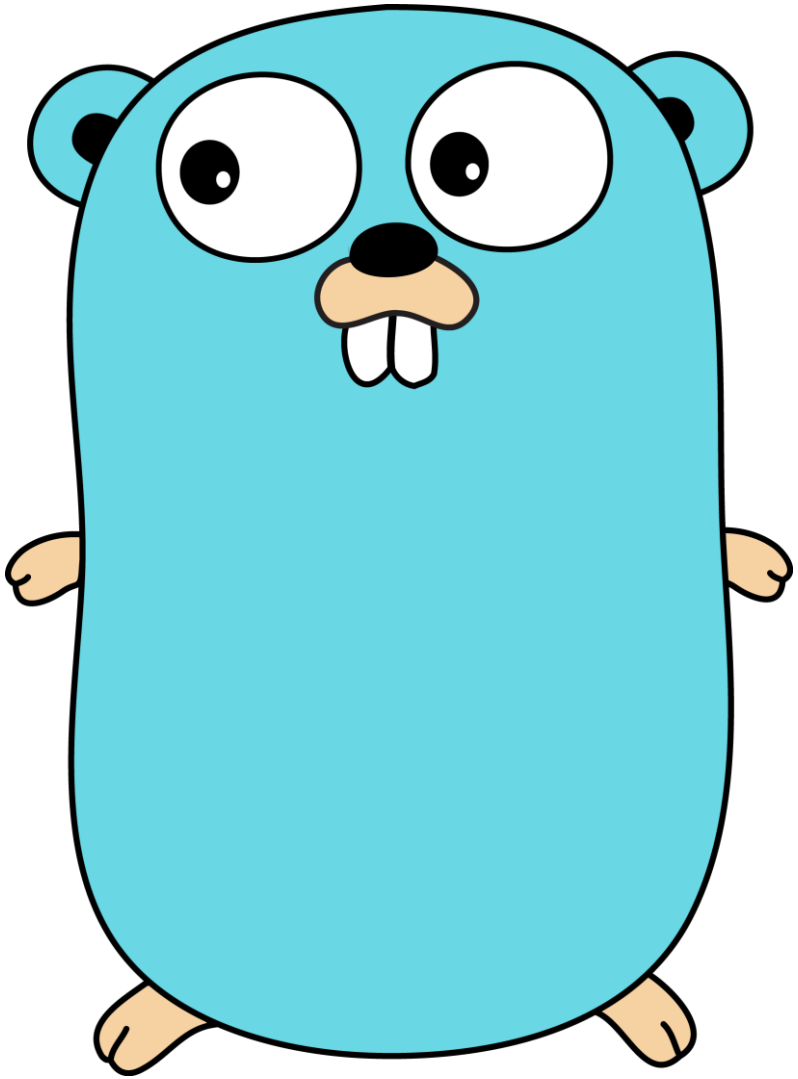


FTD Golang

What is Go?

Go is:

- open source
- concurrent
- garbage-collected
- efficient
- scalable
- simple
- fun
- boring (to some)

http://golang.org

History

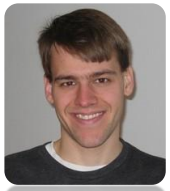
Design began in late 2007.

Became open source in November 2009.

Today's version is 1.2.2.

Key Players:

Ian Lance Taylor
(GCC)

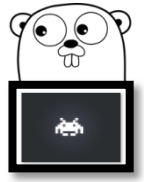


Russ Cox
(Plan9/CSP)

Robert Griesemer
(Hotspot JVM)

Rob Pike (58)
(Unix/UTF-8)

Kenneth Thompson (71)
(B and C lang/Unix/UTF-8)



Google has big problems

Go is a programming language designed by Google to help solve Google's problems.

What problems?

- C++ for servers, plus lots of Java and Python
- thousand of engineers
- gazillions of lines of code
- distributed build system
- zillions of machines, which are treated as a modest number of compute clusters

The reason for Go

Goals:

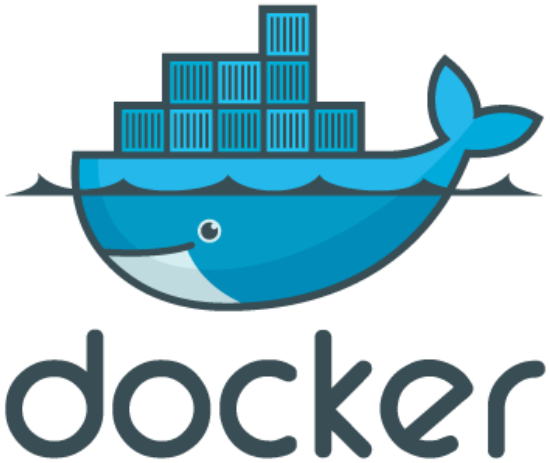
- eliminate slowness
- eliminate clumsiness
- improve effectiveness
- maintain (even improve) scale

Go was designed by and for people who write – and read and debug and maintain – large software systems.

