Goroutines and channels - simple, but powerful tools in Go arsenal

15 March 2016

About me

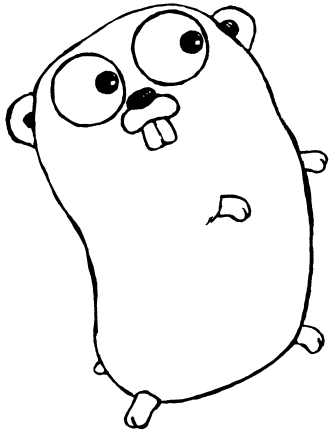
Krzysztof Kwapisiewicz, Software Developer at CodiLime

krzysztof.kwapisiewicz@codilime.com
@kwapik

github.com/kwapik/goroutines-and-channels-talk

References

- "Go Concurrency Patterns" (https://www.youtube.com/watch?v=f6kdp27TYZs) by Rob Pike
- Effective Go (https://golang.org/doc/effective_go.html#concurrency)
- 1 year of relationship with Gopher



Goroutines

Independently executing function, launched by a Go statement.

Has own call stack which grows and shrinks as required.

Very cheap. Do not be afraid to run 5 digit number of them.

Multiplexed onto multiple OS thread.

Life without goroutines

```
func main() {
    cheer("Yay! I'm a function!")
}

func cheer(msg string) {
    for i := 0; i < 5; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Goroutines - simple_example

```
func main() {
    go cheer("Yay! I'm a goroutine!")
}

func cheer(msg string) {
    for i := 0; i < 5; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Goroutines - hotfix/simple_example

```
func main() {
    go cheer("Yay! I'm a goroutine!")
    time.Sleep(time.Second)
}

func cheer(msg string) {
    for i := 0; i < 5; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Goroutines - simple_example TODO

Waiting is not very reliable way to ensure goroutine finished all important stuff.

We want to exchange information.

Channels

Provide connection between goroutines so they can communicate.

```
// Declaring and initializing
c := make(chan int)

// Sending on a channel
c <- 42

// Receiving from a channel
answer := <-c
```

```
c := make(chan int, 2)

c <- 13
c <- 37

first := <-c
second := <-c
```

Synchronization

```
func main() {
    c := make(chan int)

    go func() {
        cheer("Yay! I'm a goroutine!")
        c <- 1
    }()

    <-c
    fmt.Println("Bye!")
}
```

Run

Multiple channels

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() { cheerChannel(c1, "Heeeey!") }()
    go func() { cheerChannel(c2, "Hello!") }()

    for {
        var msg string
        msg = <-c1
        fmt.Println(msg)
        msg = <-c2
        fmt.Println(msg)
    }
}
```

Run

Multiple channels with select

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() { cheerChannel(c1, "Heeeey!") }()
    go func() { cheerChannel(c2, "Hello!") }()

    for {
        select {
        case msg := <-c1:
            fmt.Println(msg)
        case msg := <-c2:
            fmt.Println(msg)
        }
    }
}
```

Run

Timeout single operation

```
func main() {
    c := make(chan string)

    go func() { cheerChannel(c, "Heeeey!") }()

    for {
        select {
        case msg := <-c:
            fmt.Println(msg)
        case <-time.After(time.Second):
            fmt.Println("Timeout!")
            return
        }
    }
}
```

Run

Timeout all operations

```
func main() {
    c := make(chan string)

    go func() { cheerChannel(c, "Heeeey!") }()

    timeout := time.After(5 * time.Second)
    for {
        select {
        case msg := <-c:
            fmt.Println(msg)
        case <-timeout:
            fmt.Println("Timeout!")
            return
        }
    }
}
```

Run

Unlimited workers

```
type Task struct {
    message string
    delay   time.Duration
}

func main() {
    tasks := make(chan Task, 1000)
    go taskProducer(tasks, 1000)

    for {
        select {
        case task := <-tasks:
            go func() {
                time.Sleep(task.delay)
                fmt.Println("Processed task:", task.message)
            }()
        default:
            fmt.Println("No tasks for now")
            time.Sleep(time.Second)
        }
    }
}
```

Run

Limited workers

