Introducción a Google Go

13 de diciembre 2013

Antonio Nicolás Pina
Scalia

¿Por qué prefieres un lenguaje de scripting?

Razones para usar un lenguaje de scripting

- **Velocidad de desarrollo.**
- Expresividad.
- No hay necesidad de compilación.
- Recolección de basura.
- Abstracciones de alto nivel.
- ¿Despliegue?

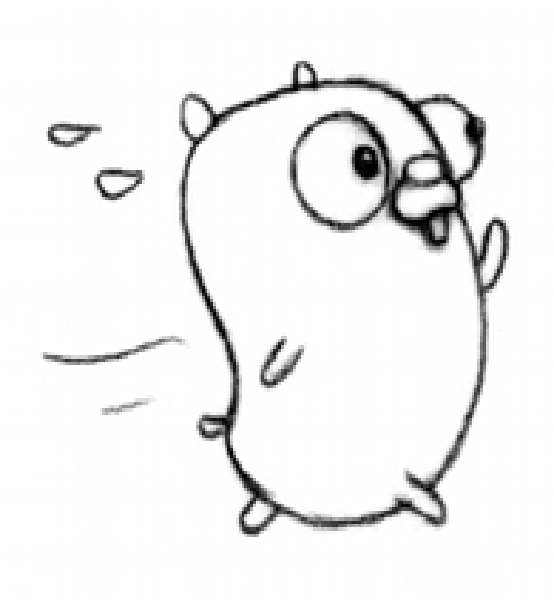
Razones para usar un lenguaje compilado

- **Velocidad de ejecución.**
- Tipado estático.
- Comprobaciones en tiempo de compilación.
- ¿Despliegue?

Y si...

- **Legible** y claro.
- Lenguaje compilado, pero de muy rápida compilación.
- Comprobaciones **estrictas** en tiempo de compilación.
- Fuertemente tipado.
- Recolección de basura.
- **Facilidad de despliegue.**
- Y muchas otras particularidades inusuales en los lenguajes más populares, útiles para el desarrollador.

Go 1.0 publicado en Marzo de 2012



Golang, un nuevo lenguaje

- No ha habido grandes avances en lenguajes de programación... ¡en casi una década!
- PHP nació en el año 1995
- Ruby también fue creado en 1995
- Node.js...

Go te ayuda a desarrollar correctamente

- Sintaxis concisa.
- Diseñado para ser eficiente en tiempo de desarrollo.
- Rápido en tiempo de ejecución.
- Es un lenguaje concurrente, y con un buen soporte para paralelización.
- Soporta tests y benchmarks de forma nativa.

Go te ayuda a desarrollar cómodamente

- Soporta nativamente Unicode.
- Recolector de basura.
- Devolución de múltiples valores.
- Las funciones son tipos de primer orden.
- Elimina lo que muchos consideran necesario: objetos e interfaces. En su lugar, presenta alternativas diferentes.

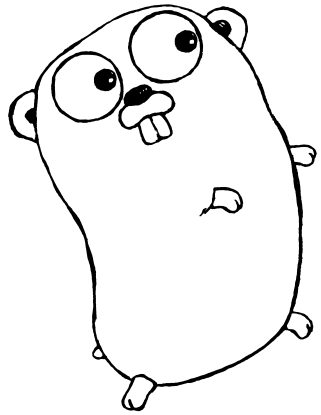
Go te obliga a programar bien

- Comprobaciones **muy** estrictas en tiempo de compilación.
- Comprobación estricta de tipos, no existen "tipos compatibles".
- No existe el tipo "float".
- Desde Go 1.0, no se garantiza el orden de iteración sobre mapas.

Go te insta a que tu código sea legible.

