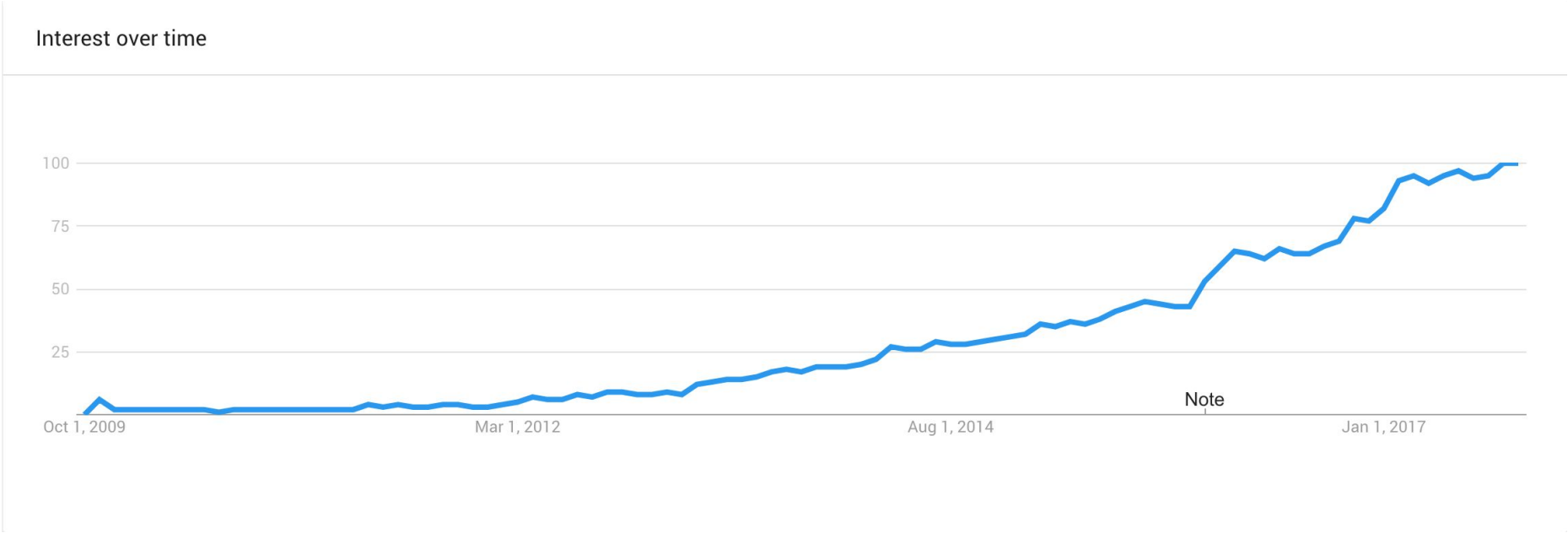
Go: Your Next Backend Language

Waqqas Sheikh
Software Engineer at TribalScale

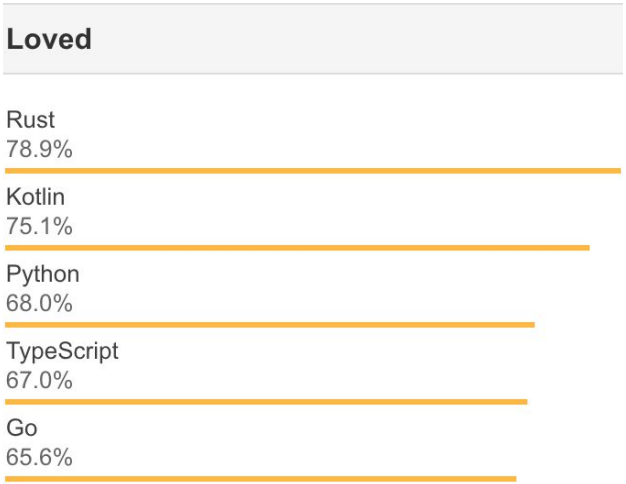
github.com/w-k-s @WaqqasTheWicked

Rising Interest in Go



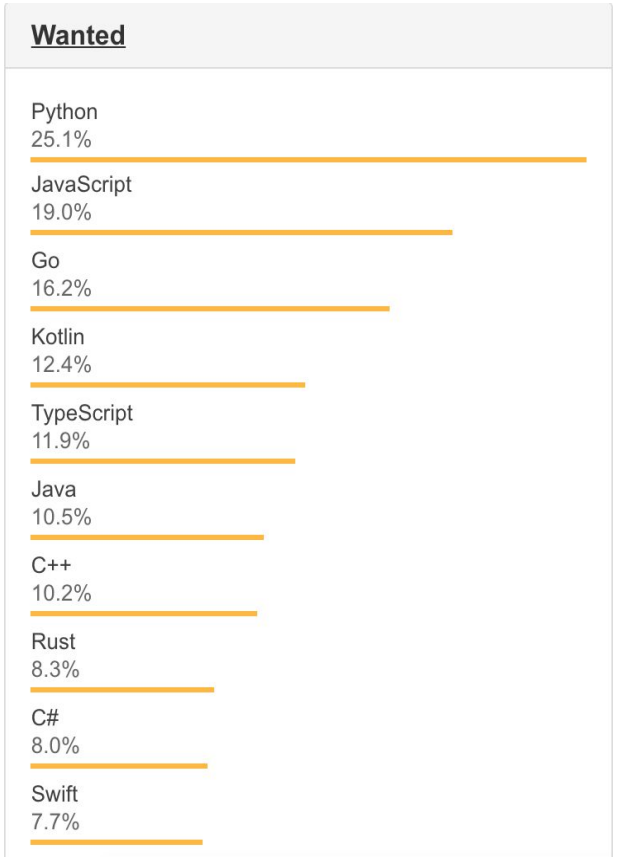
Google Trends

Top 5 most loved languages of 2018



StackOverflow Survey 2018

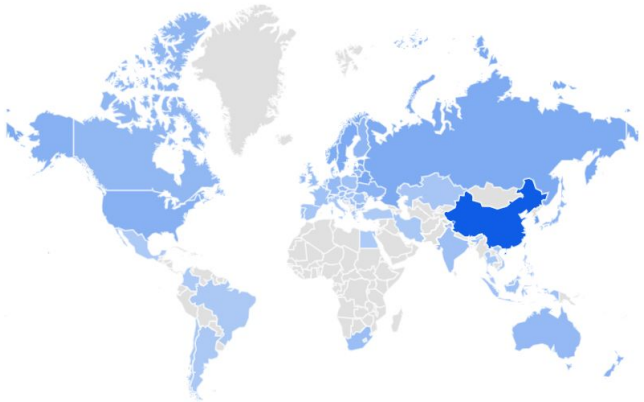
Top 3 most wanted languages of 2018



StackOverflow Survey 2018

Go’s increasing popularity around the world

Interest by region



1	China	100
2	Singapore	47
3	Hong Kong	46
4	South Korea	40
5	Estonia	32

Google Trends

List of companies using Go

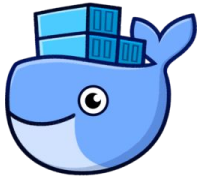
Companies using Go


Google


Uber


Twitter


Dropbox


Docker


SoundCloud


Slack


Netflix


Facebook


Github


Shopify


Square


Reddit


Medium


SpaceX

Namshi

...and many more!

What makes Go so cool?

What makes Go cool?

1. **Scalable Concurrency**

   Go's goroutines are very lightweight (thousands of them can run inside a single OS thread)

   "*Do not communicate by sharing memory; instead, share memory by communicating*"

2. **Easy Deployment**

   Source code and all of its dependencies are compiled (quickly) into a single binary that's ready to deploy.

   Compiler supports cross-compilation.

