

Introduction to Go

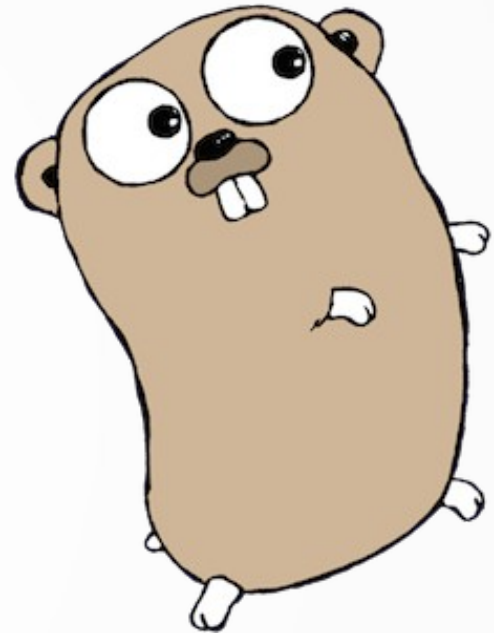
**Programming Models for Emerging
Platforms**

THE
C
PROGRAMMING
LANGUAGE

+

Memory
Safety

=

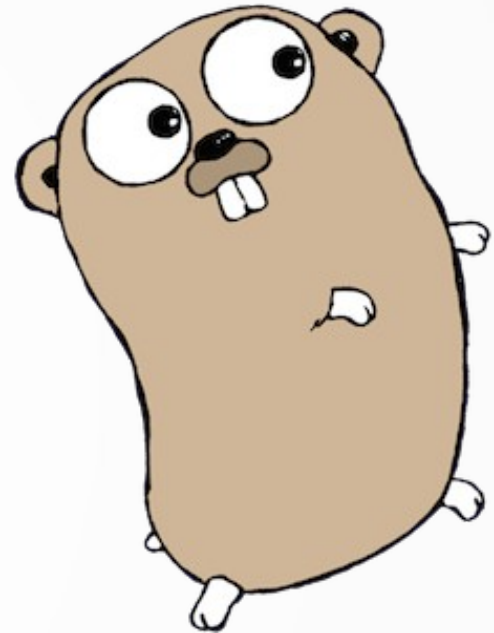


THE
C
PROGRAMMING
LANGUAGE

+

**Memory
Safety** =

(And some other
nice features)



Go was born out of frustration with existing languages...
...One had to choose either efficient compilation,
efficient execution, or ease of programming...[choosing]
ease over safety and efficiency by moving to
dynamically typed languages such as Python and
JavaScript rather than C++ or, to a lesser extent, Java.

Very High Level Go

- Go
 - Compiles quickly
 - Type safe
 - Minimal

We were not alone in our concerns. After many years with a pretty quiet landscape for programming languages, Go was among the first of several new languages—Rust, Elixir, Swift, and more—that have made programming language development an active, almost mainstream field again.

We were not alone in our concerns. After many years with a pretty quiet landscape for programming languages, Go was among the first of several new languages—Rust, Elixir, Swift, and more—that have made programming language development an active, almost mainstream field again.

Not true at all!

Don't Be Fooled

- Go is not:
 - A revolution in language design
 - A bunch of new ideas
 - Any new ideas

Don't Be Fooled

- Go is:
 - Well designed, in the sense that language features are orthogonal
 - Very clearly selected syntax for productivity
 - Strongly typed (we just say type safe)
 - Memory safe (garbage collection)
 - Built in notion of concurrency

