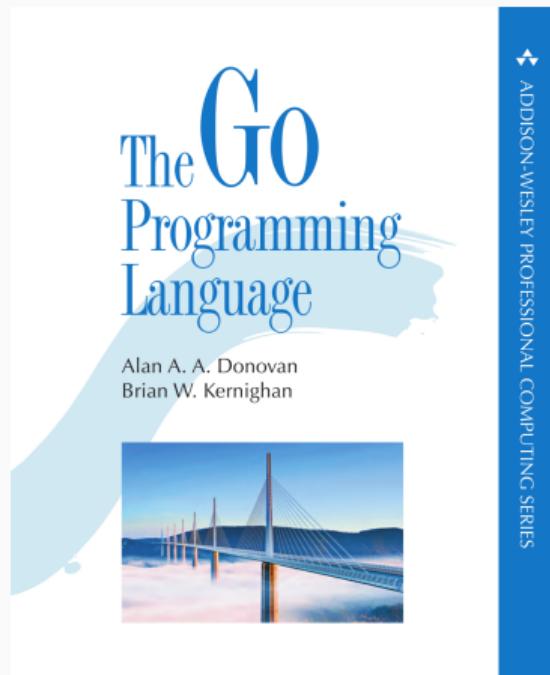


Programming in Go

Matt Holiday
Christmas 2020



The Book



ISBN 978-0-13-419044-0

“Anything with Brian Kernighan’s name on it is worth reading.”

— Matt Holiday

Why Go?

A better C from the folks who created Unix & C

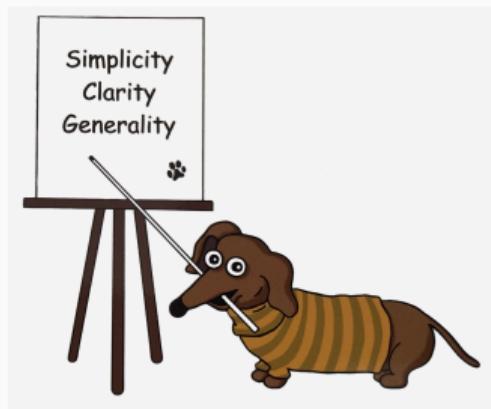
“Go is about language design in the service of software engineering.” — Rob Pike

from *Software Engineering at Google*

It's programming if “clever” is a compliment,
but it's software engineering if “clever” is an accusation.



Simplicity



The Practice of Programming

“Programs must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson

“If our basic tool, the language in which we design and code our programs, is also complicated, **the language itself becomes part of the problem rather than part of its solution.**” — Tony Hoare

Design goals

“Go is an attempt to combine the **ease of programming** of an interpreted, dynamically typed language with the **efficiency and safety** of a statically typed, compiled language.” — *Go FAQ*

Overall goals:

- **simplicity, safety, and readability**
- ease of expressing algorithms
- orthogonality
- one right way to do things

A language that fits in your head

“One of the reasons I enjoy working with Go is that I can mostly hold the spec in my head — and when I do misremember parts it’s a few seconds’ work to correct myself. It’s quite possibly the only non-trivial language I’ve worked with where this is the case.”

— Eleanor McHugh

“Clear is better than clever” — Go Proverb

Go is boring



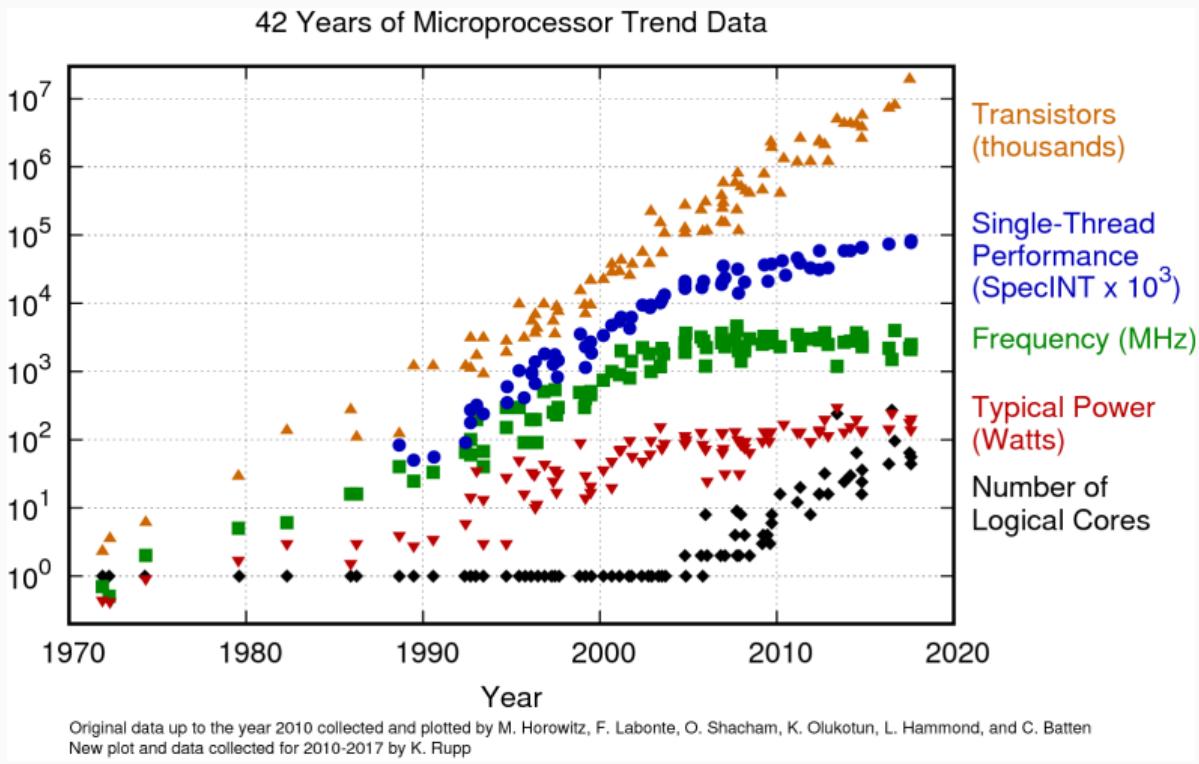
◀ SOFTWARE ENGINEERING

Go is Boring...And That's Fantastic!

A deep dive into why the world depends on simple, reliable, well-understood technologies

<https://www.capitalone.com/tech/software-engineering/go-is-boring>

Why did they do it?



Why did they do it?

A new language for a new computing landscape:

- multicore processors
- networked systems
- massive clusters
- the web programming model (think REST)
- huge programs
- large numbers of developers
- long build times

##	Language	Year
1	JavaScript	1995
2	Python	1991
3	Java	1995
4	C#	2000
5	C++	1983
6	C	1972

Concurrency was an afterthought in older languages ☹

It's just a computer you rent

 Iron

Home Products ▾ Blog ▾ Resources ▾

How We Went from 30 Servers to 2: Go

By Dylan Stamat | March 12, 2013 | 74 

When we built the first version of [IronWorker](#), about 3 years ago, it was written in Ruby and the API was built on Rails. It didn't take long for us to start getting some pretty heavy load and we quickly reached the limits of our [Ruby](#) setup. Long story short, we switched to [Go](#). For the long story, keep reading, here's how things went down.

The Original Setup



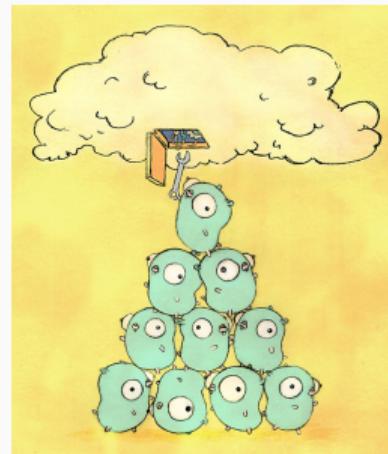
A cartoon illustration featuring Iron Man on the left and Superman on the right. They are standing in front of a server rack with a large yellow and black circular sign above it that has the Iron logo on it. The background is split into green on the left and orange on the right.

<https://blog.iron.io/how-we-went-from-30-servers-to-2-go>

Built for the cloud

It's taking over the infrastructure/container/cloud world:

- Docker
- Kubernetes
- Helm
- Drone
- Rancher
- Prometheus
- Grafana
- CoreOS (etcd, flannel)
- CockroachDB
- DropBox
- CloudFlare



©Renée French

"Think about it like this . . . if you can write something in Go just as [quickly] as you could in Python and:

- gain the speed and robustness of a compiled, statically-typed language without *all* of the rope to hang yourself
- clearly express concurrent solutions to parallelizable problems
- sacrifice little to nothing in terms of functionality
- unambiguously produce consistently styled code (thank you `gofmt`)

What would you choose?"