3. **Feature-light**

   Go is very minimal when it comes to language syntax and features.

   Developers can get up to speed with it quickly and immediately become productive.

4. **Performance**

   Very fast compilation.

   Faster performance compared to interpreted languages (Python, JavaScript etc.)

What's Inside

Goals of this talk

- Get you interested and excited about Go
- Try it out for yourself, make your own Go web app
- Consider Go for your next project

Motivation for Go

Authors

- Go was developed at Google starting in 2009.
- Go v1.0.0 was released on March 2012

Ken Thompson

Co-creator of C and UNIX

Rob “Commander”Pike

Co-creator of UTF-8, Plan 9 OS

Robert Griesemer

Distributed Systems Developer @Google

Motivation

The state of programming languages in 2009

Languages	Performance	Verbosity	Complexity
Python/PHP	Low	Low	Low
NodeJS	Medium	Low	Low
Java/C#	Medium	High	Medium
C++	High	Medium	High
C	High	Low	High
???	High	Low	Low

Tech Talk: Public static void

Goals

The Goals for the “Go” Programming language were:

1. **Statically typed**

   The lack of a type system is seen as a limitation in scripting languages such as JavaScript.

2. **Readable and minimal boilerplate**

   Enterprise languages such as Java, C# typically involved too much boilerplate code for routine tasks

3. **Scalable for large systems**

   Go had to be lightweight yet performant like C, so that it could cope with large scale.

4. **Designed for multi-processing**

   In 2009, multi-core processors existed but few languages were taking advantage of the capability.

Tech Talk: Simplicity is complicated

Languages	Performance	Verbosity	Complexity
Python/PHP	Low	Low	Low
NodeJS	Medium	Low	Low
Java/C#	Medium	High	Medium
Go	Medium	Medium	Low
C++	High	Medium	High
C	High	Low	High

Source

Hello World

Hello World

```
package main

import "fmt"

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

Run

```
go run main.go
```

Language Overview

Language Features

01 The Syntax

02 Concurrency

03 Networking

04 Garbage Collection

05 Tooling

Let’s get to know the Syntax

Declaring Variables

1. **Full declaration**

   Variables are declared with the keyword `var` followed by the name and type:

```
package main

func main() {
      var age int = 9
      fmt.Println(age)
}
```

2. **Type inference**

   The type can be omitted wherever it can be inferred:

```
package main

func main() {
      var age = 9
      fmt.Println(age)
}
```

Declaring Variables

3. **Dot Colon Operator (:=)**
   Declares and initializes the variable in one go (pun-intended).

```
package main

func main() {
      age := 9
      fmt.Println(age)
}
```

Exported and Non-exported symbols

- Case matters!
- Any symbol that starts with an **Uppercase** letter is **exported** (available outside its package i.e. **public**)
- Any symbol that starts with a **lowercase** letter is **non-exported** (**private** to the package)

```
package foo

import "fmt"

const helloWorld = "HELLO_WORLD"

func PrintHelloWorld(){
      fmt.Println(helloWorld);
}
```

```
package main

import (
      "foo"
      "fmt"
)

const hello = “HELLO”

func main() {

      foo.PrintHelloWorld()            //Works
      fmt.Println(hello)               //Works
      fmt.Println(foo.helloWorld)      //Won’t work

