Go Language

September 2016

Giacomo Tartari
PhD student, University of Tromsø

Go

Why?

Rob Pike's take (one of the Go instigator) (http://talks.golang.org/2012/splash.article)

Briefly

Languages used at Google at the time (mostly Java, C++, and Python) were not satisfactory

- slow builds
- uncontrolled dependencies
- each programmer using a different subset of the language
- poor program understanding (code hard to read, poorly documented, and so on)
- duplication of effort
- difficulty of writing automatic tools
- cross-language builds

Engineering not Research

C-like syntax, simple and fast to learn

Compiled and type safe (static and strong typed)

Concurrent (CSP-like)

Garbage collected

Composition and not inheritance

Batteries included, rich std lib

Hello World

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Run

Simple and fast to learn

The code does what it says

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func (v1 Vertex) Add(v2 Vertex) Vertex {
    return Vertex{
        X: v1.X + v2.X,
        Y: v1.Y + v2.Y,
    }
}

func main() {
    p := Vertex{1, 2}
    q := Vertex{X: 4}

    fmt.Println(p.Add(q))
    fmt.Printf("%T\n", q)
}
```

Run

Visibility

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println(math.pi)
}
```

Run

Composition not inheritance

```
type Person struct {
    Name string
    Age  int
}

type User struct {
    P  Person
    Id int
}

func main() {
    u := User{}
    u.P.Name = "Adam"
    u.P.Age = 42
    u.Id = 1
    fmt.Printf("%+v\n", u)
}
```

Run

Embedding ~ inheritance

```
type Person struct {
    Name string
    Age  int
}

type User struct {
    Person
    Id     int
}

func main() {
    u := User{}
    u.Name = "Adam" // u.Person.Name = "Adam"
    u.Age = 42      //u.Person.Age = 42
    u.Id = 1
    fmt.Println(u)
}
```

Run

Embedding ~ inheritance

```
type Person struct {
    Name string
}

func (p Person) Greet() string {
    return "Hello " + p.Name
}

type User struct {
    Person
    Id int
}

func main() {
    u := User{}
    u.Name = "Adam"
    u.Id = 1
    fmt.Println(u.Greet())
}
```

Run

Encapsulation

```
type Person struct {
    Name string
}

func (p Person) Greet() string {
    return "Hello " + p.Name
}

type User struct {
    Person
    Id int
}

func (u User) Greet() string {
    return fmt.Sprintf("%s, Id: %d\n", u.Person.Greet(), u.Id)
}
func main() {
    u := User{}
    u.Name = "Adam"
    u.Id = 1
    fmt.Println(u.Greet())
}
```

Run

Interfaces

Just behavior

An interface type is defined as a set of method signatures

A value of interface type can hold any value that implements those methods

Interfaces are implemented implicitly

Go tour (https://tour.golang.org/methods/9)

Implicit implementation

```
type Greeter interface {
    Greet() string
}

func (p Person) Greet() string {
    return "Hello " + p.Name
}

func (u User) Greet() string {
    return fmt.Sprintf("%s, Id: %d\n", u.Person.Greet(), u.Id)
}
func main() {
    var g Greeter
    u := User{}
    u.Name = "Cody Coder"
    u.Id = 1
    g = u
    fmt.Println(g.Greet())

    p := Person{Name: "Tony Tester"}
    g = p
    fmt.Println(g.Greet())
}
```

Run

Caveat emptor

```
package main

import "fmt"

func main() {
    var i interface{}
    x := 127
    i = x
    fmt.Println("i:", i)
    v := struct{ π, e float32 }{3.14159, 2.71828}
    i = v
    fmt.Println("i:", i)
}
```

Run

Type assertions

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string)
    fmt.Println(s)

    s, ok := i.(string)
    fmt.Println(s, ok)

    f, ok := i.(float64)
    fmt.Println(f, ok)

    f = i.(float64) // panic
    fmt.Println(f)
}
```

Run

See Go tour

Assignability and type conversions

Read the docs

Effective Go (https://golang.org/doc/effective_go.html)

Language Specs (https://golang.org/ref/spec)

Or just try it out with 10 line of code