<https://word.bitly.com/post/29550171827/go-go-gadget>

Once more, with feeling

“A language that doesn’t have everything is actually easier to program in than some that do.” — Dennis Ritchie

Programming in Go

Matt Holiday
Christmas 2020



Hello, world!

Hello, playground!

Simple programs run at <https://play.golang.org>



The screenshot shows a web browser window for "The Go Playground" at play.golang.org. The window has a title bar with standard OS X icons and a URL field showing the site address. Below the title bar is a navigation bar with buttons for "Run", "Format", "Imports", "Share", and "About". The main area contains a code editor with the following Go code:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground!")
9 }
10
11
12
13
14
15
```

Below the code editor is a terminal window displaying the output of the program:

```
Hello, playground!
Program exited.
```

Playground limitations

There's no input or output *except* writing to

- `stdout`
- `stderr`

For obvious reasons, you can't write to files or open a network socket or run a web server

More information

Get all the info at <https://golang.org/doc/>

The screenshot shows a web browser window displaying the official Go documentation at <https://golang.org/doc/>. The page has a clean, modern design with a light blue header bar. The main navigation menu includes links for "Documents", "Packages", "The Project", "Help", "Blog", "Play", "Search", and a magnifying glass icon. Below the menu, the title "The Go Programming Language" is displayed, followed by a brief introduction to the language's features: expressiveness, conciseness, efficiency, concurrency, and modular construction. The page is organized into several sections: "Installing Go", "Getting Started", "Learning Go", and "A Tour of Go". Each section contains descriptive text and links to further resources. A small, friendly cartoon owl icon is positioned in the bottom right corner of the page area.

The Go programming language is an open source project to make programmers more productive.

Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

Installing Go

Instructions for downloading and installing the Go compilers, tools, and libraries.

Getting Started

Learning Go

A Tour of Go

An interactive introduction to Go in three sections. The first section covers basic syntax and data structures; the second discusses methods and interfaces; and the third introduces Go's concurrency primitives. Each section concludes with a few exercises so you can practice what you've learned. You can [take the tour online](#) or install it locally with:

Hello, world!

What the simplest program looks like:

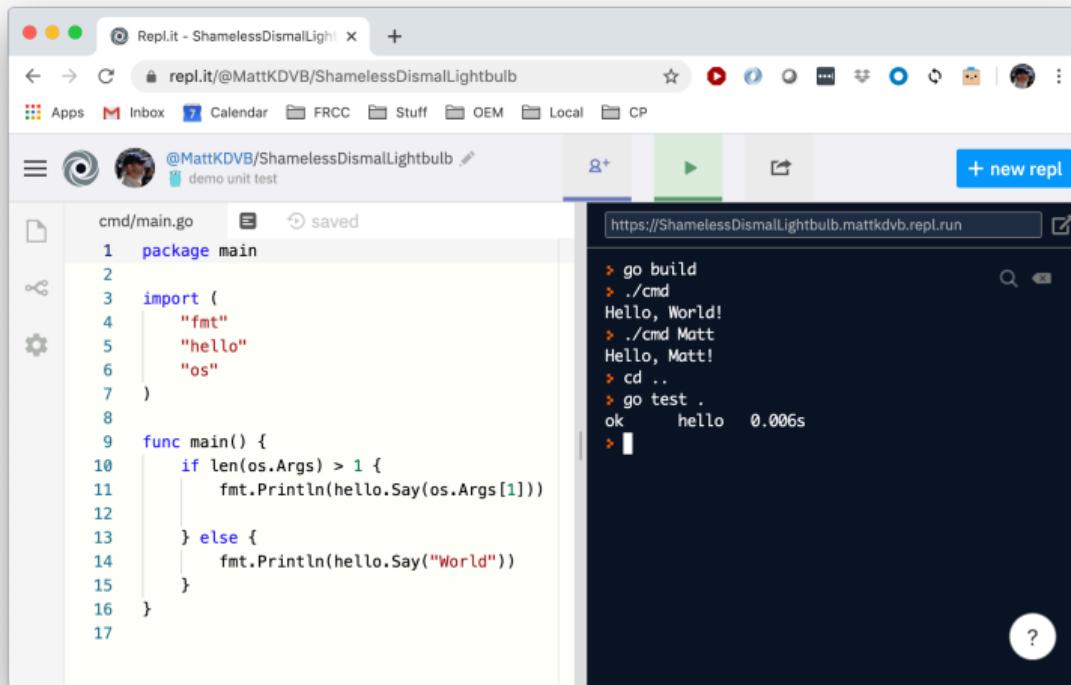
```
package main

import "fmt"

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

Hello, repl.it!

You can also use [repl.it](#)



The screenshot shows the repl.it web interface with a Mac-style window title bar. The main area has a toolbar with icons for file operations, a search bar, and a user profile. Below the toolbar is a navigation bar with links like Apps, Inbox, Calendar, etc. The central workspace consists of two panes: a code editor on the left and a terminal window on the right.

Code Editor:

```
cmd/main.go
1 package main
2
3 import (
4     "fmt"
5     "hello"
6     "os"
7 )
8
9 func main() {
10    if len(os.Args) > 1 {
11        fmt.Println(hello.Say(os.Args[1]))
12    } else {
13        fmt.Println(hello.Say("World"))
14    }
15
16 }
17
```

Terminal Window:

```
https://ShamelessDismalLightbulb.mattkdvb.repl.run
> go build
> ./cmd
Hello, World!
> ./cmd Matt
Hello, Matt!
> cd ..
> go test .
ok    hello  0.006s
> |
```

Installation

Start from the Go downloads page: <https://golang.org/dl>

Mac: use `brew install go` (or use the installer package)

Homebrew installation: <https://brew.sh>

Windows: open the installer (MSI) file and follow the prompts to install the Go tools
(otherwise you can download a ZIP file, but you have to set some environment stuff)

Linux: download the archive and extract it into `/usr/local`, creating a Go tree in `/usr/local/go`

```
$ sudo tar -C /usr/local -xzf go1.15.6.linux-amd64.tar.gz
```

and don't forget to add `/usr/local/go/bin` to `$PATH` (Linux)

Running a program

From the command line:

```
$ go run .
Hello, world!
```

Later we'll talk about how to build binaries that stick around

Programming in Go

Matt Holiday
Christmas 2020



A Simple Example

Read a name from the command line

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Printf("Hello, %s!\n", os.Args[1])
}
```

Running a program with a bug

From the command line:

```
$ go run . ## no name
panic: runtime error: index out of range [1] with length 1

goroutine 1 [running]:
main.main()
    /Users/mholiday/go/src/hello/main.go:9 +0xce
exit status 2
```

What went wrong? We read past the end of `os.Args`!

Build a unit test

```
//file: cmd/main.go
package main

import (
    "fmt"
    "hello"
    "os"
)

func main() {
    if len(os.Args) > 1 {
        fmt.Println(hello.Say(os.Args[1]))
    } else {
        fmt.Println(hello.Say("world"))
    }
}
```

Build a unit test

```
//file: hello.go
package hello

import "fmt"

func Say(name string) string {
    return fmt.Sprintf("Hello, %s!", name)
}
```

Build a unit test

```
//file: hello_test.go
package hello

import "testing"

func TestSayHello(t *testing.T) {
    want := "Hello, test!"
    got := Say("test")

    if want != got {
        t.Errorf("wanted %s, got %s", want, got)
    }
}
```

Running a unit test

From the command line:

```
$ go test ./...
ok      hello  0.003s
?      hello/cmd [no test files]
```

Something a little fancier

```
//file: hello.go
package hello

import "strings"

func Say(names []string) string {
    if len(names) == 0 {
        names = []string{"world"}
    }

    return "Hello, " + strings.Join(names, ", ") + "!"
}
```

Something a little fancier

```
//file: hello_test.go
package hello

import "testing"

func TestSay(t *testing.T) {
    subtests := []struct {
        items  []string
        result string
    }{
        {
            result: "Hello, world!",
        },
        {
            items:  []string{"Matt"},
            result: "Hello, Matt!",
        },
        ...
    }
}
```

Something a little fancier