```
poolSize := 5
pool := make(chan bool, poolSize)

for {
    select {
    case task := <-tasks:
        pool <- true
        go func() {
            defer func() { <-pool }()
            time.Sleep(task.delay)
            fmt.Println("Processed task:", task.message)
        }()
    default:
        fmt.Println("No tasks for now")
        time.Sleep(time.Second)
    }
}
```

Run

Results

```
    responseChan := make(chan Response, 1000)

    go func() {
        for {
            fmt.Println((<-responseChan).result)
        }
    }()

    for {
        select {
        case task := <-tasks:
            pool <- true
            go func() {
                defer func() { <-pool }()
                time.Sleep(task.delay)
                responseChan <- Response{result: task.message}
            }()
        default:
            fmt.Println("No tasks for now")
            time.Sleep(time.Second)
        }
    }
```

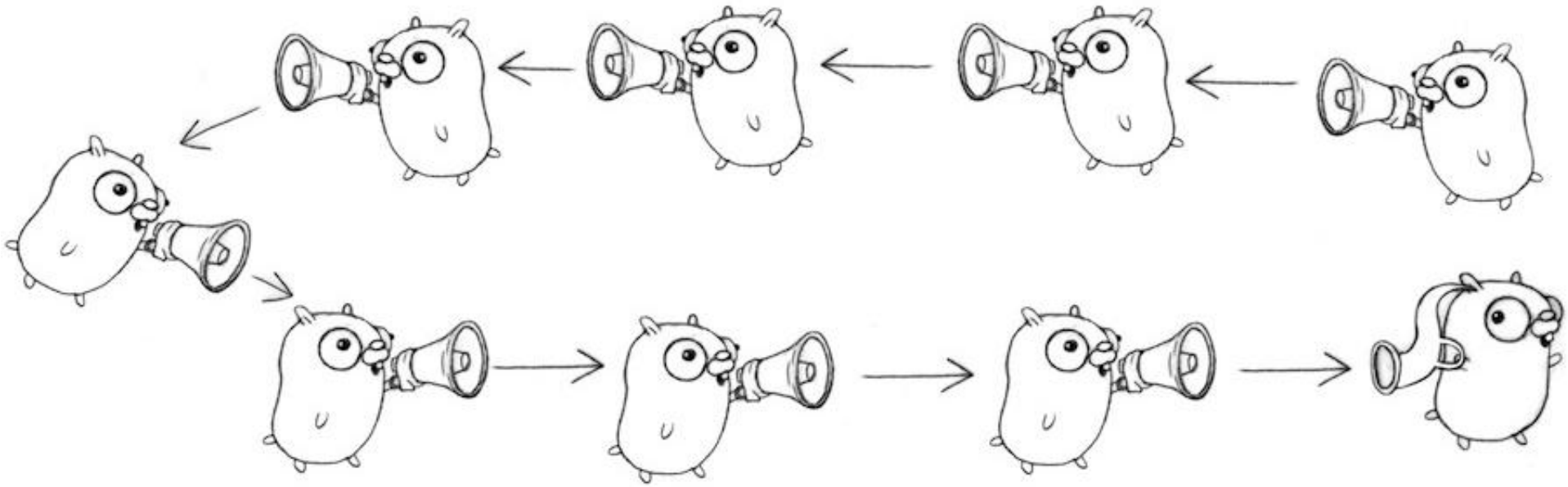
Run

Killing result reader

```
    stopChan := make(chan bool)
    time.AfterFunc(3*time.Second, func() { stopChan <- true })
    for {
        select {
        case task := <-tasks:
            pool <- true
            go func() {
                defer func() { <-pool }()
                time.Sleep(task.delay)
                responseChan <- Response{result: task.message}
            }()
        case <-stopChan:
            return
        }
    }
```

Run

How fast it can go?



How fast it can go? - code

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 100000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Run

Thank you