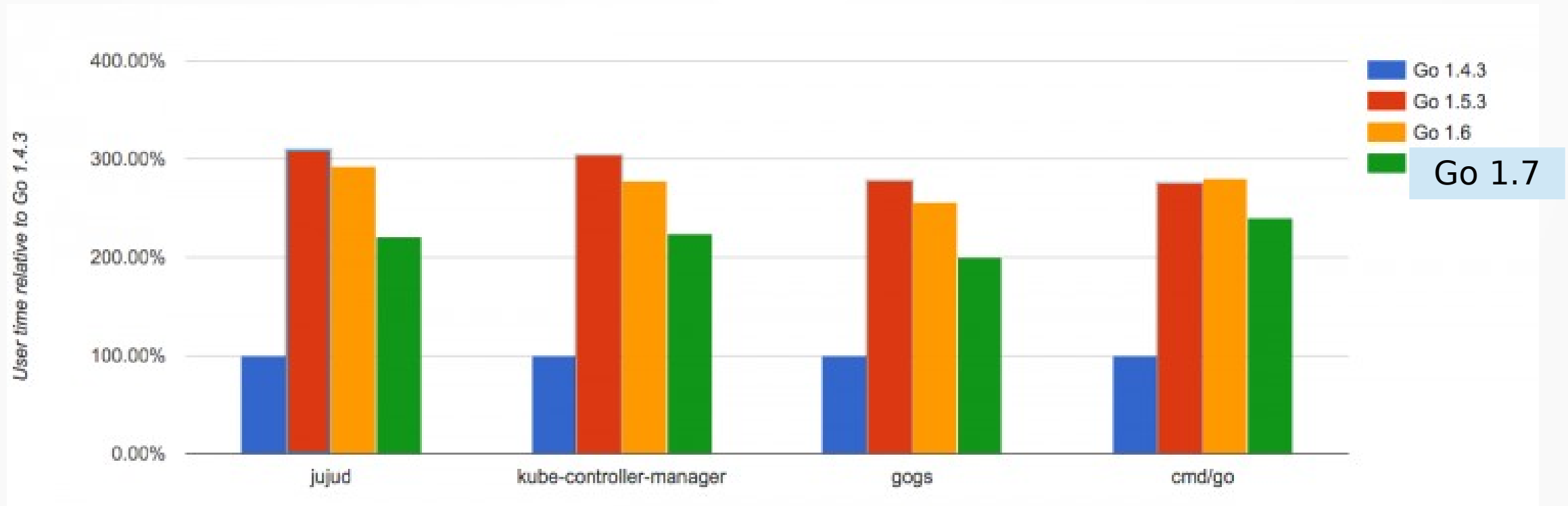
Why I like Go...

- Most programmers still favor imperative programming
- If we are to use an outdated model, we should at least use it well
- Go is basically C with some nice features
- It also has a very nice software ecosystem that integrates with github

Quick Anecdote

- Go
 - **Compiles quickly**
 - Type safe
 - Minimal

Quick Anecdote



Go Features

- Useful syntax
- Arrays and Maps (light-weight generics)
- Slices
- Packages (and package programming)
- Interfaces
- Channels
- Go Routines

Go Features

- Example (list.go)
 - Basic syntax, struct, interface

Go Interface

- We did not explicitly **extend** or **implement** the interface
- Pros
 - All we had to do was create the interface, both `IntLinkedList` and `IntArrayList` “magically” fit
 - Requires less changes to existing code (big plus)
- Cons
 - Hard to know who implements what
 - “Accidentally” fit interfaces

Go Interface

- Structural Subtyping (Go)
 - Feels like “duck typing” in dynamic languages
- Nominal Subtyping (Java)

Go Interface

- Example (list.go)
 - Stringer interface, Sort interface

Go Pointers

- Semantics the same as C
 - Always pass by value
 - If you want to mutate, need a pointer
- However
 - Memory can escape the stack into the heap in go (you can safely return &val)
 - Garbage collected

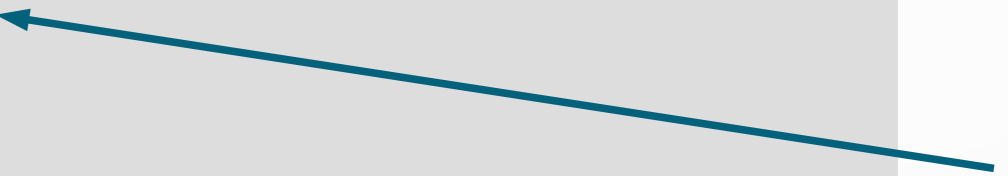
Go Pointers

```
func NewFile(fd int, name string) *File {  
    if fd < 0 {  
        return nil  
    }  
    f := File{fd, name, nil, 0}  
    return &f  
}
```