Go's purpose is not research programming language design.

Go's purpose is to make its designers' programming lives better.

Who uses?


docker


juju


SOUNDCLOUD


YouTube


tumblr.


Google


CANONICAL


amazon.com


heroku


NOKIA


bitly


vimeo


mozilla
FOUNDATION


globo.com

Important stuff

- Object oriented without inheritance
- Syntax sugar declaration
- Strong and Static types
- Interfaces, but no declaration (duck typing)
- Functions and Methods (with receivers)
- No exceptions (error interface)
- **Visibility with the first letter of symbol**
- Unused imports and variables occurs compile error
- Excellent and complete standard library

Go is a tool

- Go is a tool for managing Go source code.

build	compile packages and dependencies
clean	remove object files
env	print Go environment information
fix	run go tool fix on packages
fmt	run gofmt on package sources
get	download and install packages and dependencies
install	compile and install packages and dependencies
list	list packages
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	run go tool vet on packages

Cross compilation

Needs execute $GOROT/src/make

$GOOS	$GOARCH	
darwin	386	-- 32 bit MacOSX
darwin	amd64	-- 64 bit MacOSX
freebsd	386	
freebsd	amd64	
linux	386	-- 32 bit Linux
linux	amd64	-- 64 bit Linux
linux	arm	-- RISC Linux
netbsd	386	
netbsd	amd64	
openbsd	386	
openbsd	amd64	
plan9	386	
windows	386	-- 32 bit Windows
windows	amd64	-- 64 bit Windows

C? Go? Cgo!

- Cgo lets Go packages call C code. Given a Go source file written with some special features, cgo outputs Go and C files that can be combined into a single Go package.
- Go's use SWIG.

```
package cgoexample
/*
	#include <stdio.h>
	#include <stdlib.h>

	void myprint(char* s) {
		printf("%s", s);
	}
*/
import "C"

import "unsafe"

func main() {
	cs := C.CString("Hello from stdio\n")
	C.myprint(cs)
	C.free(unsafe.Pointer(cs))
}
```

How to Write Go Code

- Go code must be kept inside a workspace. A workspace is a directory hierarchy with three directories at its root:
  - ***src*** contains Go source files organized into packages (one package per directory)
  - ***pkg*** contains package objects
  - ***bin*** contains executable commands
- Environment variables
  - **GOROOT**, **GOPATH**, **GOOS**, **GOARCH**...

Godoc

- The Go project takes documentation seriously. Documentation is a huge part of making software accessible and maintainable.
- The convention is simple: to document a type, variable, constant, function, or even a package, write a regular comment directly preceding its declaration, with no intervening blank line.

$ godoc [flag] package [name ...]

```
// Package ftd provides content for learning Go
package ftd

// Greet greets ftd's participant
func Greet() { ... }
```

Testing

- Package testing provides support for automated testing of Go packages. It is intended to be used in concert with the “go test” command, which automates execution of any function of the form.

```
func TestXxx(*testing.T)
```

```
import "testing"

func TestAverage(t *testing.T) {
        var v float64
        v = Average([]float64{1,2})
        if v != 1.5 {
                t.Error("Expected 1.5, got ", v)
        }
}
```

Benchmark

- The benchmark function must run the target code b.N times. The benchmark package will vary b.N until the benchmark function lasts long enough to be timed reliably.

```
func BenchmarkXxx(*testing.B)
```

```
func BenchmarkHello(b *testing.B) {
        for i := 0; i < b.N; i++ {
                fmt.Sprintf("hello")
        }
}
```

Talk is cheap. Show me the ‘Go’ code!

```
$ go run hello.go // returns Hello World

package main

import "fmt"

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

Packages

- Package main;
- Unique name for paths, not name;
- init() function
- Packages are not per file

```
package ftd
```

```
package ftd/hello
```

Imports

- An import declaration states that the source file containing the declaration depends on functionality of the imported package and enables access to exported identifiers of that package.

Import declaration	Local name of Greet
`import "ftd/hello"`	`ftd.Greet`
`import gr "ftd/hello"`	`gr.Greet`
`import . "ftd/hello"`	`Greet`
`import _ "ftd/hello"`	`initialization`
`import "github.com/user/project"`	Remote import paths

Types

- A type determines the set of values and operations specific to values of that type. Types may be named or unnamed.

Predeclared identifiers:

Types

bool byte complex64 complex128 error float32 float64 int int8 int16 int32 int64 rune string uint uint8 uint16 uint32 uint64 uintptr

Constants

true false iota

Zero value

nil

Functions

append cap close complex copy delete imag len make new panic print println real recover

Type Conversions

- Basically, you can convert between a named typed and its underlying type.