```
    . . .
    {
        items: []string{"Matt", "Cary", "Anne"},
        result: "Hello, Matt, Cary, Anne!",
    },
}

for _, st := range subtests {
    if s := Say(st.items); s != st.result {
        t.Errorf("got %s, gave %v, wanted %s", s,
                 st.items, st.result)
    }
}
```

Something a little fancier

```
//file: cmd/main.go
package main

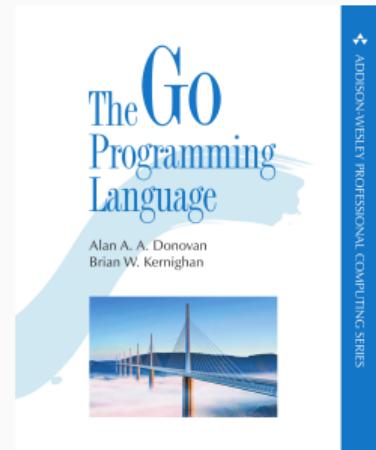
import (
    "fmt"
    "hello"
    "os"
)

func main() {
    fmt.Println(hello.Say(os.Args[1:]))
}
```

Significant changes since the book

Language:

1. struct conversion with mismatched tags (1.8)
2. type aliases (1.9)
3. **Go modules** (1.11+)
4. improved numeric literals and error handling (1.13)
5. overlapping interfaces (1.14)



Runtime:

1. performance improvements in every release
2. full (async) preemption of goroutines (1.14)

Go modules

Go modules are the one really big change from the *GOPL* book
(instead of using \$GOPATH)

You create a root directory with the module name, e.g., `hello`

Run `go mod init hello` to create the file `go.mod`:

```
module hello
```

```
go 1.14
```

This file will also end up with a list of 3rd-party dependencies
(later)

Programming in Go

Matt Holiday
Christmas 2020



Basic Types

Keywords & symbols

Only 25 keywords; you may not use these as names:

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

Plus a bunch of operators & symbols:

+	&	+=	&=	&&	==	!=	(
-	-	-=	=		<	<=	[
*	^	*=	^=	<-	>	>=	{
/	<<	/=	<<=	++	=	:=	,
%	>>	%=	>>=	--	--	:
	&^		&^=				

Predeclared identifiers

You can use these as names, shadowing the built-in meaning,
but you really don't want to do that!

Constants:

```
true false iota nil
```

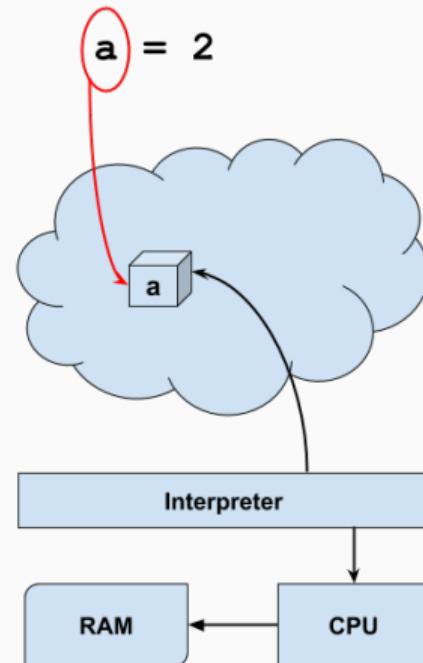
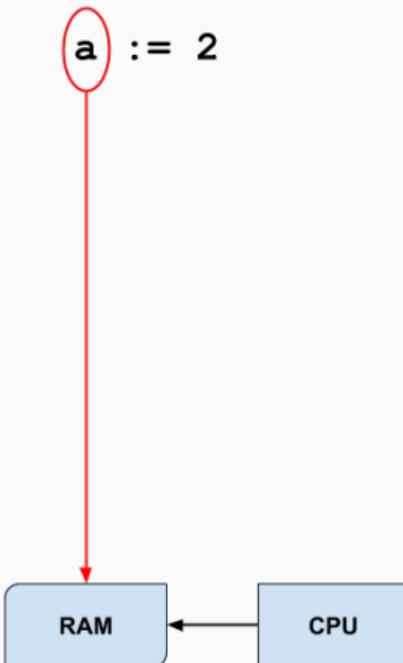
Types:

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

Functions:

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

Machine-native vs interpreted



Integers

“Unsized” integers default to the machine’s natural wordsize:

- 64 bits on my laptop
- 32 bits on my Raspberry Pi

`int` is the default type for integers in Go, even lengths

Signed	Unsigned
<code>int</code>	<code>uint</code>
<code>int64</code>	<code>uint64</code>
<code>int32</code>	<code>uint32</code>
<code>int16</code>	<code>uint16</code>
<code>int8</code>	<code>uint8</code>

“Real” and “imaginary” numbers

Non-integers are represented in floating point:

- floating point numbers:
`float32 float64`
- complex (imaginary) floating point numbers:
`complex64 complex128`

Don't use floating point for monetary calculations!

Try a package like [Go money](#)

Simple declarations

Anywhere:

```
var a int  
  
var (  
    b = 2  
    f = 2.01  
)
```

Only inside functions:

```
c := 2
```

Number conversions

Conversions may change the value

```
func main() {
    a := 2
    b := 2.01

    fmt.Printf("a: %-4v %[1]T\n", a)      // fmt.Printf("a: %-4d %[1]T\n", a)
    fmt.Printf("b: %-4v %[1]T\n", b)      // fmt.Printf("b: %-4.2f %[1]T\n", b)

    var size float32 = 1.9

    y := int(size)          // truncated to 1
    z := float32(y)         // still 1.0 from 1
}
```

Once the number's been rounded down, it stays that way

Simple types

Special types:

- `bool` (boolean) has two values `false`, `true`
these values are **not** convertible to/from integers!
- `error`: a special type with one function, `Error()`
an `error` may be nil or non-nil
- Pointers are physically addresses, logically opaque
a pointer may be nil or non-nil
no pointer manipulation except through package `unsafe`

Initialization

Go initializes all variables to “zero” by default if you don’t:

- All numerical types get 0 (float 0.0, complex 0i)
- `bool` gets `false`
- `string` gets "" (the empty string, length 0)
- Everything else gets `nil`:
 - pointers
 - slices
 - maps
 - channels
 - functions (function variables)
 - interfaces
- For aggregate types, all members get their “zero” values

Constants

Only numbers, strings, and booleans can be constants (immutable)

Constant can be a literal or a compile-time function of a constant

```
const (
    a = 1                      // int
    b = 2 * 1024                // 2048
    c = b << 3                 // 16384

    g uint8 = 0x07              // 7
    h uint8 = g & 0x03          // 3

    s = "a string"
    t = len(s)                  // 8
    u = s[2:]                   // SYNTAX ERROR
)
```

Examples

```
// x and y get the values passed in by the caller

func do(x, y int) int {
    const t = 21           // type int by default
    const z = false       // type bool from the value

    var i uint8 = 255     // explicit type uint8
    var j = 256            // type int by default
    var k int              // 0 by default

    var m                  // SYNTAX ERROR, no type/value
    var n = nil             // SYNTAX ERROR, no type

    v := 0                  // short declaration, int
    w := func() { . . . }   // short declaration, function

    return k
}
```

Examples

```
// explicit conversion is required for integer types

func do(x, y int) int {
    k := x + y                      // k int
    m := uint32(k)                  // int conversion
    n := 3 * m                       // still uint32

    k = m                           // TYPE MISMATCH
    k = int(m)                     // int conversion

    var i uint8 = 255

    j := i++                         // SYNTAX ERROR
    b := k = 0                        // SYNTAX ERROR

    return m                         // TYPE MISMATCH
    return int(m)                   // int conversion
}
```

Programming in Go

Matt Holiday
Christmas 2020



Strings

Strings

Types related to strings:

- **byte**: a synonym for `uint8`
- **rune**: a synonym for `int32` for characters
- **string**: an **immutable** sequence of “characters”
 - physically a sequence of bytes (UTF-8 encoding)
 - logically a sequence of (unicode) runes

Runes (characters) are enclosed in single quotes: 'a'

“Raw” strings use backtick quotes: `string with "quotes"`

They also don’t evaluate escape characters such as \n

String-related types

Let's see rune vs byte in a string:

```
package main
import "fmt"

func main() {
    s := "élite"
    fmt.Printf("%8T %[1]v\n", s)
    fmt.Printf("%8T %[1]v\n", []rune(s))
    fmt.Printf("%8T %[1]v\n", []byte(s))
}
```

é is one rune (character) but two bytes in UTF-8 encoding:

```
string élite
[]int32 [233 108 105 116 101]
[]uint8 [195 169 108 105 116 101]
```

String-related types, in Chinese

I can't do this in the slides:



The Go Playground interface is shown, featuring a code editor and a results panel.

Code (main.go):

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     s := "你好 世界"
9
10    fmt.Printf("%#T %[1]v\n", s)
11    fmt.Printf("%#T %[1]v\n", []rune(s))
12    fmt.Printf("%#T %[1]v\n", []byte(s))
13 }
14
15
16
```

Output:

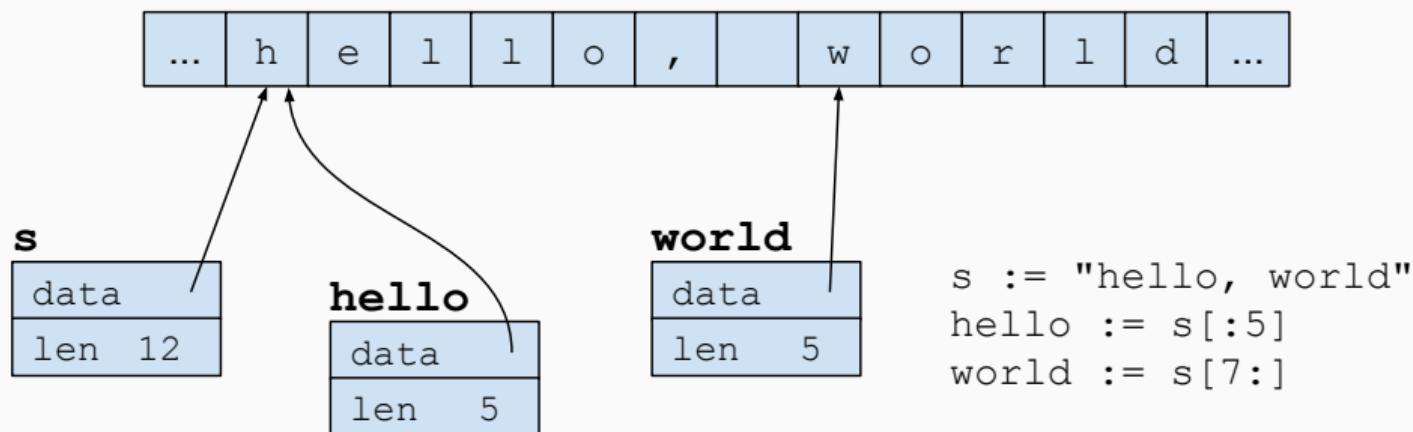
```
string 你好 世界
[]int32 [20320 22909 32 19990 30028]
[]uint8 [228 189 160 229 165 189 32 228 184 150 231 149 140]
```

Program exited.

String structure

The internal string representation is a pointer and a length

Strings are **immutable** and can share the underlying storage



Strings

Strings are a sequence of characters and are **immutable**

The built-in `len` function calculates the length

Strings overload the addition operator (+ and +=)

```
s := "the quick brown fox"

a := len(s)                      // 19
b := s[:3]                        // "the"
c := s[4:9]                       // "quick"
d := s[:4] + "slow" + s[9:]      // replaces "quick"

s[5] = 'a'                         // SYNTAX ERROR
s += "es"                          // now plural (copied)
```

Strings are passed *by reference*, thus they aren't copied

String functions

Package `strings` has many functions on strings

```
s := "a string"  
  
x := len(s)          // built-in, = 8  
  
strings.Contains(s, "g")    // returns true  
strings.Contains(s, "x")    // returns false  
  
strings.HasPrefix(s, "a")   // returns true  
strings.Index(s, "string") // returns 2  
  
s = strings.ToUpper(s)     // returns "A STRING"
```

Note that we assign the result of `ToUpper` back to `s`

Programming in Go

Matt Holiday
Christmas 2020



Composite Types

Composite types

string

a	r	e	a
---	---	---	---

[4]int

1	12	4	8
---	----	---	---

[]int

1	13	3	17
---	----	---	-----	-----	----

map[string]int

"to"	1
"from"	12
"into"	3
"above"	0

Arrays

Arrays are typed by size, which is *fixed* at compile time

```
// all these are equivalent
```

```
var a [3]int
var b [3]int{0, 0, 0}
var c [...]{0, 0, 0}    // sized by initializer
```



```
var d [3]int
d = b                      // elements copied
```



```
var m [...]int{1, 2, 3, 4}
```



```
c = m                      // TYPE MISMATCH
```

Arrays are passed *by value*, thus elements are copied

Slices

Slices have variable length, backed by some array; they are copied when they outgrow that array

```
var a []int          // nil, no storage
var b = []int{1, 2}    // initialized

a = append(a, 1)      // append to nil OK
b = append(b, 3)      // []int{1, 2, 3}

a = b                // overwrites a

d := make([]int, 5)   // []int{0, 0, 0, 0, 0}
e := a                // same storage (alias)

e[0] == b[0]           // true
```

Slices are passed *by reference*; no copying, updating OK

The off-by-one bug

Slices are indexed like [8:11]

(read as the starting element and *one past* the ending element, so this way we have $11 - 8 = 3$ elements in our slice)

For loops work the same way in most cases:

```
for i := 8; i < 11; i++ { // in math written [8, 11)
    . . .
}
```



Read it on Wikipedia [OB1](#)

Slices

```
package main
import "fmt"

func main() {
    t := []byte("string")    // 0:s 1:t 2:r 3:i 4:n 5:g
    fmt.Println(len(t), t)   // 6 bytes in t
    fmt.Println(t[2])        // 1 item
    fmt.Println(t[:2])       // 2 items
    fmt.Println(t[2:])       // 6-2 items
    fmt.Println(t[3:5])      // 5-3 items
}
```

```
6 [115 116 114 105 110 103]
114
[115 116]
[114 105 110 103]
[105 110]
```

Slices vs arrays

Most Go APIs take slices as inputs, not arrays

Slice	Array
Variable length	Length fixed at compile time
Passed by reference	Passed by value (copied)
Not comparable	Comparable (==)
Cannot be used as map key	Can be used as map key
Has copy & append helpers	—
Useful as function parameters	Useful as “pseudo” constants

Arrays as pseudo-constants

It can be useful to have fixed-size tables of values in some algorithms, treated as constant data

```
// from the file crypto/des/const.go in the DES package

// Used to perform an initial permutation of a 64-bit
// input block.
var initialPermutation = [64]byte{
    6, 14, 22, 30, 38, 46, 54, 62,
    4, 12, 20, 28, 36, 44, 52, 60,
    2, 10, 18, 26, 34, 42, 50, 58,
    0, 8, 16, 24, 32, 40, 48, 56,
    7, 15, 23, 31, 39, 47, 55, 63,
    5, 13, 21, 29, 37, 45, 53, 61,
    3, 11, 19, 27, 35, 43, 51, 59,
    1, 9, 17, 25, 33, 41, 49, 57,
}
```

Examples

```
var w = [...]int{1, 2, 3}    // array of len(3)
var x = []int{0, 0, 0}        // slice of len(3)

func do(a [3]int, b []int) []int {
    a = b                      // SYNTAX ERROR
    a[0] = 4                    // w unchanged
    b[0] = 3                    // x changed

    c := make([]int, 5)         // []int{0, 0, 0, 0, 0}
    c[4] = 42
    copy(c, b)                 // copies only 3 elts

    return c
}

y := do(w, x)
fmt.Println(w, x, y)          // [1 2 3] [3 0 0] [3 0 0 0 42]
```

Maps

Maps are dictionaries: indexed by key, returning a value

You can read from a nil map, but inserting will panic

```
var m map[string]int      // nil, no storage
p := make(map[string]int) // non-nil but empty

a := p["the"]            // returns 0
b := m["the"]            // same thing
m["and"] = 1              // PANIC - nil map
m = p
m["and"]++               // OK, same map as p now
c := p["and"]            // returns 1
```

Maps are passed *by reference*; no copying, updating OK

The type used for the key must have == and != defined (*not slices, maps, or funcs*)

Maps

Maps can't be compared to one another; maps can be compared only to `nil` as a special case

```
var m = map[string]int{
    "and": 1,
    "the": 1,
    "or":  2,
}

var n map[string]int

b := m == n                      // SYNTAX ERROR
c := n == nil                      // true
d := len(m)                        // 3
e := cap(m)                        // TYPE MISMATCH
```