}
```

If Statements

- No parenthesis
- You can **declare a variable inside the if statement**, just like you would in a for-loop statement.

  (Really handy when working with maps)

Run

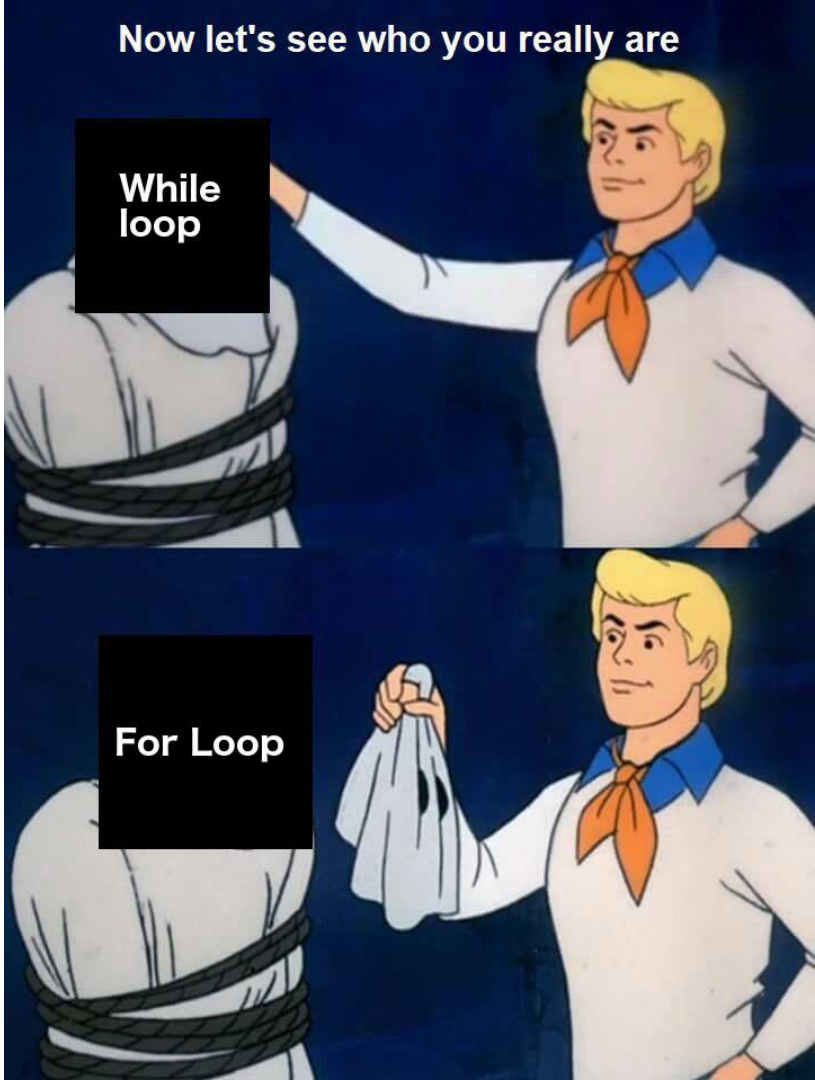
```
package main

import "fmt"

const drivingAge = 18

func main() {
	age := 9

	if canDrive := age >= drivingAge; canDrive {
		fmt.Println("Would you like to buy a car?")
	} else {
		fmt.Println("Would you like to buy a toy?")
	}
}
```

For loops

- Go has only one kind of loop: **for-loops**
- The reason being, all other loops can be easily represented with a for-loop.

Run

```
package main

import "fmt"

func main() {
	fmt.Println("Odd Numbers")

	for i := 0; i < 10; i++ {
		if odd := i%2 != 0; odd {
			fmt.Println(i)
		}
	}
}
```



Arrays, Slices & for-range

- Go has a few built-in generic container types
- **Arrays**: Fixed-size storage for items of the same type

```
var grades [5]string
grades2 := [5]string{"A","B","C","D","E"}
```

- **Slices**: Dynamically sized storage for items of the same time

```
var grades []string
grades2 := []string{"A","B","C","D","E"}
grades2 = append(grades2,”F”)
```

- **for-range** loops can be used to iterate over containers

```
numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

for index, number := range numbers {
      fmt.Printf("%d + 1 = %d\n", index, number)
}
```

Run

Maps

- **Map**: Key-value storage for items of the same type

Run