```
type Mystring string
var myStr Mystring = Mystring("awesome")
var str string = string(myStr)
```

- There are few rules to keep in mind when it comes to type conversions.

Variables

- Declaration one or more variables
- You can declare multiple variables at once.
- Infer the type of initialized variables.
- Zero-valued initialization
- Go supports constants of character, string, boolean, and numeric values (arbitrary precision)

```
var number = 12
var event string = "ftd"
isFtd := true // syntax sugar with type inference
const n = 500000000
```

Iota

- Go's iota identifier is used in in const declarations to simplify definitions of incrementing numbers. Because it can be used in expressions, it provides a generality beyond that of simple enumerations.

```
const (
    Sunday = iota     // 0
    Monday            // 1
    Tuesday           // 2
    Wednesday         // 3
    Thursday          // 4
    Friday            // 5
    Saturday          // 6
)
```

Arrays

- An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length and is never negative. (Arrays are values)

```
[32]byte
[2] struct { x, y int32 }
[1000]*float64
array := [...]float64{7.0, 8.5, 9.1}
names := [2]string{"FTD", "Golang"}
```

Slices

- Slices are a key data type in Go, giving a more powerful interface to sequences than arrays.

```
s := make([]string, 3)
s[0] = "a"
s[1] = "b"
s[2] = "c"

s1 := s[0:2]  // [a b]

s2 := append(s1, "d", "e", "f") // [a b d e f]
```

Maps

- Maps are Go's built-in associative data type (sometimes called hashes or dicts in other languages).

```
m := make(map[string]int)
m["one"]  = 1
m["nine"] = 9

delete(m, "one")

n := map[string]int{"foo": 1, "bar": 2}
```

Range

- range iterates over of elements in a variety of data structures.

```
nums := []int{2, 3, 4}

for idx, num := range nums {
        fmt.Println("Index:", idx, "Value:", num)
}

kvs := map[string]string{"a": "apple", "b": "banana"}
for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
}

for i, c := range "go" {
        fmt.Println(i, c)
}
```

Control structures

- If/Else statement without parentheses and initialization statement
- There is no ternary if in Go
- For only in three forms
- For range clause (array, slice, string, or map, or channel read)
- Switch statements express conditionals across many branches.

Functions

- A function declaration binds an identifier, the function name, to a function.
- Multiple return values
- Named result parameters

```
func plus(a int, b int) int {
    return a + b
}
```

Multiple return values

- One of Go's unusual features is that functions and methods can return multiple values.

```
func (file *File) Write(b []byte) (n int, err error)
```

Named result parameters

- When named, they are initialized to the zero values for their types when the function begins; if the function executes a return statement with no arguments.

```
func Connect(host string) (result int, err error) {
    if host == nil {
        result = 0
        error = errors.New("No internet connection")
        return
    }
    ...
}
```

Variadic Functions

- Variadic functions can be called with any number of trailing arguments. (aka varargs).

```
func sum(nums ...int) {
        total  := 0
        for _, num := range nums {
                total += num
        }
        fmt.Println("Result:", total)
}

nums := []int{2,4,6,8,10}
sum(s...) // Result: 30
```

Logging

- Package log implements a simple logging package. It defines a type, Logger, with methods for formatting output.

```
import "log"

func Fatal(v ...interface{})
func Println(v ...interface{})
func Printf(format string, v ...interface{})

func New(out io.Writer, prefix string, flag int) *Logger
```

Closures

- Go supports anonymous functions, which can form closures. Anonymous functions are useful when you want to define a function inline without having to name it.

```
plus := func(a, b int) int {
        return a + b
}

fmt.Println(plus(2, 2))
```

Pointers

- Go supports pointers, allowing you to pass references to values and records within your program.
- No pointer arithmetic

```
i := 1
var ip *int = &i
fmt.Println("Value:", i)
fmt.Println("Pointer:", &i)
fmt.Println("Pointer:", ip)
```

Structs

- Go's structs are typed collections of fields. They're useful for grouping data together to form records.

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

Person{"Costinha", 68}
Person{Name:"Costinha", Age:68}

&Person{"Costinha", 68}    // struct pointer
&Person{Age:11}            // struct pointer
```

Allocation

- Go has two allocation primitives, the built-in functions:
  - **new(T)**: it returns a pointer to a newly allocated zero value of type T.
  - **make(T, len, cap)**: it creates slices, maps, and channels only, and it returns an initialized (notzeroed) value of type T (not *T)

```
type Person struct {
     Name string
}

p  := new(Person) // *Person
p.Name = "Golang"


nums := make([]int, 10) // slice []int length 10
```