Maps

Maps have a special two-result lookup function

The second variable tells you if they key was there

```
p := map[string]int{}           // non-nil but empty

a := p["the"]                   // returns 0
b, ok := p["and"]               // 0, false

p["the"]++

c, ok := p["the"]               // 1, true

if w, ok := p["the"]; ok {
    // we know w is not the default value
    . .
}
```

Built-ins

Each type has certain built-in functions

<code>len(s)</code>	<code>string</code>	string length
<code>len(a), cap(a)</code>	<code>array</code>	array length, capacity (constant)
<code>make(T, x)</code>	<code>slice</code>	slice of type T with length x and capacity x
<code>make(T, x, y)</code>	<code>slice</code>	slice of type T with length x and capacity y
<code>copy(c, d)</code>	<code>slice</code>	copy from d to c; # = min of the two lengths
<code>c=append(c, d)</code>	<code>slice</code>	append d to c and return a new slice result
<code>len(s), cap(s)</code>	<code>slice</code>	slice length and capacity
<code>make(T)</code>	<code>map</code>	map of type T
<code>make(T, x)</code>	<code>map</code>	map of type T with space hint for x elements
<code>delete(m, k)</code>	<code>map</code>	delete key k (if present, else no change)
<code>len(m)</code>	<code>map</code>	map length

Make nil useful

Nil is a type of zero: it indicates the absence of something

Many built-ins are safe: `len`, `cap`, `range`

```
var s []int
var m map[string]int

l := len(s)                      // length of nil slice is 0

i, ok := m["int"]                 // 0, false for any missing key

for _, v := range s {             // skip if s is nil or empty
    ...
}
```

“Make the zero value useful.” — Rob Pike

“Understanding nil”

See Francesc Campoy’s video at

<https://www.youtube.com/watch?v=ynoY2xz-F8s>



GopherCon 2016: Francesc Campoy - Understanding nil

“Understanding nil”

♥ Dimitri Fontaine liked

 Programming Wisdom @CodeWisdom · 13h

“A language that doesn't affect the way you think about programming is not worth knowing.” - Alan J. Perlis

7 167 627

Programming in Go

Matt Holiday
Christmas 2020



Control Structures

Sequence

The simplest type of program has no “control structures”

It just flows from top to bottom (sequential execution)

```
package main
import (
    "fmt"
    "math"
)
func main() {
    a, b, c := -0.5, 0.5, 5.0
    x := math.Sqrt(b*b - 4*a*c) / (2 * a)
    y1, y2 := -b + x, -b - x

    fmt.Printf("%5.4f, %5.4f\n", y1, y2) // -3.7016, 2.7016
}
```

If-then-else

The next type of structure is a choice between alternatives

All if-then statements require braces

```
if a == b {  
    fmt.Println("a equals b")  
} else {  
    fmt.Println("a is not equal to b")  
}
```

They can start with a short declaration or statement

```
if err := doSomething(); err != nil {  
    return err  
}
```

For loops

The loop control structure provides automatic repetition

There is only **for** (no **do** or **while**) but with options

1. Explicit control with an index variable

```
for i := 0; i < 10; i++ {  
    fmt.Printf("(%d, %d)\n", i, i*i)  
}  
  
// prints (0, 0) up to (9, 81)
```

Three parts, all optional (initialize, check, increment)

The loop ends when the explicit check fails (e.g., $i == 10$)

For loops

2a. Implicit control through the `range` operator for arrays & slices

// one var: i is an index 0, 1, 2, ...

```
for i := range myArray {  
    fmt.Println(i, myArray[i])  
}
```

// two vars: i is the index, v is a value

```
for i, v := range myArray {  
    fmt.Println(i, v)  
}
```

The loop ends when the range is exhausted

For loops

2b. Implicit control through the range operator for maps

```
// one var: k is key  
  
for k := range myMap {  
    fmt.Println(k, myMap[k])  
}  
  
// two vars: k is the key, v is a value  
  
for k, v := range myMap {  
    fmt.Println(k, v)  
}
```

The loop ends when the range is exhausted

For loops

3. An infinite loop with an explicit break

```
i, j := 0, 3  
// this loop must be made to stop  
  
for {  
    i, j = i + 50, j * j  
  
    fmt.Println(i, j)  
  
    if j > i {  
        break          // when i = 150, j = 6561  
    }  
}
```

There is also `continue` to make an iteration start over

For loops

Here's a **common mistake**

If you only want `range` values, you need the blank identifier:

```
// two vars: _ is the index (ignored),  
//           v is the value  
  
for _, v := range myArray {  
    fmt.Println(v)  
}
```

Sometimes you may not get a compile error for a type mismatch if you use only the one-var format (a slice of `ints`!)

The `_` is an untyped, reusable “variable” placeholder

Labels and loops

Sometimes we need to break or continue the outer loop
(nested loop for quadratic search)

```
outer:  
    for k := range testItemsMap {           // keys  
        for _, v := range returnedData {    // values in list  
            if k == v.ID {                  // found it!  
                continue outer  
            }  
        }  
        t.Errorf("key not found: %s", k)  
    }  
}
```

We need a label to refer to the outer loop

Switch

A switch is another choice between alternatives

It is a shortcut replacing a series of if-then statements

```
switch a := f.Get(); a {  
    case 0, 1, 2:  
        fmt.Println("underflow possible")  
  
    case 3, 4, 5, 6, 7, 8:  
  
    default:  
        fmt.Println("warning: overload")  
}
```

Alternatives may be empty and **do not fall through** (break is not required)

Switch on true

Arbitrary comparisons may be made for an switch with no argument

```
a := f.Get()

switch {
case a <= 2:
    fmt.Println("underflow possible")

case a <= 8:
    // evaluated in order

default:
    fmt.Println("warning: overload")
}
```

Packages

Everything lives in a package

Every standalone program has a `main` package

```
package main

import "fmt"

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

Nothing is “global”; it’s either in your package or in another

It’s either at `package` scope or `function` scope

Package-level declarations

You can declare anything at *package* scope

```
package secrets

const DefaultUUID = "00000000-0000-0000-0000-000000000000"
var secretKey string

type k8secret struct {
    ...
}

func Do(it string) error {
    ...
}
```

But you can't use the short declaration operator :=

Packages control visibility

Every name that's **capitalized** is exported

```
package secrets

import . .

type internal struct {
    .
}

func GetAll(space, name string) (map[string]string, error) {
    .
}
```

That means another package in the program can import it
(within a package, *everything* is visible even across files)

Imports

Each *source file* in your package must import what it needs

```
package secrets

import (
    "encoding/base64"
    "encoding/json"
    "fmt"
    "os"
    "strings"
)
```

It may only import what it needs; unused imports are an error

Generally, files of the same package live together in a directory

No cycles

A package “A” cannot import a package that imports A

```
package A  
  
import "B"  
  
//-----
```

```
package B  
  
import "A"    // WRONG
```

Move common dependencies to a third package, or eliminate them

Initialization

Items within a package get initialized before `main`

```
const A = 1

var B int = C
var C int = A

func Do() error {
    . .
}

func init() {
    . .
}
```

Only the runtime can call `init`, also before `main`

What makes a good package?

A package should embed deep functionality behind a simple API

```
package os

func Create(name string) (*File, error)
func Open(name string) (*File, error)

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

The Unix file API is perhaps the best example of this model

Roughly five functions hide a lot of complexity from the user

Declarations & Compatibility

Declaration

There are six ways to introduce a name:

- Constant declaration with `const`
- Type declaration with `type`
- Variable declaration with `var`
(must have type or initial value, sometimes both)
- Short, initialized variable declaration of any type `:=`
only inside a function
- Function declaration with `func`
(methods may *only* be declared at package level)
- Formal parameters and named returns of a function

Variable declarations

There are several ways to write a variable declaration:

```
var a int           // 0 by default
```

```
var b int = 1
```

```
var c = 1          // int
```

```
var d = 1.0         // float64
```

```
var (
    x, y int
    z    float64
    s    string
)
```

Short declarations

The short declaration operator `:=` has some rules:

1. It can't be used outside of a function
2. It must be used (instead of `var`) in a control statement (`if`, etc.)
3. It must declare at least one *new* variable

```
err := doSomething();
```

```
err := doSomethingElse(); // WRONG
```

```
x, err := getSomeValue(); // OK; err is not redeclared
```

4. It won't re-use an existing declaration from an outer scope

Shadowing short declarations

Short declarations with `:=` have some gotchas

```
func main() {
    n, err := fmt.Println("Hello, playground")

    if _, err := fmt.Println(n); err != nil {
        fmt.Println(err)
    }
}
```