Composite Literal (creates File)



Return *File ("escapes")



Would crash in C. Proper idiom in Go (escape analysis makes this possible)

Go Arrays and Slices

- Arrays
 - `var arr [3]int`
 - `var arr2 [3]int`
 - Arrays are values in Go (copy semantics)
 - “`arr2 = arr`” will copy arr into arr2
 - “`foo(arr)`” will copy arr to foo
 - `[3]int` and `[6]int` are distinct types

Go Arrays and Slices

```
type Slice struct {  
    arr *[size]T  
    len, cap int  
}
```

- Slices
 - `var sl []int`
 - Contains a pointer to some concrete array
 - Passing a slice copies the slice, but since `arr` is pointer, does not lead to an array copy
 - **append** builtin handles reallocation of underlying array
 - Bounds check performed on **len**, not **cap**

Go Arrays and Slices

```
func grow(s1 []int, n int) {  
    for i := 0; i < n; i++ {  
        s1 = append(s1, n-i-1)  
    }  
}  
  
func main() {  
    s1 := make([]int, 0, 10)  
    grow(s1, 10)  
    for _, v := range s1 {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

What is the output?

Go Arrays and Slices

```
func grow(sl []int, n int) {  
    for i := 0; i < n; i++ {  
        sl = append(sl, n-i-1)  
    }  
}  
  
func main() {  
    sl := make([]int, 0, 10)  
    grow(sl, 10)  
    for _, v := range sl {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

Nothing. Pass by value.

Go Arrays and Slices

```
func grow(sl []*int, n int) {  
    for i := 0; i < n; i++ {  
        *sl = append(*sl, n-i-1)  
    }  
}
```

```
func main() {  
    sl := make([]*int, 0, 10)  
    grow(&sl, 10)  
    for _, v := range sl {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

Pass a pointer to the slice to append

May seem confusing, but the rule is always the same. Pass by value.

Go Arrays and Slices

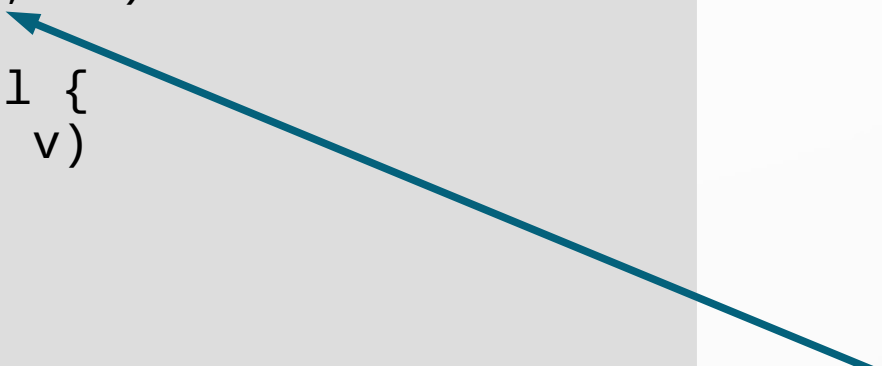
```
func grow(sl []int, n int) {  
    for i := 0; i < n; i++ {  
        sl[i] = n-i-1  
    }  
}  
  
func main() {  
    sl := make([]int, 0, 10)  
    grow(sl, 5)  
    for _, v := range sl {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

What about now?

