAAAAAgata
```



```
package main
```

Transmiterea ca parametru a unui **WaitGroup**
se face prin referinta

```
import (  
    "fmt"  
    "sync"  
)
```

```
func f(s string, wg *sync.WaitGroup) {  
    defer wg.Done()  
    for i := 0; i < 5; i++ {  
        fmt.Print(s)  
    }  
}
```

```
func main() {  
    var wg sync.WaitGroup  
    wg.Add(1)  
    go f("A", &wg)  
    wg.Add(1)  
    go f("B", &wg)  
  
    wg.Wait()  
    fmt.Println("gata")  
}
```

```
> go run concABW.go  
BBBBBAAAAAgata
```



```
package main
```

```
import (  
    "fmt"  
    "sync"  
)
```

```
func f(s string, wg *sync.WaitGroup) {  
    defer wg.Done()  
    for i := 0; i < 5; i++ {  
        fmt.Print(s)  
    }  
}
```

wg actioneaza ca un contor care este incrementat cu valoarea transmisa prin **Add**.
Wait blocheaza executia pana cand contorul are valoarea 0.

```
func main() {  
    var wg sync.WaitGroup  
    wg.Add(2)  
    go f("A", &wg)  
    go f("B", &wg)  
  
    wg.Wait()  
    fmt.Println("gata")  
}
```

```
> go run concABW.go  
BBBBBAAAAAgata
```



Closures

```
func main() {  
    var wg sync.WaitGroup  
    for _, salutation := range []string{"hello",  
                                         "greetings", "good day"} {  
        wg.Add(1)  
        go func() {  
            defer wg.Done()  
            fmt.Println(salutation)  
        }()  
    }  
    wg.Wait()  
}
```

```
> go run gocl.go  
greetings  
hello  
good day  
  
> go run gocl.go  
good day  
hello  
greetings  
  
> go run gocl.go  
greetings  
good day  
hello
```

<https://go.dev/wiki/LoopvarExperiment>



Observatie: Closures si Wait

```
import (  
    "fmt"  
    "sync")  
  
var salutation string  
var nume string = "loana"  
var wg sync.WaitGroup  
  
func g() func(s string) {  
    var salutation string  
    defer wg.Done()  
    salutation = "welcome"  
    return func(s string) {  
        fmt.Println(salutation, s, nume)}}}
```

```
func main() {  
    salutation = "hello"  
    wg.Add(1)  
    go g()(" you")  
    wg.Wait()  
    fmt.Println(salutation, nume)}
```

```
> go run closure1.go  
welcome you loana  
hello loana
```

```
> go run closure1.go  
hello loana
```

nu s-a executat functia closure



Observatie: Closures si Wait

```
import (  
    "fmt"  
    "sync")  
  
var salutation string  
var nume string = "loana"  
var wg sync.WaitGroup  
  
func g() func(s string) {  
    var salutation string  
    salutation = "welcome"  
    return func(s string) {  
        defer wg.Done()  
        fmt.Println(salutation, s, nume)}}}
```

```
func main() {  
    salutation = "hello"  
    wg.Add(1)  
    go g()(" you")  
    wg.Wait()  
    fmt.Println(salutation, nume)}
```

```
> go run closure1.go  
welcome you loana  
hello loana
```

```
> go run closure1.go  
hello loana
```



- Gorutinele comunica intre ele folosind **canale**
- Canalele pot fi cu capacitate (buffered) sau fara capacitate (unbuffered, cu capacitate 0)
 - `c := make(chan int)` (este echivalent cu `make(chan int,0)`)
 - `c := make(chan int, 3)`
- Operatiile cu canale:
 - `c <- x` // valoarea x este trimisa pe canal (scriere)
 - `x = <-c` // x primeste o valoare de pe canal (citire)
 - `close(c)` // canalul este inchis; se pot face citiri, dar valoarea citita va fi 0 (valoarea nula a tipului respectiv; nu se pot face scrieri)
- Comunicarea pe canalele fara capacitate este **sincrona**: un mesaj este transmis (scris) numai daca exista si cineva care va primi (citi) acel mesaj; in caz contrar, gorutina care trimite mesajul este blocata pana cand cea care primeste este disponibila.
- Comunicarea pe canalele cu capacitate este **asincrona**. Accesul la canale este blocant: o gorutina care incearca sa citeasca dintr-un canal gol va astepta pana cand canalul continue o valoare si orice gorutina care vrea sa scrie pe un canal va astepta pana cand exista o locatie libera.



Sincronizare folosind canale in loc de sync.WaitGroup