Compile error: the first `err` is unused

This follows from the scoping rules, because `:=` is a declaration and the second `err` is in the scope of the `if` statement

Shadowing short declarations

Short declarations with := have some gotchas

```
func BadRead(f *os.File, buf []byte) error {
    var err error

    for {
        n, err := f.Read(buf) // shadows 'err' above

        if err != nil {
            break           // causes return of WRONG value
        }

        foo(buf)
    }

    return err // will always be nil
}
```

Structural typing

It's the same type if it has the same *structure or behavior*

```
a := [...]int{1, 2, 3}  
b := [3]int{}
```

```
a = b           // OK
```

```
c := [4]int{}
```

```
a = c           // NOT OK
```

Go uses *structural* typing in most cases

Structural typing

It's the same type if it has the same structure or behavior:

- arrays of the same size and base type
- slices with the same base type
- maps of the same key and value types
- structs with the same sequence of field names/types
- functions with the same parameter & return types

Named typing

It's the only the same type if it has the same *declared type name*

```
type x int

func main() {
    var a x          // x is a defined type; base int
    b := 12         // b defaults to int
    a = b           // TYPE MISMATCH
    a = 12          // OK, untyped literal
    a = x(b)        // OK, type conversion
}
```

Go uses *named* typing for non-function *user-declared* types

Numeric literals

Go keeps “arbitrary” precision for literal values (256 bits or more)

- Integer literals are untyped
 - assign a literal to any size integer without conversion
 - assign an integer literal to float, complex also
- Ditto float and complex; picked by syntax of the literal
`2.0` or `2e9` or `2.0i` or `2i3`
- Mathematical constants can be very precise
`Pi = 3.14159265358979323846264338327950288419716939937510582097494459`
- Constant arithmetic done at compile time doesn't lose precision

Operators

Basic operators

Arithmetic: numbers only except + on string

+ - * / % ++ --

Comparison: only numbers/strings support order

== != < > <= >=

Boolean: only booleans, with shortcut evaluation

! && ||

Bitwise: operate on integers

& | ^ << >> &[^]

Assignment: as above for binary operations

= += -= *= /= %=
&= |= ^= <<= >>= &[^]=

Operator precedence

There are only five levels of precedence, otherwise left-to-right:

Operators like multiplication:

* / % << >> & &[^]

Operators like addition:

+ - | ^

Comparison operators:

== != < <= > >=

Logical and:

&&

Logical or:

||

Programming in Go

Matt Holiday
Christmas 2020



Input/Output

Standard I/O

Unix has the notion of three standard I/O streams

They're open by default in every program

Most modern programming languages have followed this convention:

- Standard input
- Standard output
- Standard error (output)

These are normally mapped to the console/terminal but can be *redirected*

```
find . -name '*.go' | xargs grep -n "rintf" > print.txt
```

Formatted I/O

We've been using the `fmt` package to do I/O

By default, we've been printing to standard output

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("printing a line to standard output")

    fmt.Fprintln(os.Stderr, "printing to error output")
}
```

A whole family of functions

The `fmt` package uses reflection and can print anything;
some of the functions take a *format string*

```
// always os.Stdout  
  
fmt.Println(...interface{}) (int, error)  
fmt.Printf(string, ...interface{}) (int, error)  
  
// print to anything that has the correct Write() method  
  
fmt.Fprintln(io.Writer, ...interface{}) (int, error)  
fmt.Fprintf(io.Writer, string, ...interface{}) (int, error)  
  
// return a string  
  
fmt.Sprintln(...interface{}) string  
fmt.Sprintf(string, ...interface{}) string
```

Format codes

The `fmt` package uses format codes reminiscent of C

<code>%s</code>	the uninterpreted bytes of the string or slice
<code>%q</code>	a double-quoted string safely escaped with Go syntax
<code>%c</code>	the character represented by the corresponding Unicode code point
<code>%d</code>	base 10
<code>%x</code>	base 16, with lower-case letters for a-f
<code>%f</code>	decimal point but no exponent, e.g. 123.456
<code>%t</code>	the word true or false
<code>%v</code>	the value in a default format when printing structs, the plus flag (<code>%+v</code>) adds field names
<code>%#v</code>	a Go-syntax representation of the value
<code>%T</code>	a Go-syntax representation of the type of the value
<code>%%</code>	a literal percent sign; consumes no value [escape]

Read the godoc, Luke: <https://golang.org/pkg/fmt/>

Format code examples

A few examples:

```
a := 12  
b := 345  
c := 1.2  
d := 3.45
```

```
fmt.Printf("%d %d\n", a, b)          // 12 345  
fmt.Printf("%#x %x\n", a, b)          // 0xc 159  
fmt.Printf("%f %.2f\n", c, d)          // 1.200000 3.45
```

```
fmt.Println()
```

```
fmt.Printf("|%6d|%6d|\n", a, b)      // | 12| 345|  
fmt.Printf("|%06d|%06d|\n", a, b)      // |000012|000345|  
fmt.Printf("|%-6d|%-6d|\n", a, b)      // |12| 345|  
fmt.Printf("|%.2f|%.2f|\n", c, d)      // | 1.20| 3.45|
```

Format code examples

`%#v` and `%T` are very useful for describing what something is:

```
s := []int{1, 2, 3}
a := [3]rune{'a', 'b', 'c'}
m := map[string]int{"and":1, "or":2}

fmt.Printf("%T\n", s)      // []int
fmt.Printf("%v\n", s)      // [1 2 3]
fmt.Printf("%#v\n", s)     // []int{1, 2, 3}

fmt.Printf("%T\n", a)      // [3]int32
fmt.Printf("%q\n", a)      // ['a' 'b' 'c']
fmt.Printf("%v\n", a)      // [97 98 99]
fmt.Printf("%#v\n", a)     // [3]int32{97, 98, 99}

fmt.Printf("%T\n", m)      // map[string]int
fmt.Printf("%v\n", m)      // map[and:1 or:2]
fmt.Printf("%#v\n", m)     // map[string]int{"and":1, "or":2}
```

File I/O

Package `os` has functions to open or create files,
list directories, etc. and hosts the `File` type

Package `io` has utilities to read and write; `bufio` provides the
buffered I/O scanners, etc.

Package `io/ioutil` has extra utilities such as reading an entire file
to memory, or writing it out all at once

Package `strconv` has utilities to convert to/from string representations

Con•cat•enate

```
package main
import ("io"; "log"; "os")

func main() {
    for _, fname := range os.Args[1:] {
        file, err := os.Open(fname)

        if err != nil {
            log.Fatal(err)
        }

        if _, err = io.Copy(os.Stdout, file); err != nil {
            log.Fatal(err)
        }

        file.Close()
    }
}
```

Reading a file

Wait, what's going on here?

```
if f, err := os.Open(fname); err != nil {  
    fmt.Fprintln(os.Stderr, "bad file:", err)  
} . . .
```

We often call functions whose 2nd return value is a possible error

```
func Open(name string) (*File, error)
```

where the `error` can be compared to `nil`, meaning no error

Always check the error — the file might not really be open!

Reading a file and calculating its size

```
package main

import ("fmt"; "io/ioutil"; "os")

func main() {
    fname := os.Args[1]
    if f, err := os.Open(fname); err != nil {
        fmt.Fprintln(os.Stderr, "bad file:", err)
    } else if d, err := ioutil.ReadAll(f); err != nil {
        fmt.Fprintln(os.Stderr, "can't read:", err)
    } else {
        fmt.Printf("The file has %d bytes\n", len(d))
    }
}
```

If run on itself (the source file), it prints “The file has 333 bytes”

```
package main

import ("bufio"; "fmt"; "os"; "strings")

func main() {
    for _, fname := range os.Args[1:] {
        var lc, wc, cc int

        file, err := os.Open(fname)

        if err != nil {
            fmt.Fprintln(os.Stderr, err)
            continue
    }}
```

```
scan := bufio.NewScanner(file)

for scan.Scan() {
    s := scan.Text()

    cc += len(s)
    wc += len(strings.Fields(s))
    lc++

}

fmt.Printf(" %7d %7d %7d %s\n", lc, wc, cc, fname)
file.Close()
}
```

Programming in Go

Matt Holiday
Christmas 2020



Functions

Functions in Go

Functions are “first class” objects; you can:

- Define them — even inside another function
- Create anonymous *function literals*
- Pass them as function parameters / return values
- Store them in variables
- Store them in slices and maps (but not as keys)
- Store them as fields of a structure type
- Send and receive them in channels
- Write methods against a function type
- Compare a function var against `nil`

Function scope

Almost anything can be defined inside a function