```
func main() {
	codes := map[string]string{
		"United Arab Emirates": "AE",
		"United States":        "US",
		"United Kingdom":       "UK",
	}

	country := "Kuwait"

	if code, found := codes[country]; found {
		fmt.Printf("%s's country code is %s.\n", country, code)
	} else {
		fmt.Printf("We don't know %s's country code.\n", country)
	}

	for key, value := range codes {
		fmt.Printf("%s:\t%s\n", key, value)
	}
}
```

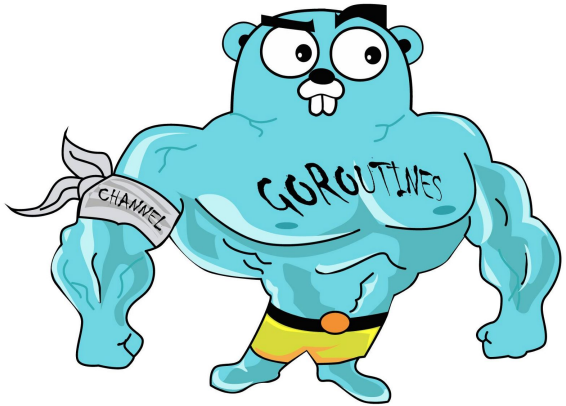
Multiple Returns

- Functions in go can return multiple values
- Handled at the assembly level.[1]
- Typically used to return an error or a success-boolean

Run

```
package main

import (
	"fmt"
	"strings"
)

func splitName(fullName string) (string, string) {

	names := strings.Split(fullName, " ")
	return names[0], names[1]
}

func main() {
	fullName := "John Smith"

	firstName, lastName := splitName(fullName)

	fmt.Printf(
		"My first name is '%s' and my last name is '%s.\n",
		 firstName,
		lastName,
	)
}
```

And now for the interesting stuff



Concurrency

Goroutines

Goroutines are functions that run concurrently with other functions.
Not the same as threads! You can think of them as mini-threads… *very* mini-threads!

Cheap

- Cost of goroutine is 2kb of stack space. A java thread costs around 1MB![1]

Lightweight

- Goroutines are grouped into a single OS thread.
  You can have as many as a 1000 goroutines in a single thread[2]

Non-blocking

- If one goroutine blocks (e.g. for I/O), the remaining goroutines are moved to a new OS thread.

Parallel Execution

- Go will take advantage of multiple cores if they are available[3]

Concurrency has never been easier

- To start a new goroutine, simply prepend the function call with **go**

Run

```
package main

import (
	"fmt"
	"time"
)

func numbers() {
	for i := 1; i <= 5; i++ {
		time.Sleep(250 * time.Millisecond)
		fmt.Printf("%d ", i)
	}
}

func alphabets() {
	for i := 'a'; i <= 'e'; i++ {
		time.Sleep(400 * time.Millisecond)
		fmt.Printf("%c ", i)
	}
}

func main() {
	go numbers()
	go alphabets()
	time.Sleep(3000 * time.Millisecond)
	fmt.Println("Done!")
}
```

Communicating between goroutines

- **Channels** are used to send messages to functions running on seperate goroutines.
- A **channel** passes a message of a given type

1. **Create** a channel

```
myChannel := make(chan string)
```

2. **Pass** a channel into the goroutine function

```
go loadWeather(myChannel)
```

3. **Send a message** from within the goroutine function

```
myChannel <- “It’s going to be a bright… bright... sunny day”
```

4. **Wait for the message** outside the goroutine

```
fmt.Println( <- myChannel)
```

```go
package main

//...imports

const API_KEY = "e7ae82efd03a0ff7b280ca934b105c65"

func loadWeather(c chan string) {
	resp, err := http.Get("http://api.openweathermap.org/data/2.5/weather?q=dubai&appid=" + API_KEY)
	if err != nil {
		panic(err)
	}
	bytes, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		panic(err)
	}
	c <- string(bytes)
}

func countMilliseconds() {
	for counter := 1; ; counter++ {
		time.Sleep(1 * time.Millisecond)
		fmt.Println(counter)
	}
}

