Go

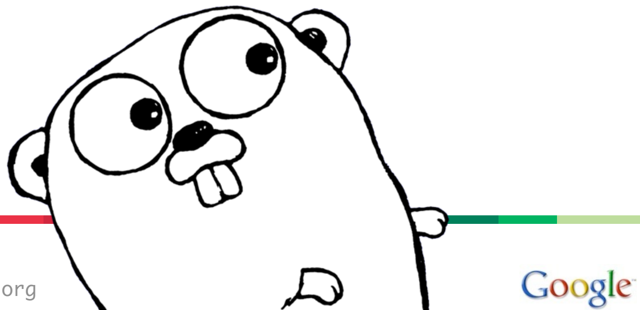
Sébastien Binet

LAL/IN2P3

2013-12-18

go

http://golang.org



Why Go ?

- Statically typed languages are efficient, but typically bureaucratic and overly complex.
- Dynamic languages can be easy to use, but are error-prone, inefficient, and break down at scale.
- Concurrent programming is hard (threads, locks, headache.)

- *“Speed, reliability, or simplicity: pick two.”* (sometimes just one)
- Can’t we do better ?

What is Go ?

- Go is a modern, general purpose language

- Compiles to native machine code (32/64-bit x86, ARM)
- Statically typed
- Lightweight syntax
- Simple type system
- Garbage collected
- Concurrency primitives

Tenets of Go's design

- Simplicity
  - Each language feature should be easy to understand
- Orthogonality
  - Go's features should interact in predictable and consistent ways.
- Readability
  - What is written on the page should be comprehensible with little context.

“Consensus drove the design. Nothing went into the language until [Ken Thompson, Robert Griesemer and myself] all agreed that it was right. Some features didn’t get resolved until after a year or more of discussion.“

Rob Pike

Hello, World

```go
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```

Hello, World - 2.0

Serving “Hello, world” at http://localhost:8080/world

```go
package main

import (
  "fmt"
  "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, "+r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

A simple type system

`Go` is statically typed, but type inference saves repetition.

- Java:
  - `Integer i = new Integer(1);`
- `C/C++`:
  - `int i = 1;`

```go
i := 1                         // type int
pi := 3.142                    // type float64
greeting := "Hello, you"       // type string

// type func(int, int) int
mul := func(x, y int) int { return x * y }
```

Types and methods

You can define methods on any type:

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

func (p Point) Abs() float64 {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

p := Point{4, 3} // type Point
x := p.Abs()     // == 5.0
```

http://play

Types and methods

You can define methods on *any* type:

```
type MyFloat float64

func (m MyFloat) Abs() float64 {
    f := float64(m)
    if f < 0 {
        return -f
    }
    return f
}

f := MyFloat(-42)
x := f.Abs()     // == 42.0
```

http://play

`Go` "objects" are just values. There is no "box".

Interfaces

Interfaces specify behaviors. An interface type defines a set of methods:

```
type Abser interface {
    Abs() float64
}
```

A type that implements those methods implements the interface:

```
func PrintAbs(a Abser) {
    fmt.Printf("Absolute value: %.2f\n", a.Abs())
}

PrintAbs(MyFloat(-10))
PrintAbs(Point{3, 4})
```

http://play

Types implement interfaces implicitly. There is no “implements” declaration.

Concurrency

In UNIX, we think about processes connected by pipes:

```
find ~/go/src/pkg | grep _test.go$ | xargs wc -l
```

Each tool designed to do one thing and to do it well.

- The Go analogue: goroutines connected by channels.

Concurrency: goroutines

Goroutines are like threads:

- they share memory.

But cheaper:

- Smaller, segmented stacks
- Many goroutines per operating system thread.

Start a new gorountine with the `go` keyword:

```
i := pivot(s)
go sort(s[:i])
go sort(s[i:])
```

Concurrency: channels

Channels are a typed conduit for:

- Synchronization.
- Communication.

The channel operator `<-` is used to send and receive values:

```
func compute(ch chan int) {
    ch <- someComputation()
}

func main() {
    ch := make(chan int)
    go compute(ch)
    result := <-ch
    fmt.Printf("result= %d\n", result)
}
```

Concurrency: synchronization

Look back at the sort example: how to tell when it's done ? Use a channel to synchronize goroutines:

```
done := make(chan bool)
doSort := func(s []int) {
  sort(s)
  done <- true
}

i := pivot(s)
go doSort(s[:i])
go doSort(s[i:])
<-done
<-done
```

Unbuffered channels operations are synchronous; the send/receive happens only when both sides are ready.

Concurrency: communication

A common task: many workers feeding from a task pool. Traditionally, worker threads contend over a lock for work:

```
type Task struct { /* some state */ }
type Pool struct {
  Mu    sync.Mutex
  Tasks []Task
}

func worker(pool *Pool) { // many of these run conc.
  for {
    pool.Mu.Lock()
    task := pool.Tasks[0]
    pool.Tasks = pool.Tasks[1:]
    pool.Mu.Unlock()
    process(task)
  }
}
```

Concurrency: communication

A `Go` idiom: many worker goroutines receive tasks from a channel.

```
type Task struct { /* some state */ }
func worker(in, out chan *Task) {
  for {
    task := <-in
    process(task)
    out <- task
  }
}
func main() {
  pending, done := make(chan *Task), make(chan *Task)
  go sendWork(pending)
  for i := 0; i < 10; i++ {go worker(pending, done)}
  consumeWork(done)
}
```

Concurrency: philosophy

- Goroutines give the efficiency of an asynchronous model, but you can write code in a synchronous style.
- Easier to reason about: write goroutines that do their specific jobs well, and connect them with channels.
  - in practice, this yields simpler and more maintainable code.
- Think about the concurrency issues that matter:

> *“Don’t communicate by sharing memory.*
> *Instead, share memory by communicating.*

Go: tooling

Great support for locating, retrieving, building and installing a package or command (and its dependencies) from any DVCS repository:

```
$ go get github.com/sbinet/igo
# now, igo can be used
$ igo
```

VERY fast compile/edit/run cycle (faster than C/FORTRAN/python !)

Go: tooling

Running tests is also integrated within the `go` tool:

```
$ cd github.com/hwaf/hwaf
$ go test -v
=== RUN TestHwafBoost
--- PASS: TestHwafBoost (3.20 seconds)
=== RUN TestHwafBoostBogusConfigureCmd
--- PASS: TestHwafBoostBogusConfigureCmd (2.51 seconds
=== RUN TestHwafBoostEmptyLib
--- PASS: TestHwafBoostEmptyLib (2.03 seconds)
[...]
=== RUN TestWorchVcs
--- PASS: TestWorchVcs (14.88 seconds)
=== RUN TestWorchAutoconf
--- PASS: TestWorchAutoconf (27.36 seconds)
PASS
ok      github.com/hwaf/hwaf    143.850s
```

Go: tooling

Want to quickly test a program ?

```
$ cat foo.go
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```

```
$ time go run foo.go
Hello, World

0.25s user 0.10s system 95% cpu 0.362 total
```

Rich library support

- Diverse, carefully-constructed, consistent standard library
  - More than 150 packages
  - Constantly under development, improving every day
  - Well documented: `http://golang.org/pkg`
- Many great external libraries too
  - documented on `http://godoc.org`
  - searchable on `http://go-search.org`
  - MySQL, MongoDB, SQLite3 database drivers,
  - SDL bindings, OpenGL, OpenAL,
  - Protocol Buffers,
  - OAuth,
  - HDF5, (and ROOT) bindings
  - ...

Go: what is it good for ?

- initially called a *systems language*.
- unexpected interest from users of scripting languages.
  - Attracted by an easy, reliable language that performs well.
- Diverse uses across the community:
  - scientific computing,
  - web applications,
  - graphics and sound,
  - network tools,
  - ...

Go: a general purpose language

Go is Open Source

- Development began at Google in 2007 as a 20% project.
- Released under a BSD-style license in November 2009.
- Since its release, more than 200 non-Google contributors have submitted over 1000 changes to the Go core.
- >10 Google employees work on Go full-time.

Resources

- http://golang.org
- http://golang.org/pkg
- http://golang.org/doc
- http://play.golang.org
- http://tour.golang.org
- http://blog.golang.org
- http://godoc.org
- http://go-search.org