Methods

- Go supports methods defined on struct types. Called receiver method.

```
type rect struct {
        width, height int
}

func (r *rect) area() int {
        return r.width * r.height
}

func (r rect) perim() int {
        return 2*r.width + 2*r.height
}
```

Interfaces

- Interfaces are named collections of method signatures.
- No implements keyword
- Duck typing support

```
type Animal interface {
        Name() string
}

type Dog struct{}

func (dog Dog) Name() string {
        return "AuAu"
}
```

Errors

- In Go it’s idiomatic to communicate errors via an explicit, separate return value.
- error interface must be implemented

```
func (e InternetError) Error() string {
      return "No internet connection"
}
```

Defer

- Defer is used to ensure that a function call is performed later in a program’s execution, usually for purposes of cleanup.
- Recover from the run-time panic
- Deferred functions are executed in LIFO order

```
defer func()
```

Files

- Package io provides basic interfaces to I/O primitives and package ioutil implements some I/O utility functions.

```
bs, err := ioutil.ReadFile("test.txt")
if err != nil {
    // handle the error here
}
fmt.Println(string(bs))
```

Files (cont.)

- Write a file

```
file, err := os.Create("test.txt")
if err != nil {
    // handle the error here return
}

defer file.Close()

file.WriteString("test")
```

Sorting

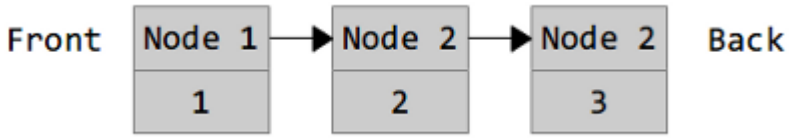
- Go’s sort package implements sorting for builtins and user-defined types. We’ll look at sorting for builtins first.
- Sorting by functions support

```
strs := []string{"f", "t", "d"}
sort.Strings(strs)
fmt.Println("Strings:", strs) // [d f t]

ints := []int{7, 2, 4}
sort.Ints(ints)
fmt.Println("Ints:", ints)    // [2 4 7]
```

Containers

- Go has several more collections available in the container package. (heap, list and ring).




```
import "container/list"

l := list.New()              // returns an initialized list.
e4 := l.PushBack(4)          // inserts element at the back of list and returns element
e1 := l.PushFront(1)         // inserts element at the front of list and returns element
l.InsertBefore(3, e4)        // inserts element before mark and returns element
l.InsertAfter(2, e1)         // inserts element after mark and returns element

for e := l.Front(); e != nil; e = e.Next() {
     fmt.Println(e.Value)
}
```

Goroutines

- A lightweight thread of execution.
- To invoke this function in a goroutine, use go f(s). This new goroutine will execute concurrently with the calling one.

- M:N threading model

- Environment variable **GOMAXPROCS**, define runtime native threads

Goroutines (cont.)

- Examples

```
go f("Goroutine")

go func(msg string) {
      fmt.Println(msg)
}("Goroutine")
```

Channels

- A channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type. The value of an uninitialized channel is nil.

```
messages := make(chan string) //unbuffered chan
go func() { messages <- "ping" }()

msg := <-messages
fmt.Println(msg)
```

Channels (cont.)

- By default channels are unbuffered, meaning that they will only accept sends (chan <-) if there is a corresponding receive (<- chan) ready to receive the sent value. Buffered channels accept a limited number of values without a corresponding receiver for those values.

```
messages := make(chan string, 2) //buffered chan

messages <- "ftd"
messages <- "Golang"

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

Handling panics

- Two built-in functions, panic and recover, assist in reporting and handling run-time panics and program-defined error conditions.

```
func panic(interface{})
```

```
panic(42)
panic("unreachable")
panic(Error("cannot parse")) // runtime.Error
```

```
func recover() interface{}
```

```
defer func() {
        if x := recover(); x != nil {
                log.Printf("run time panic: %v", x)
        }
}()
```

JSON

- Go offers built-in support for JSON encoding and decoding, including to and from built-in and custom data types.

```
package "ecoding/json"

func Marshal(interface{}) ([]byte, error)
func Unmarshal(data []byte, v interface{}) error
```

References

- The Go Programming Language Specification
  http://golang.org/ref/spec

- Effective Go
  http://golang.org/doc/effective_go.html

- Go by example
  https://gobyexample.com/

Get involved!

- Go Nuts
  https://groups.google.com/forum/#!forum/golang-nuts
- GopherCon
  http://gophercon.com/
- Go user groups
  https://code.google.com/p/go-wiki/wiki/GoUserGroups


THANK YOU!
FTD GROUP