- No existen excepciones, sólo errores excepcionales.
- No se permiten imports o variables no usadas.
- Uso de punteros sin aritmética de punteros.
- Se prohiben expresiones que puedan resultar en código ilegible. Por ejemplo:

```
var2 = var++
```

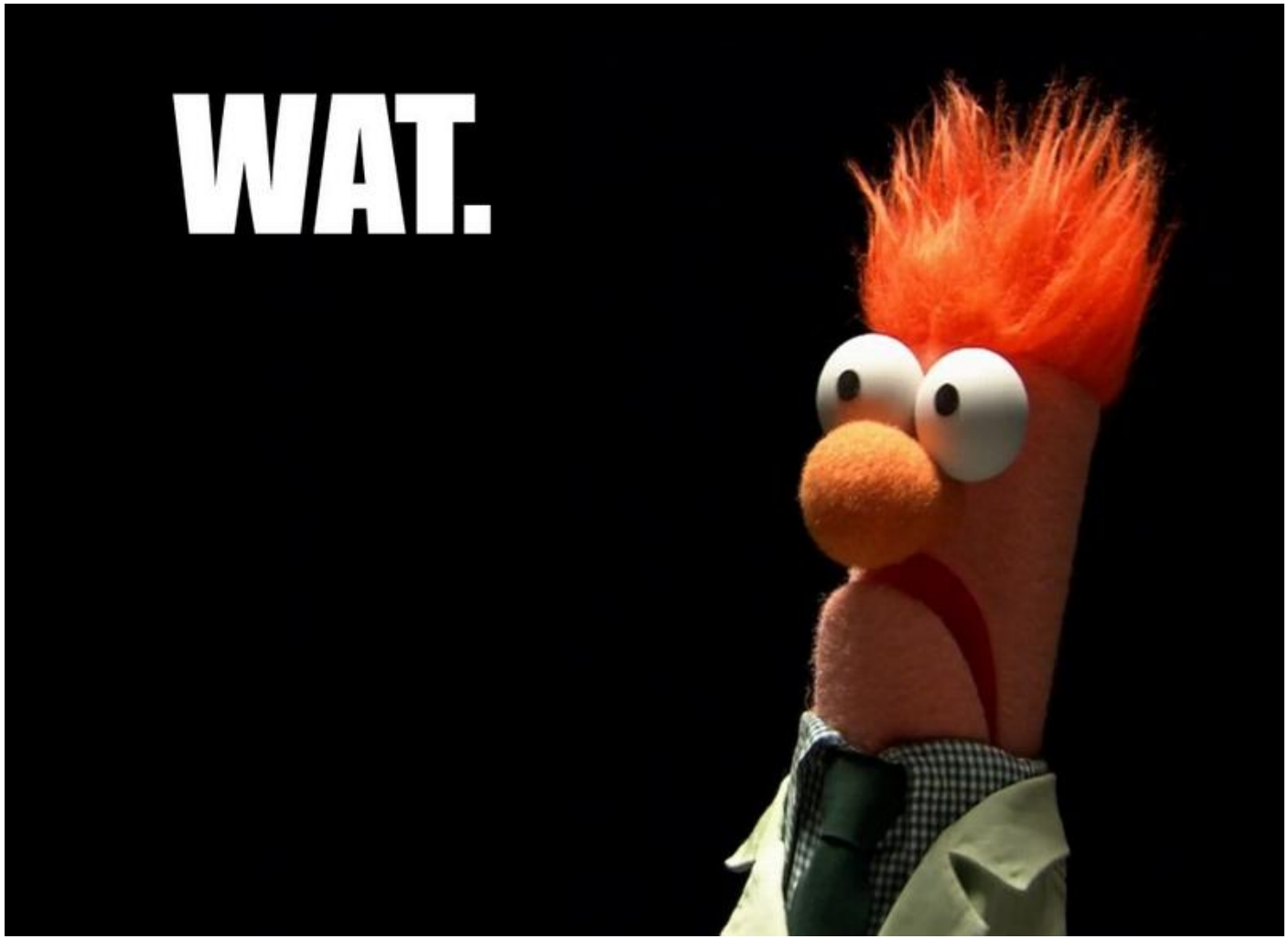


¿Cuál es tu lenguaje favorito?