```
func Do() error {
    const a = 21

    type b struct {
        . . .
    }

    var c int

    reallyDoIt := func() { // only anonymous funcs with assignment
        . . .
    }
}
```

Methods cannot be defined in a function (only at package scope)

Function signatures

The *signature* of a function is the order & type of its parameters and return values

It does not depend on the *names* of those parameters or returns

```
var try func(string, int) string  
  
func Do(a string, b int) string {  
    . . .  
}  
  
func NotDo(x string, y int) (a string) {}  
    . . .  
}
```

These functions have the same *structural* type

Parameter terms

A function declaration lists **formal** parameters

```
func do(a, b int) int { ... }
```

A function call has **actual** parameters (a/k/a “arguments”)

```
result := do(1, 2)
```

A parameter is passed **by value** if the function gets a copy;
the caller can't see changes to the copy

A parameter is passed **by reference** if the function can modify
the actual parameter such that the caller sees the changes

Parameter passing

By value:

- numbers
- bool
- arrays
- structs

By reference:

- things passed by pointer (`&x`)
- strings (but they're immutable)
- slices
- maps
- channels

Parameter passing

Parameters may be passed *by value*

```
func do(b [3]int) int {
    b[0] = 0
    return b[1]
}

func main() {
    a := [3]int{1, 2, 3}
    v := do(a)

    fmt.Println(a, v)      // [1,2,3] 2
}
```

Here `do` gets a copy of the array so any change to it is not seen by the caller

Parameter passing

Parameters may be passed *by reference*

```
func do(b []int) int {
    b[0] = 0
    return b[1]
}

func main() {
    a := []int{1, 2, 3}
    v := do(a)

    fmt.Println(a, v)      // [0,2,3] 2
}
```

Here `do` gets a copy of the slice descriptor which *refers to* the same backing array, so the caller sees changes

Parameter passing

Parameters may be passed *by value* or *by reference*

```
func do(m1 map[int]int) {
    m1[3] = 1
    m1 = make(map[int]int)
    m1[4] = 4
    fmt.Println(m1)           // map[4:4]
}

func main() {
    m := map[int]int{4: 1}
    fmt.Println(m)           // map[4:1]
    do(m)
    fmt.Println(m)           // map[3:1 4:1]
}
```

We can re-assign `m1` because the formal parameter is a local variable

Parameter passing

Parameters may be passed *by value* or *by reference*

```
func do(m1 *map[int]int) {
    (*m1)[3] = 1
    *m1 = make(map[int]int)
    (*m1)[4] = 4
    fmt.Println(*m1)           // map[4:4]
}

func main() {
    m := map[int]int{4: 1}
    fmt.Println(m)             // map[4:1]
    do(&m)
    fmt.Println(m)             // map[4:4]
}
```

The map pointer `m` allows replacing the caller's entire map with a new one

Parameter passing: the ultimate truth

Parameters may be passed *by value* or ~~by reference~~

Actually, **all** parameters are passed by copying something
(i.e., by value)

If the thing copied is a pointer or descriptor, then the shared
backing store (array, hash table, etc.) can be changed through it

Thus we think of it as “by reference”

Return values

Functions can have multiple return values

```
func doIt(a int, b []int) int {  
    . . .  
    return 1  
}  
  
func doItAgain(a string) (int, error) {  
    . . .  
    return 1, nil  
}
```

Every return statement must have all the values specified

Recursion

A function may call itself; the trick is knowing when to stop

```
func walk(node *tree.T) int {
    if node == nil {
        return 0
    }

    return node.value + walk(node.left) + walk(node.right)
}
```

This works because each function call adds context to the stack
and unwinds it when done

If you don't have good stopping criteria, the program will crash

Defer

Deferred execution

How do we make sure something gets done?

- close a file we opened
- close a socket / HTTP request we made
- unlock a mutex we locked
- make sure something gets saved before we're done
- ...

The `defer` statement captures a function *call* to run later

Defer

We need to ensure the file closes no matter what

```
func main() {
    f, err := os.Open("my_file.txt")

    if err != nil {
        . . .
    }

    defer f.Close()

    // and do something with the file
}
```

The call to `Close` is guaranteed to run at *function exit*
(don't defer closing the file until we know it really opened!)

Defer

We need to ensure the file closes no matter what

```
func main() {
    f := os.Stdin

    if len(os.Args) > 1 {
        if f, err := os.Open(os.Args[1]); err != nil {
            . . .
        }
        defer f.Close()
    }

    // and do something with the file
}
```

Notice that the `defer` will *not* execute when we leave the `if` block

Defer gotcha #1

The scope of a `defer` statement is the *function*

```
func main() {
    for i := 1; i < len(os.Args); i++ {
        f, err := os.Open(os.Args[i])
        .
        .
        .
        defer f.Close()
        .
        .
        .
    }
}
```

The deferred calls to `Close` must wait until function exit
(we might run out of file descriptors before that!)

Defer gotcha #2

Unlike a closure, defer copies arguments to the deferred call

```
func main() {
    a := 10

    defer fmt.Println(a)

    a = 11

    fmt.Println(a)
}

// prints 11, 10
```

The parameter a gets **copied** at the defer statement (not a reference)

Defer gotcha #2

A `defer` statement runs before the `return` is “done”

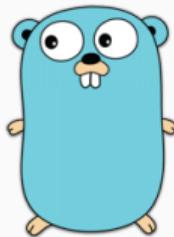
```
func doIt() (a int) {
    defer func() {
        a = 2
    }()
    a = 1
    return
}
// returns 2
```

We have a named return value and a “naked” return

The deferred anonymous function can update that variable

Programming in Go

Matt Holiday
Christmas 2020



Closures

Scope vs lifetime

Scope is static, based on the code at compile time

Lifetime depends on program execution (*runtime*)

```
package xyz

func doIt() *int {
    var b int
    ...
    return &b
}
```

Variable b can only be seen inside doIt, but its value will live on

The value (object) will live so long as part of the program keeps a pointer to it

What is a closure?

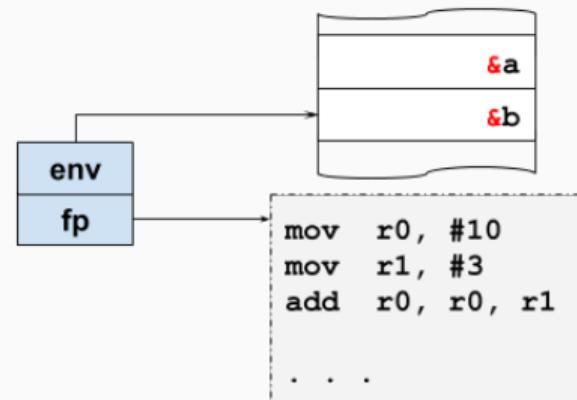
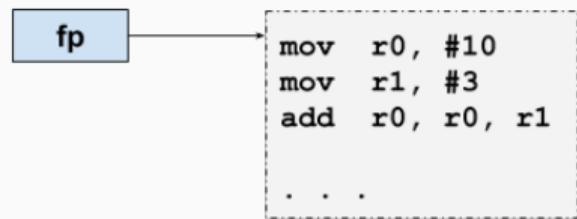
A *closure* is when a function inside another function “closes over” one or more local variables of the outer function

```
func fib() func() int {  
    a, b := 0, 1  
  
    return func() int {  
        a, b = b, a+b  
        return b  
    }  
}
```

The inner function gets a **reference** to the outer function's vars

Those variables may end up with a much longer *lifetime* than expected — as long as there's a reference to the inner function

Closures: how they work



Closures: scope vs lifetime

```
package main
import "fmt"

func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return b
    }
}

func main() {
    f := fib()
    for x := f(); x < 100; x = f() {
        fmt.Println(x) // 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
    }
}
```

Closures: scope vs lifetime

The inner variables continue to live on

```
func fib() func() int {
    a, b := 0, 1

    // return a closure over a & b
}

func main() {
    f := fib()

    // f keeps ahold of a and b and updates them

    fmt.Println(f(), f(), f(), f(), f(), f())
}
```

The inner function continues to mutate the variables it references

Closures: unique environment

The inner variables are unique to each closure

```
func fib() func() int {
    a, b := 0, 1
    // return a closure over a & b
}

func main() {
    f, g := fib(), fib()