func main() {
	weatherChannel := make(chan string)
	go loadWeather(weatherChannel)
	go countMilliseconds()
	fmt.Println(<-weatherChannel)
}
```

Communicating with multiple channels

- The **select** statement lets you wait for messages from multiple channels at once.
- It works a lot like a switch statement.

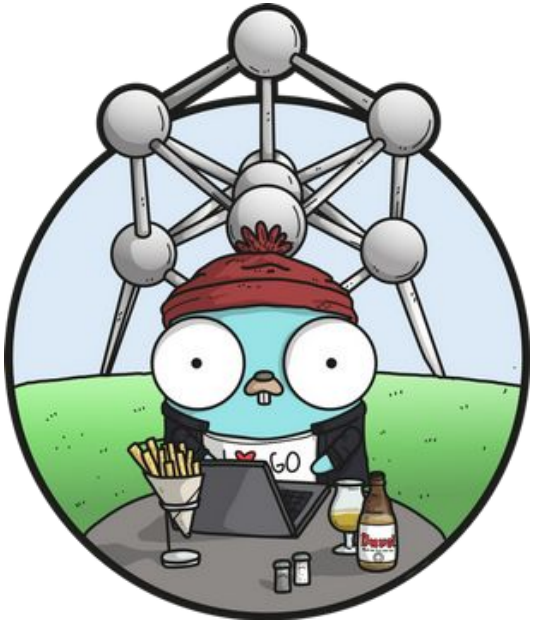
```
select {
  case message1 := <- channel1:
    //do something
  case message2 := <- channel2:
    //do something
}
```

- A case statement is triggered whenever its channel sends a message.

Real world example

Run

```
package main

func server1(ch chan string) {
	time.Sleep(6 * time.Second)
	ch <- "from server1"
}
func server2(ch chan string) {
	time.Sleep(3 * time.Second)
	ch <- "from server2"

}
func main() {
	output1 := make(chan string)
	output2 := make(chan string)
	go server1(output1)
	go server2(output2)

	var s1 string
	var s2 string
	for {
		select {
		case s1 = <-output1:
			fmt.Println(s1)
		case s2 = <-output2:
			fmt.Println(s2)
		}
		if len(s1) != 0 && len(s2) != 0 {
			break
		}

	}
}
```



Networking

The net/http package

- `net/http` is Go's very powerful http package.
- Enables you to build a modern backend application without the need of any third-party framework!

Includes:

- HTTP and HTTPS server
- Static File Server
- Request router with pattern matching
- HTML templating engine
- Full HTTP/2 support

- Go also has an experimental golang/x/autocert package that automatically acquires and updates SSL certificates from LetsEncrypt!

```
package main

import (
	// Don't ever write imports like this
	"fmt"; "io"; "log"; "net/http"; "strings"
)

func indexHandler(w http.ResponseWriter, req
*http.Request) {
	io.WriteString(w, "Hello, world!\n")
}

func greetHandler(w http.ResponseWriter, req
*http.Request) {

	name := strings.TrimPrefix(req.URL.Path,
"/greet/")

	var message = "You didn't tell me your name!\n"

	if len(name) > 0 {
		message = fmt.Sprintf("Hello, %s\n", name)
	}

	io.WriteString(w, message)
}

func main() {
	handler := http.NewServeMux()
	handler.HandleFunc("/greet/", greetHandler)
	handler.HandleFunc("/", indexHandler)
	log.Fatal(http.ListenAndServe(":8080", handler))
}
```

HTTP/2.0 Push

- HTTP 2.0 supports **server-side push**
- Server can **push related files** it knows that the client will need.
  E.g. index.min.css and index.min.js with index.html
- Page loads a lot faster!

HTTP/2.0 Push

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {

      //Try to case ResponseWriter to Pusher. If ok, client supports push
    if pusher, ok := w.(http.Pusher); ok {

        // Push is supported.
        if err := pusher.Push("/index.min.js", nil); err != nil {
            log.Printf("Failed to push: %v", err)
        }
    }

})
```

See the difference!