Un ejemplo de C

```
int (*(*(*f3)(int))(double))(float);

          f3                          -- f3
         *f3                          -- is a pointer
        (*f3)(   )                    -- to a function
        (*f3)(int)                    --   taking an int parameter
       *(*f3)(int)                    --   returning a pointer
      (*(*f3)(int))(      )           --   to a function
      (*(*f3)(int))(double)           --     taking a double parameter
     *(*(*f3)(int))(double)           --     returning a pointer
    (*(*(*f3)(int))(double))(     )   --     to a function
    (*(*(*f3)(int))(double))(float)   --       taking a float parameter
int (*(*(*f3)(int))(double))(float)   --       returning int
```

StackOverflow (http://stackoverflow.com/questions/10758811/c-syntax-for-functions-returning-function-pointers)


WAT.

El mismo ejemplo en Go

```
func (int) func(float64) func(float32) int
```

VS

```
int (*(*(*f3)(int))(double))(float)
```

Un ejemplo de Java

```
public static final String readFile(String filename) {
  StringBuilder sb = new StringBuilder();
  try {
    BufferedReader br = new BufferedReader(new FileReader(filename));

    int read = 0;
    char[] buffer = new char[1024];
    while( -1 != (read = br.read(buffer, 0, buffer.length)) ) {
      sb.append(buffer, 0, read);
    }
    br.close();
  } catch (IOException e) {
    Log.e("HTTP", "readStream", e);
    Crittercism.logHandledException(e);
    return null;
  }

  return sb.toString();
}
```

El mismo ejemplo en Go

```
data, err := ioutil.ReadFile("file")
```

Sin usar ioutil:

```
func readFile(filename string) (string, error) {
    buf := bytes.NewBuffer(nil)

    f, err := os.Open(filename)
    if nil != err {
        return ``, err
    }
    defer f.Close()

    _, err = io.Copy(buf, f)
    if nil != err {
        return ``, err
    }

    return string(buf.Bytes())
}
```

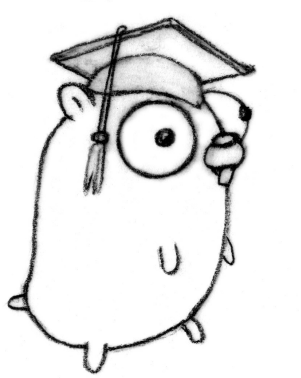
Golang soporta Unicode totalmente

```
package main

import "fmt"

func main() {
    ಠ益ಠ, ಠ_ಠ, 東京事変 := `Привет`, fmt.Println, `мир`

    ಠ_ಠ(ಠ益ಠ, 東京事変)
}
```

Run


Srsly?

Tipos de datos en Go



Tipos numéricos

int y bool

```
int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64.
```

- Desde Go 1.1, *int* es de 64 bits cuando se ejecuta en CPU de 64 bits.
- Personalmente, me gusta más especificar siempre el tamaño.

```
var x uint32 = 3
```

- O también:

```
x := uint32(3)
```

- Para booleanos: *true*, *false* y operadores *a la C*.

Ejemplos

Ejemplo incorrecto

```
func main() {
    var x uint32 = 42
    var y uint8 = x
    fmt.Printf("y vale %d.\n", y)
}
```

Run

¿Y ahora?

```
func main() {
    var x uint32 = 42
    var y uint = x
    fmt.Printf("y vale %d.\n", y)
}
```

Run

Más ejemplos

```
func main() {
    x := true
    if x == true {
        fmt.Println(`Verdadero`)
    } else {
        fmt.Println(`Falso`)
    }
}
```

Run

De otra forma:

```
func main() {
    switch {
    case true:
        fmt.Println(`es true`)
    case false:
        fmt.Println(`es false`)
    default:
        fmt.Println(`?`)
    }
}
```

Run

Otros tipos numéricos

- No se define un tipo *float*. En su lugar, existen *float32* y *float64*

```
var (
  x float32 = 2
  m float64 = 6.023e23
)
```

- Números complejos. Si, Go soporta números complejos: *complex64* y *complex128*.

```
func main() {
    var z complex128 = cmplx.Pow(math.E, -1i*math.Pi)
    fmt.Printf("z = %f.\n", z)
}
```

Run

array

Array

- Se puede declarar un *array* de cualquier tipo, nativo o no.

```
var (
  x [5]int32
  y [10][2]interface{}
)
```

- No son referencias, por lo que se copia la memoria al pasarlos a una función.
- **Recuerda**: Go **siempre** pasa los parámetros por valor.
- **Recuerda**: Go **siempre** inicializa **todo** a su zero-value.