Go Arrays and Slices

```
func grow(sl []int, n int) {  
    for i := 0; i < n; i++ {  
        sl[i] = n-i-1  
    }  
}  
  
func main() {  
    sl := make([]int, 0, 10)  
    grow(sl, 5)  
    for _, v := range sl {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

What about now?



Will crash because of
len = 0

Go Arrays and Slices

```
func grow(sl []int, n int) {
    for i := 0; i < n; i++ {
        sl[i] = n-i-1
    }
}

func main() {
    sl := make([]int, 10, 10)
    grow(sl, 5)
    for _, v := range sl {
        fmt.Printf("%d ", v)
    }
    fmt.Printf("\n")
}
```

4 3 2 1 0 0 0 0 0 0

Go Arrays and Slices

```
func grow(s1 []int, n int) {  
    for i := 0; i < n; i++ {  
        s1[i] = n-i-1  
    }  
}
```

```
func main() {  
    s1 := make([]int, 10, 10)  
    grow(s1, 5)  
    s12 := s1[0:4]  
    for _, v := range s12 {  
        fmt.Printf("%d ", v)  
    }  
    fmt.Printf("\n")  
}
```

Take a “slice” from a slice

4 3 2 1

Slicing shares the underlying array
(constant creation time)

Go Maps

- `var offered map[string]bool`
- `offered := make(map[string]bool)`
- Mostly similar to maps in other languages
- Couple of differences
 - zero value for missing entrys
 - comma, ok idiom

Go Maps (Zero Value)

```
func main() {  
    offered := make(map[string]bool)  
    offered["CS476"] = true  
    if offered["CS476"] {  
        fmt.Printf("CS476 offered!\n")  
    }  
    if offered["CS471"] {  
        fmt.Printf("CS471 offered!\n")  
    }  
}
```

Entries not in map will default to the zero value for the output type

Using map[string]bool creates a makeshift "set"

Go Maps (Comma Ok Idiom)

```
func main() {  
    nstudents := make(map[string]int)  
    nstudents["CS476"] = 12  
    if n, ok := nstudents["CS476"]; ok {  
        fmt.Printf("CS476 has %d students!\n", n)  
    }  
    if n, ok := nstudents["CS471"]; ok {  
        fmt.Printf("CS471 has %d students!\n", n)  
    }  
}
```

Go allows variable declaration in if statement. (Useful syntax)

True if entry exists, then execute body of if statement with **n**

Entry doesn't exist, never executes

Go Maps (Comma Ok Idiom)

```
func main() {  
    nstudents := make(map[string]int)  
    nstudents["CS476"] = 12  
    if n, ok := nstudents["CS476"]; ok {  
        fmt.Printf("CS476 has %d students!\n", n)  
    }  
    if n, ok := nstudents["CS471"]; ok {  
        fmt.Printf("CS471 has %d students!\n", n)  
    }  
}
```

Go allows variable declaration in if statement. (Useful syntax)

True if entry exists, then execute body of if statement with **n**

Zero value wont work in this case (would return 0)

Entry doesn't exist, never executes

Go's Lack of Generics

- Go does not have support for Generics
 - slices and maps are language builtins that can hold any data type, but that's it
- Go has slices, maps, and interfaces
- Is this enough?

Go's Lack of Generics

- Generics are formally referred to as *parametric polymorphism*
- Parametric polymorphism greatly increases complexity of implementing compiler
- I agree with the Go designers in this case:
 - Trade-off between complexity and expressiveness
 - Go is at least consistent in its value for simplicity
 - I feel that its not until we reach *higher-order parametric polymorphism* that the expressiveness outweighs the complexity

Go Package System

- Really just a convenient way to organize code (a module)
- Considering the Go designers have recently added modules, I'd say they got this wrong
- But still a step above C/Java

Go Package System

- Example (course.golang.org/)

Go Package System

- init allows for “safe” package-level programming
 - go will enforce initialization order
- Uppercase name will be exported
- Packages conveniently integrate with github
 - go get github.com/path-to-package
 - go get github.com/hawx/img

Acknowledgments

- https://golang.org/doc/effective_go.html
- <https://golang.org/ref/spec>
- <https://dave.cheney.net/tag/performance>
-