The encoding package

- Go's encoding package supports encoding into several human-readable formats:
  - XML
  - JSON
  - CSV
- Simply **tag** the name of the fields in your struct and make sure they're **public**
- Example

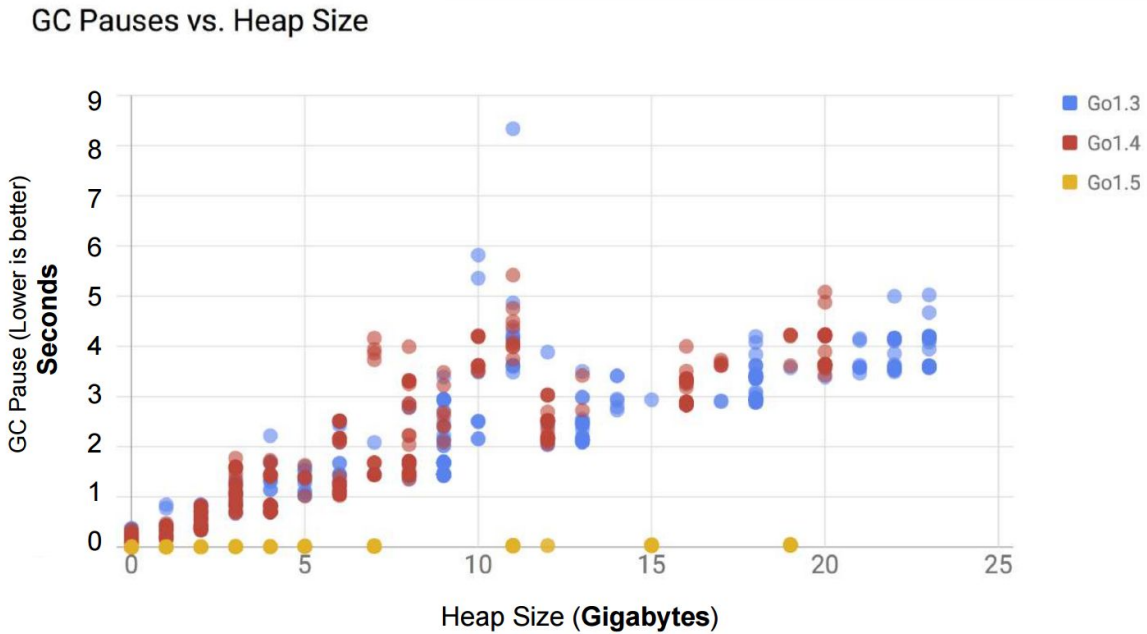
Garbage Collection

Go’s Garbage Collector

- Go is **garbage-collected**. You can use pointers without worrying about releasing memory.
- Typically, **garbage collection does not scale well** because garbage collectors pause programs to collect memory. This is called **stop-the-world**.
- Stop-the-world is unavoidable and it **causes latency.**
- Go attempts to reduce the duration of stop-the-world**.**
- **Go’s garbage collector runs concurrently** with the program, looking for unreachable objects.[1]
- There are **2 stop-the-worlds per cycle**, each one lasting for **less than 1ms** (and this is for **very large** heaps)

Google

Garbage Benchmark



Language Tools

gofmt

- gofmt is a tool that automatically formats go source code according to standard specs
- All go code looks the same, easier to read, no arguments amongst developers!

```
gofmt -w /path/to/package/or/file
```

Gofmt's style is nobody's favourite, but gofmt is everybody's favourite
\- *Go Proverbs*

Other go tools

- dep: Dependency management, as powerful as npm or pip
- go run -race: detects data race conditions while your code is running
- go fix: This tool prints warnings about API changes (e.g. function deprecations e.t.c) and makes the changes itself where possible. You should run this tool everytime you update your golang installation.
- go test: executes unit tests

  `go test -coverprofile`: Generates browser friendly unit test coverage report

- gdb: GNU Debug bridge can be used for debugging.