Array

- Nos podemos ahorrar escribir el tamaño usando "..." cuando se especifica un literal.

```
func main() {
    var x [3]int
    y := [...]float32{3, 2, 4}
    fmt.Println(x, y)
}
```

Run

- ¡Re-slicing!

```
func main() {
    x := [...]int{1, 2, 3, 4, 5}
    fmt.Println(x[:3], x[2:], x[2:4])
}
```

Run

- Dado que el tamaño se especifica en la declaración, son **inmutables**.


RLY O?
WWW.ICANHASFORCE.COM

Array o slice, esa es la cuestión

- Go proporciona un *wrapper* sobre los arrays inmutables, lo que llama **slice**.

```
var x []int = make([]int, 3)
y := []int{1, 2, 3}
```

vs

```
var x [3]int
y := [...]int{1, 2, 3}
```

- Un slice es una referencia al array subyacente, por lo que siguen siendo inmutables.
- Dado que son una referencia, su zero-value es **nil**.
- Para crear un slice, se puede utilizar un literal, o bien utilizar la función **make()**.

Slice

- Utilizando **make()**, podemos especificar un tamaño y una capacidad inicial.
- Tamaño: Número de elementos que el slice contiene actualmente.
- Capacidad: tamaño del array subyacente.

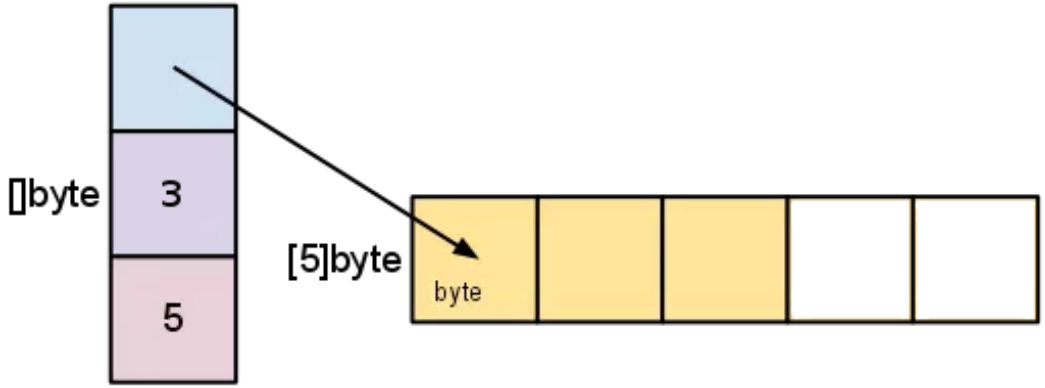
```
x := make([]float32, 3, 10)
```

- Por defecto, la capacidad se supone igual al tamaño, por lo que no es necesario indicarla.
- Para añadir uno o varios elementos, se utiliza la función **append**:

```
x = append(x, 2)
x = append(x, 3, 4, 5)
```

Diferencia entre array y slice

- Un slice también puede ser "re-sliced". Esto no cambia mas que el puntero hacia el array, no copia memoria.
- Ambos se comportan como se espera con las funciones **len**, **cap** y **copy**.
- Pero además, un slice nos ofrece la posibilidad de añadir elementos, ampliando la capacidad del array. Pista: **realloc**.



map

Mapas

- Como un HashMap de Java, o un Hash de Ruby.
- *Mapea* cualquier valor *comparable* a cualquier cosa.

```
var m map[string]interface{}
```

- Aquí, es clásico utilizar **interface{}** del mismo modo que **Object** en Java.
- Al igual que los array y slices, son iterables. ¡Pero **ojo**! **NO** se garantiza el orden.
- Del mismo modo que los *slices*, su zero-value es **nil**.

```
func main() {
    m := make(map[string][]int32)
    m[`hola`] = []int32{1, 2}
    m[`otro`] = []int32{3, 4}
    for key, value := range m {
        fmt.Println(key, value)
    }
}
```