```
func f(s string, c chan string) {  
    for i := 0; i < 30; i++ {  
        fmt.Print(s)  
    }  
    c <- "gata"  
}
```

```
func main() {  
    done := make(chan string)  
    go f("A", done)  
    s := <-done  
    fmt.Println(s)  
}
```



Comunicare folosind canale

```
func f(s string, c chan string) {  
    defer close(c)  
    for i := 0; i < 30; i++ {  
        c <- s  
    }  
}
```

```
func main() {  
    buf := make(chan string)  
    go f("A", buf)  
    s, ok := <-buf  
    for ok {  
        fmt.Println(s)  
        s, ok = <-buf  
    }  
}
```

ok va fi **false** cand canalul este inchis



Comunicare folosind canale

```
func f(s string, c chan string) {  
    defer close(c)  
    for i := 0; i < 30; i++ {  
        c <- s  
    }  
}
```

```
func main() {  
    buf := make(chan string)  
    go f("A", buf)  
  
    for s := range buf {  
        fmt.Println(s)  
    }  
}
```

se poate folosi **range** pentru a itera prin valorile unui canal



Comunicare folosind canale unidirectionale: `chan<-` si `<-chan`

```
func f(s string, c chan<- string) {  
    defer close(c)  
    for i := 0; i < 3; i++ {  
        c <- s  
    }  
}
```

```
func main() {  
    buf := make(chan string)  
    go f("A", buf)  
  
    for s := range buf {  
        fmt.Println(s)  
    }  
}
```

canalele pot fi unidirectionale:
parametrul functiei f poate fi folosit numai pentru trimiterea mesajelor,
iar incercare de a-l folosi pentru citire va da eroare



Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {  
    defer close(c)  
    for i := 0; i < 3; i++ {  
        c <- s  
    }  
}
```

apelul unei functii anonime

```
func main() {  
    bufA := make(chan string)  
    bufB := make(chan string)  
    go f("A", bufA)  
    go f("B", bufB)  
    go func() {  
        for a := range bufA {  
            fmt.Println(a)}}()  
    go func() {  
        for b := range bufB {  
            fmt.Println(b)}}()  
    var input string  
    fmt.Scanln(&input)}
```

Este necesar pentru a ne asigura ca sunt executate gorutinele.
Se poate folosi un alt canal sau Wait



Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {  
    defer close(c)  
    for i := 0; i < 5; i++ {  
        c <- s  
    }  
}
```

Toate conditiile de pe ramurile instructiunii select sunt analizate in paralel. Daca niciuna nu poate fi executata se executa **default**, iar daca default lipseste gorutina este blocata pana cand o conditie este indeplinita.

```
func main() {  
    bufA := make(chan string)  
    bufB := make(chan string)  
    go f("A", bufA)  
    go f("B", bufB)  
    for j := 0; j < 15; j++ {  
        select {  
            case a := <-bufA:  
                fmt.Println(a)  
            case b := <-bufB:  
                fmt.Println(b)  
            default:  
                fmt.Println("  .")  
        }  
    }  
}
```



Comunicare folosind canale: gorutine pentru citire si scriere

```
> go run chan5.go
.BAAAABABBB . . . .

> go run chan5.go
.AAAAABBBBB . . . .

> go run chan5.go
.BABBBBAA .AA . . .
```

```
func main() {
    bufA := make(chan string)
    bufB := make(chan string)
    go f("A", bufA)
    go f("B", bufB)
    for j := 0; j < 15; j++ {
        select {
            case a := <-bufA:
                fmt.Print(a)
            case b := <-bufB:
                fmt.Print(b)
            default:
                fmt.Print(" .")
        }
    }
}
```



Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {  
    defer close(c)  
    for i := 0; i < 5; i++ {  
        c <- s  
    }  
}
```

Functia **time.After (t)** intoarce un canal care va trimite timpul curent dupa perioada t.

```
func main() {  
    bufA := make(chan string)  
    bufB := make(chan string)  
    go f("A", bufA)  
    go f("B", bufB)  
    for j := 0; j < 15; j++ {  
        ok := true  
        for ok {  
            select {  
            case a := <-bufA: fmt.Println(a)  
            case b := <-bufB: fmt.Println(b)  
            case <-time.After(5 * time.Second):  
                ok = false  
            }  
        }  
    }  
}
```