```
package main

import "fmt"


func main() {

    // do stuff

    fmt.Println(resutlt)
}
```

Portability

Portable

```
$ go tool dist list
```

android/386
android/amd64
android/arm
android/arm64
darwin/386
darwin/amd64
darwin/arm
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
...

Very portable

...
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips64
linux/mips64le
linux/ppc64
linux/ppc64le
linux/s390x
nacl/386
nacl/amd64p32
nacl/arm
...

Can't be too portable!

...
netbsd/386
netbsd/amd64
netbsd/arm
openbsd/386
openbsd/amd64
openbsd/arm
plan9/386
plan9/amd64
plan9/arm
solaris/amd64
windows/386
windows/amd64

Cross compilation

```
$export GOOS=linux
$export GOARCH=arm

$go build
```

Done!

Easy deployment

Very easy deployment!

```
scp gobin remote@host:/path
```

Done!

Some conditions may apply

Some functionalities (net package) require to call C code

But it is possible to avoid that

```
$export CGO=0
```

Concurrent

Concurrency vs Parallelism

Rob Pike talk about the difference (http://concur.rspace.googlecode.com/hg/talk/concur.html#title-slide)

- Concurrency != parallelism
- Concurrency is the composition of independently executing processes
- Concurrency enables parallelism
- Concurrency is about structure
- Parallelism is about execution
- A concurrent program can be executed correctly on one CPU

Concurrency at language level

- *go* statement allows us to run functions independently in different **goroutines**
- Goroutines live in the same address space
- Think of them as a very lightweight threads

Hello Goroutines

```
package main

import (
    "fmt"
    "time"
)

func main() {
    say("world")
    say("hello")
}

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Printf("%s\n", s)
    }
}
```

Run

Hello Channels

```
package main

import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

Run

Goroutine

Workers with random amount of work

```
func main() {
    t0 := time.Now()
    for i := 0; i < 5; i++ {
        do(rand.Intn(1000), i)
    }
    fmt.Printf("total time %v\n", time.Now().Sub(t0))
}

func do(work, id int) {
    t0 := time.Now()
    time.Sleep(time.Duration(work) * time.Millisecond)
    fmt.Printf("done %d [%v]\n", id, time.Now().Sub(t0))
}
```

Run

Goroutine & channels

```
func main() {
    t0 := time.Now()
    done := make(chan string)
    for i := 0; i < 5; i++ {
        go do(rand.Intn(1000), i, done)
    }
    for i := 0; i < 5; i++ {
        fmt.Println(<-done)
    }
    fmt.Printf("total time %v\n", time.Now().Sub(t0))

}

func do(work, id int, ch chan string) {
    t0 := time.Now()
    time.Sleep(time.Duration(work) * time.Millisecond)
    ch <- fmt.Sprintf("done %d [%v]", id, time.Now().Sub(t0))
}
```

Run

Avoid channels in signatures

```
func main() {
    seed := time.Now().UnixNano()
    rand.Seed(seed)
    t0 := time.Now()
    done := make(chan string)
    for i := 0; i < 5; i++ {
        go func(wid int) {
            res := do(rand.Intn(1000), wid)
            done <- res
        }(i)
    }
    for i := 0; i < 5; i++ {
        fmt.Println(<-done)
    }
    fmt.Printf("total time %v\n", time.Now().Sub(t0))

}

func do(work, id int) string {
    t0 := time.Now()
    time.Sleep(time.Duration(work) * time.Millisecond)
    return fmt.Sprintf("done %d [%v]", id, time.Now().Sub(t0))
}
```