Run

Mapas

```
const (
    CHULI     = 1
    KILOCHULI = 1e3 * CHULI
    MEGACHULI = 1e6 * CHULI
)

func main() {
    molonometer := make(map[string]int32)
    molonometer[`Cubert`] = 40 * MEGACHULI
    fmt.Println(molonometer[`Cubert`], molonometer[`Zoidberg`])
}
```

Run

O podemos comprobar si existe un mapping.

```
func main() {
    molonometer := map[string]int32{`Cubert`: 40 * MEGACHULI}
    if v, ok := molonometer[`Zoidberg`]; ok {
        fmt.Println(v)
    } else {
        fmt.Println(`Y U NO COOL`)
    }
}
```

Run

string

string

- Los *strings* son inmutables.

```
var variable string = "cadena"
```

- En el fondo, no es más que un []byte, por lo que puede contener **cualquier** contenido arbitrario.
- Como todos los tipos compuestos en Go, es iterable.
- En Go no hay *char*, hay *rune*, que representan codepoints unicode.
- *Rune* no es más que un alias de *int32*.

Operadores

- Los _string_'s **no** son referencias. **Recuerda**: sólo slices y maps lo son.
- Operaciones habituales a través de los paquetes *string* y *unicode*.
- Literales con "cadena" o cadena.

```
func main() {
    fmt.Println(`Esto es\nun string`)
    fmt.Println("Esto es\notro string")
}
```

Run

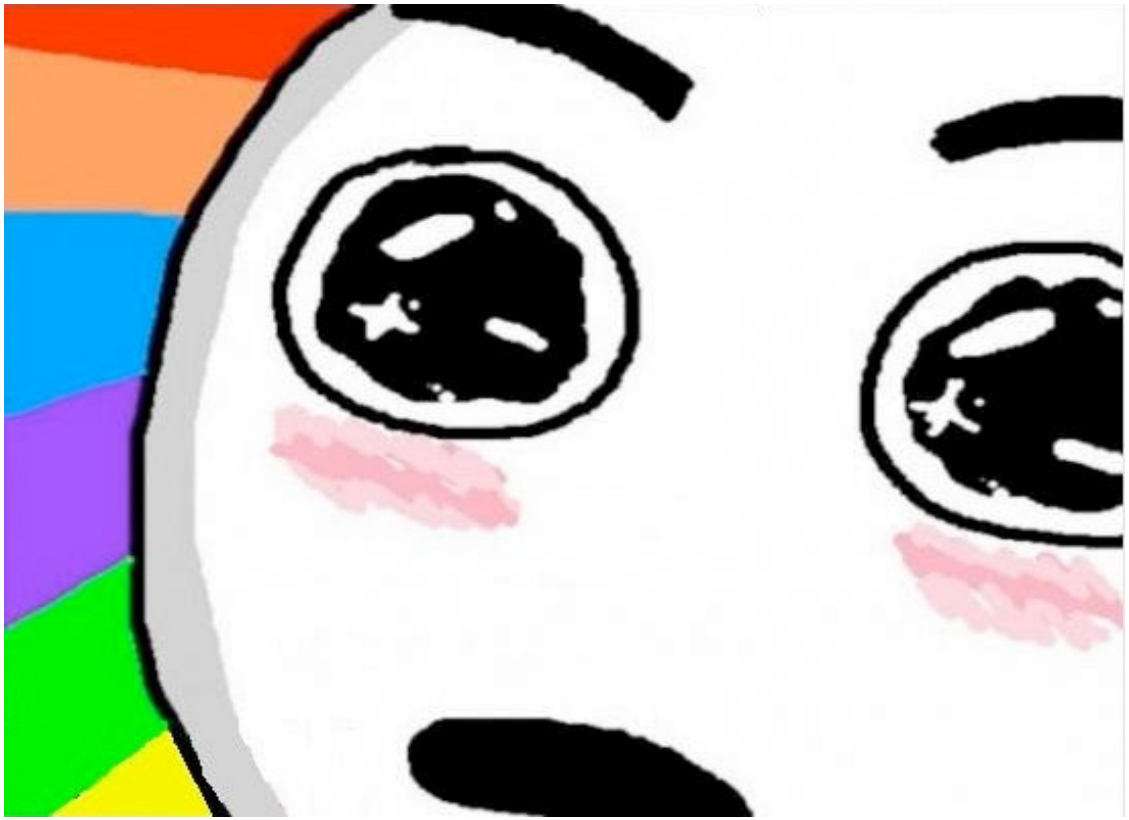
- Se puede iterar sobre ellos, rune a rune.

```
func main() {
    str := `Hola, привет!`
    for i, r := range str {
        fmt.Printf("str(%d)='%c'\n", i, r)
    }
}
```

Run

Probablemente, a estas alturas estáis...