Language Limitations

No Generics ... yet

- One of the most **heavy criticisms** of Go is that it lacks generics
- Authors of Go wanted to find a clean way of implementing generics.
- Makes **writing custom containers difficult** e.g. Sets, Trees etc.
- Go developers have resorted to **copy-pasting** code with different types (actual demo)
- Generics are **planned for Go2.**

Go2 Generics

```
package main

import "fmt"

// A contract is a block that lists statements that
// should compile without error for the given type T.
contract addable(t T){
      t += t
      t *= 2
}

func total (type T addable) (numbers ...T) {
      var total T = 0
      for _, number := range numbers {
            total += number
      }
      return total
}

func main() {
      fmt.Println(total(1, 2, 3))
      fmt.Println(total(1.0, 2.0, 3.0))
      //fmt.Println(total("hello", "world"))
}
```

```
type List(type Element) struct {
      next *List(Element)
      val  Element
}
```

Error Handling

- Go does not support exceptions.
  Instead, functions return a value and error.
- Developers **must check error is not nil**.
- Very repetitive!
- Go2 introduces the **`check`** keyword and **`handle`** block to cut down on the boilerplate.

```
package main

import(
	"net/http"
	"log"
	"os"
)

func main() {
	resp, err := http.Get("http://google.com/")
	if err != nil {
		log.Panicf("request error: %s",err.Error())
	}

	f, err := os.Create("output.txt")
	if err != nil {
		log.Panicf("file create error: %s",err)
	}
	defer f.Close()

	err = resp.Write(f)
	if err != nil{
		log.Panicf("file write error: %s",err)
	}
}
```

Go v1

```
package main

import(
	"net/http"
	"log"
	"os"
)

func main() {
	resp, err := http.Get("http://google.com/")
	if err != nil {
		log.Panicf("request error: %s",err.Error())
	}

	f, err := os.Create("output.txt")
	if err != nil {
		log.Panicf("file create error: %s",err)
	}
	defer f.Close()

	err = resp.Write(f)
	if err != nil{
		log.Panicf("file write error: %s",err)
	}
}
```

Go v2

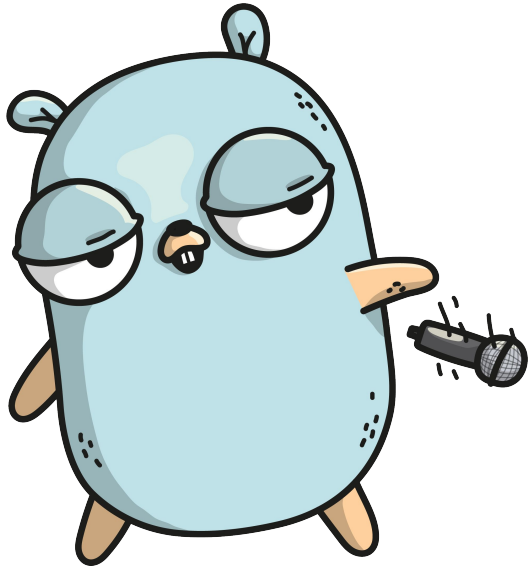
```
package main

import(
	"net/http"
	"log"
	"os"
)

func main() {

	handle err {
		log.Panicf("request error: %s",err.Error())
	}

	resp := check http.Get("http://google.com/")

	f := check os.Create("output.txt")
	defer f.Close()

	check resp.Write(f)
}
```

Go nuts!

What's Next?

What's Next

1. Take the tour
2. Build your own Go web app
3. Read Effective Go and watch Go Proverbs to learn about best practices.
4. Check out open source Go projects
5. Benchmark go
6. See what's planned for Go 2



Waqqas Sheikh

Software Engineer at TribalScale

www.tribalscale.com | @tribalscale

github.com/w-k-s @WaqqasTheWicked