Run

Communicate with more channels

```
func main() {
    timeout := time.After(3 * time.Second)
    done := make(chan string)
    t0 := time.Now()
    for i := 0; i < 5; i++ {
        go func(wid int) {
            res := do(rand.Intn(5000), wid)
            done <- res
        }(i)
    }
    for i := 0; i < 5; i++ {
        select {
        case res := <-done:
            fmt.Println(res)
        case <-timeout:
            fmt.Printf("timeout, workers done: %d\n", i)
            return
        }
    }
    fmt.Printf("total time %v\n", time.Now().Sub(t0))
}
```

Run

Workers pool

The worker, of which we'll run several concurrent instances

These workers will receive work on the jobs channel and send the corresponding results on results

We'll sleep a second per job to simulate an expensive task

```
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
}
```

Workers pool

In order to use our pool of workers we need to send them work and collect their results

We make 2 channels for this

```
    jobs := make(chan int, 100)
    results := make(chan int, 100)
```

This starts up 3 workers, initially blocked because there are no jobs yet

```
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }
```

Workers pool

Here we send 9 jobs and then close that channel to indicate that's all the work we have terminated

```
    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)
```

Finally we collect all the results of the work

```
    for a := 1; a <= 9; a++ {
        <-results
    }
```

Workers pool

```
func worker(id int, jobs <-chan int, results chan<- int) {
    for j := range jobs {
        fmt.Println("worker", id, "processing job", j)
        time.Sleep(time.Second)
        results <- j * 2
    }
}

func main() {
    jobs := make(chan int, 100)
    results := make(chan int, 100)
    for w := 1; w <= 3; w++ {
        go worker(w, jobs, results)
    }
    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)
    for a := 1; a <= 9; a++ {
        <-results
    }
}
```

Run

Go concurrency patterns

concurrency patterns (https://talks.golang.org/2012/concurrency.slide#1)

pipelines and cancellation (https://blog.golang.org/pipelines)

context (https://blog.golang.org/context)

advanced concurrency patterns (https://blog.golang.org/advanced-go-concurrency-patterns)

Batteries included

Standard library

packages (https://golang.org/pkg/)

Hello WWW

File server

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Simple static webserver:
    log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))
}
```

net/http package is production ready

Third party

GoDoc (https://godoc.org/)

Standard tools

- Go tool
- godoc
- golang.org/x/tools

Tests and benchmark

golang.org/pkg/testing/ (http://golang.org/pkg/testing/)

Put your tests/benchmark in a file ending in `_test.go`

```
import testing

func TestXxx(t *testing.T){
    ...
}

func BenchmarkXxx(b *testing.B){
    ...
}
```

```
$cd $GOPATH/src/mypackage
$go test
$go test -bench=.
```

golang.org/cmd/go/#Description_of_testing_flags (http://golang.org/cmd/go/#Description_of_testing_flags)

Perf tools

Debugging performance issues in Go (https://software.intel.com/en-us/blogs/2014/05/10/debugging-performance-issues-in-go-programs)

Profile (http://blog.golang.org/profiling-go-programs)

Trace (https://golang.org/cmd/trace/)

Race detector (https://golang.org/doc/articles/race_detector.html)

Fuzzer (https://github.com/dvyukov/go-fuzz)

Go is boring

asymptotically approaching boring (https://www.youtube.com/watch?v=4Dr8FXs9aJM)

Boring is Beautifu (https://www.youtube.com/watch?v=6l62RYyeLp8)

Good foundations are boring

github.com most stars (https://github.com/search?o=desc&q=language%3Ago&s=stars&type=Repositories)

The fun is upstairs

Go for Data Science (https://www.youtube.com/watch?v=D5tDubyXLrQ&list=PL2ntRZ1ySWBdliXelGAItjzTMxy2WQh0P&index=6)

Pachyderm (http://www.pachyderm.io/)

GoUsers (https://github.com/golang/go/wiki/GoUsers)

Thank you

Giacomo Tartari
PhD student, University of Tromsø
giacomo.tartari@uit.no (mailto:giacomo.tartari@uit.no)