Pero pronto estaréis...



O eso voy a intentar...

Las mejores cosas de Go

Duck typing

- "Si camina como un pato, nada como un pato y suena como un pato, es un pato."

```
type Stringer interface {
  String() string
}
```

- Cualquier tipo que implemente ese método "de facto", puede ser utilizado como un Stringer.

```
type Vector [3]int32

func (v Vector) String() string {
    return fmt.Sprintf(`v=(%d, %d, %d)`, v[0], v[1], v[2])
}

func main() {
    x := Vector{1, 2, 3}
    fmt.Println(x)
}
```

Run

Structs

- Combinado con los zero-values, los literales de structs son muy prácticos.

```
type resultType struct {
  Impressions  uint64  `json:"impressions"`
  Views        uint64  `json:"views"`
  Interactions float32 `json:"interactions,omitempty"`
}
ret := resultType{Impressions:imp}
```

- **Cuidado**: Los campos de structs que comienzan por mayúscula, se consideran "public", y el resto, "private".
- Además, como en el ejemplo, se pueden "anotar", para definir otras propiedades como, por ejemplo, el nombre del campo cuando se convierta a JSON.

Y ahora algo totalmente diferente

¡Structs!

- Aunque Golang no es un lenguaje orientado a objetos, pero aún así permite "herencia múltiple", el llamado "embedding".

```
type Animal struct {
  Nombre string
}
type Mascota struct {
  Achuchable bool
}
type Perro struct {
  Animal
  Mascota
  Raza TipoRaza
}
```

- Podemos invocar cualquier método definido sobre "Animal" sobre un objeto de tipo Perro, así como acceder directamente a sus campos.

```
p := Perro{Nombre: "Yaky", Achuchable = true}
```

Funciones

- Como hemos visto, es posible escribir funciones que devuelvan varios parámetros.
- No sólo es posible, sino que es ampliamente utilizado y se considera parte del "Go-style".
- Además, estos valores de retorno pueden tener opcionalmente un nombre.

```
func pow(x float64) (x2 float64, x3 float64) {
    x2 = x * x
    x3 = x2 * x
    return
}
func main() {
    fmt.Println(pow(2))
}
```

Run

Funciones

- Del mismo modo, se pueden declarar y asignar varias variables a la vez.

```
func pow(x float64) (x2, x3 float64, e error) {
    if x > 10 {
        return 0, 0, fmt.Errorf("El valor x=%f es demasiado grande", x)
    }

    return x*x, x*x*x, nil
}

func main() {
    x, y, err := pow(4)
    if err != nil {
        fmt.Printf("ERROR: %v.\n", err)
    } else {
        fmt.Println(x, y)
        x, y = y, x
        fmt.Println(x, y)
    }
}
```

Run

Closures

- Una de las mejores características del lenguaje, es su definición de las funciones como un tipo de dato más.
- Además, las funciones "literales" son *closures*, de modo que capturan cualquier variable que haya en su *scope* y pueden acceder a ella como si fuera propia.

```
func Generator(x int) func() int {
    return func() int { return x }
}
func main() {
    f := Generator(3)
    fmt.Println(f())
}
```

Run

Closures

- Se pueden utilizar también para agrupar código, aunque no sea estrictamente necesario, a veces queda más limpio.

```
addIfExists := func(name, param string) {
  value, ok := post[param]
  if ok {
    match[name] = bson.M{`$in`: value}
  }
}

addIfExists(`sex`, `sex`)
addIfExists(`mstatus`, `mstatus`)
addIfExists(`influence_zones`, `influence`)
addIfExists(`type`, `proposal_types`)
addIfExists(`catid`, `categories`)
addIfExists(`comid`, `commerces`)
```

Y ahora algo totalmente diferente

Defer

```
status     := httpClient.Status()
resultJson := httpClient.Result()

if nil != httpClient.Error() || 200 != status {
  httpClient.Close()
  return
}

var result []interface{}
if err := json.Unmarshal(resultJson, &result); nil != err {
  httpClient.Close()
  return
}

if ! result[`success`] {
  httpClient.Close()
  return
}
```

Defer

```
status, resultJson, err := httpClient.Result()
defer httpClient.Close()

if nil != err || 200 != status {
  return
}

var result []interface{}
if err := json.Unmarshal(resultJson, &result); nil != err {
  return
}

if ! result[`success`] {
  return
}
```

Concurrencia

- Go provee de diversos mecanismos para manejar la concurrencia de forma adecuada.
- Sobre todo, haremos uso de go-routinas y canales.
- La filosofía es "Share memory by communicating".
- Es importante no confundir concurrencia con paralelismo.
- Linux hasta la versión 2.0 no era paralelo, aunque si era concurrente.
- El siguiente programa NO es paralelo, aunque sí concurrente.

```
func main() {
    c := make(chan int)
    go func() { c <- 1 }()
    go func() { c <- 2 }()
    x, y := <-c, <-c
    fmt.Println(x, y)
}
```

Run

Un ejemplo real

```
func (exec *ParallelExecutor) Exec(f ParallelFunc) {
  c := make(chan ParallelResponse)
  conn, db := GetDBConn()

  go func() {
    defer func() {
      if r := recover(); nil != r {
        err, ok := r.(error)
        if !ok {
          err = fmt.Errorf("pkg: %v", r)
        }
        c <- ParallelResponse{nil, err}
      }
    }()
    defer conn.Close()

    result, err := f(db)
    c <- ParallelResponse{result, err}
  }()

  exec.channels = append(exec.channels, c)
}
```

Y ahora algo totalmente diferente

Entorno Go

- run: Ejecutar un programa "sin compilarlo".
- vet: Buscar "problemas", algo parecido a findbugs.
- fix: Reescribir sentencias para utilizar APIs más "modernas", se utiliza al actualizar Go a una nueva versión.
- fmt: ¿Espacios o tabs? ¡Nunca más! Formatea con "go fmt".
- pprof: ¿Donde pierde más tiempo mi programa? ¿Dónde consume más RAM?
- doc: Generación automática de documentación.
- test: Ejecutar los tests. También ejecuta benchmarks al pasarle el flag "-bench".
- get: Descargar dependencias de un proyecto.
- Flag "-race": Detector de condiciones de carrera, desde Go 1.1.

Deploy

- En Go se recomienda colocar la URL en el import. De esta forma, "go get" puede descargar automáticamente las dependencias y colocarlas en el $GOPATH.

```
import (
  log `github.com/cihub/seelog`
  `labix.org/v2/mgo`
)
```

- Golang compila estáticamente las dependencias.
- El resultado es un ELF (o .exe, o...) listo para ejecutar en cualquier máquina, sin instalar nada más.

¿Y esto se usa?

- dl.google.com es un servidor Go.
- Docker. Administrador de "containers" LXC.
- Packer. Herramienta para crear imágenes de máquinas virtuales automáticamente.
- Youtube. Liberó y utiliza Vitess, una herramienta Go para escalar MySQL.
- 10gen. El servicio de backups de MongoDB está escrito en GO.
- Bitly. Han liberado NSQ, una plataforma de mensajería en tiempo real.
- pool.ntp.org. Ya dispone de servidores hechos en Go.
- Y muchos, muchos otros.

code.google.com/p/go-wiki/wiki/GoUsers (https://code.google.com/p/go-wiki/wiki/GoUsers)

Thank you

Antonio Nicolás Pina
Scalia
antonio@scalia.es (mailto:antonio@scalia.es)
http://scalia.es/ (http://scalia.es/)
@ANPez (http://twitter.com/ANPez)
@scalia_es (http://twitter.com/scalia_es)