    // f & g have their own copies of a & b
    fmt.Println(f(), f(), f(), f(), f(), f())
    fmt.Println(g(), g(), g(), g(), g(), g())
}
```

They print identical lines 1 2 3 ... (think $a_1, b_1; a_2, b_2$ etc.)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func do(d func()) {
    d()
}

func main() {
    for i := 0; i < 4; i++ {
        v := func() {
            fmt.Printf("%d %p\n", i, &i)
        }
        do(v)
    }
}
```

The program prints 0, 1, 2, 3; addresses all the same (**why?**)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        s[i] = func() {
            // they all point to the same "i"
            fmt.Printf("%d %p\n", i, &i)
        }
    }

    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 4 each time; addresses all the same (**why?**)

Closure gotcha

Avoid closing over a variable that is mutating (a loop index)

```
func main() {
    s := make([]func(), 4)

    for i := 0; i < 4; i++ {
        j := i // closure capture
        s[i] = func() {
            fmt.Printf("%d %p\n", j, &j)
        }
    }
    for i := 0; i < 4; i++ {
        s[i]()
    }
}
```

The program prints 0, 1, 2, 3 as expected; addresses different

Programming in Go

Matt Holiday
Christmas 2020

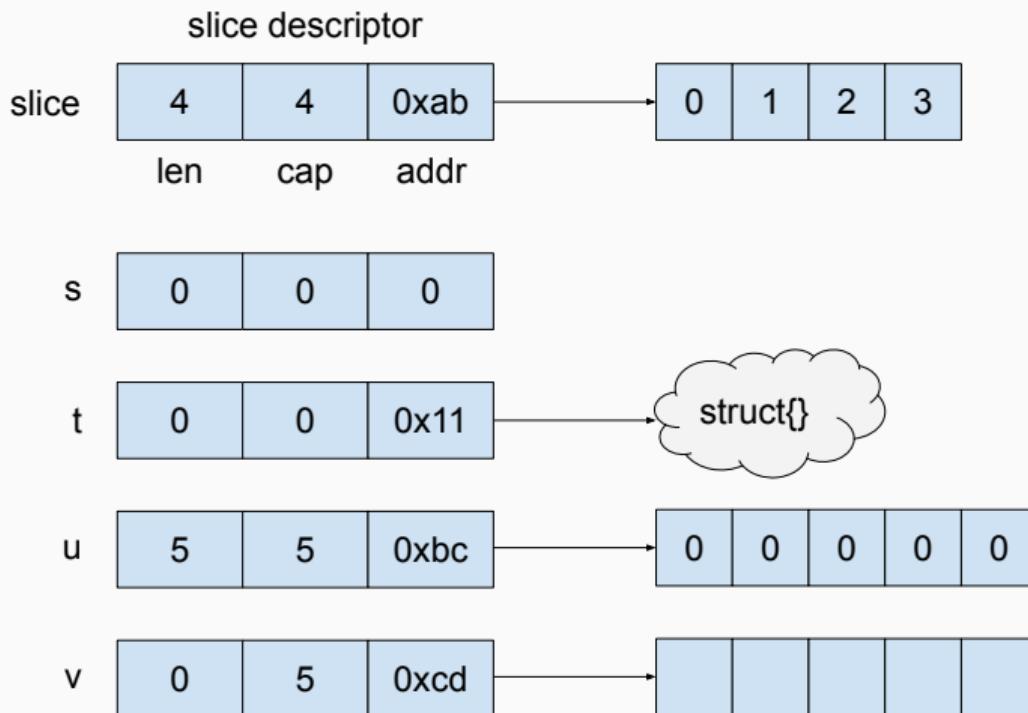


Slices in Detail

Empty vs nil slice

```
var s []int  
  
t := []int{}  
u := make([]int, 5)  
v := make([]int, 0, 5)  
  
fmt.Printf("%d, %d, %T, %5t, %#[3]v\n", len(s), cap(s), s, s == nil)  
fmt.Printf("%d, %d, %T, %5t, %#[3]v\n", len(t), cap(t), t, t == nil)  
fmt.Printf("%d, %d, %T, %5t, %#[3]v\n", len(u), cap(u), u, u == nil)  
fmt.Printf("%d, %d, %T, %5t, %#[3]v\n", len(v), cap(v), v, v == nil)  
  
0, 0, []int, true, []int(nil)  
0, 0, []int, false, []int{}  
5, 5, []int, false, []int{0, 0, 0, 0, 0}  
0, 5, []int, false, []int{}
```

Slice follow-up



Empty vs nil slice

Slices (and maps) encoding differently in JSON when nil

```
package main

import ("encoding/json"; "fmt")

func main() {
    var a []int

    j1, _ := json.Marshal(a)
    fmt.Println(string(j1))      // null

    b := []int{}

    j2, _ := json.Marshal(b)
    fmt.Println(string(j2))      // []

}
```

Ugly #1: Slice length vs capacity

```
a := [3]int{1, 2, 3}
b := a[0:1]          // b is a slice of a's first item

fmt.Println("a =", a)    // a = [1 2 3]
fmt.Println("b =", b)    // b = [1]

c := b[0:2]          // WTF? but the array has 3 entries

fmt.Println("a =", a)    // a = [1 2 3]
fmt.Println("c =", c)    // c = [1 2]

fmt.Println(len(b))     // prints 1
fmt.Println(cap(b))     // prints 3

fmt.Println(len(c))     // prints 2
fmt.Println(cap(c))     // prints 3
```

Ugly #1: Slice length vs capacity

Go 1.2 added the “three index” slice operator `[i:j:k]` where length is $j-i$ and capacity is $k-i$

```
d := a[0:1:1]          // this is what you probably meant  
  
fmt.Println("a =", a)    // a = [1 2 3]  
fmt.Println("d =", d)    // d = [1]  
  
fmt.Println(len(d))     // prints 1  
fmt.Println(cap(d))     // prints 1
```

Ugly #2: Slice mutating underlying array

```
a := [3]int{1, 2, 3}; b := a[0:1]; c := b[0:2]

b = append(b, 4)           // grows b, mutates a
fmt.Printf("a[%p] = %v\n", &a, a) // a[0xc000014020] = [1 4 3]
fmt.Printf("b[%p] = %[1]v\n", b) // b[0xc000014020] = [1 4]

c = append(c, 5)           // grows c, mutates a
fmt.Printf("a[%p] = %v\n", &a, a) // a[0xc000014020] = [1 4 5]
fmt.Printf("c[%p] = %[1]v\n", c) // c[0xc000014020] = [1 4 5]

c = append(c, 6)           // forces allocation!
fmt.Printf("a[%p] = %v\n", &a, a) // a[0xc000014020] = [1 4 5]
fmt.Printf("c[%p] = %[1]v\n", c) // c[0xc000078030] = [1 4 5 6]

c[0] = 9                   // mutates a different array!
fmt.Printf("a[%p] = %v\n", &a, a) // a[0xc000014020] = [1 4 5]
fmt.Printf("c[%p] = %[1]v\n", c) // c[0xc000078030] = [9 4 5 6]
```

Programming in Go

Matt Holiday
Christmas 2020



Homework

Homework #2

Exercise 5.5 from *GOPL*: implement `countWordsAndImages`

Actually, given some HTML as raw text, parse it into a document and then call your counting routine to detect and count words and images (you can follow the book's example).

Don't worry about getting HTML from an HTTP query; we're not there yet.

See Homework #1 for counting words.

What happens if the HTML document is empty?

Homework #2

```
package main

import (
    "bytes"
    "fmt"
    "os"
    "strings"

    "golang.org/x/net/html"
)
```

Homework #2

```
var raw = `<!DOCTYPE html>
<html>
<body>
  <h1>My First Heading</h1>
  <p>My first paragraph.</p>
  <p>HTML images are defined with the img tag:</p>
  
</body>
</html>`
```

Homework #2

```
func main() {
    doc, err := html.Parse(bytes.NewReader([]byte(raw)))

    if err != nil {
        fmt.Fprintf(os.Stderr, "parse failed: %s\n", err)
        os.Exit(-1)
    }

    words, pics := countWordsAndImages(doc)

    fmt.Printf("%d words and %d images\n", words, pics)
}

// outputs "14 words and 1 images"
```

Homework #2

```
func countWordsAndImages(doc *html.Node) (int, int) {
    var words, images int

    visit(doc, &words, &images)

    return words, images
}
```

Homework #2

```
func visit(n *html.Node, words, pics *int) {
    // if it's an element node then see what tag it has

    if n.Type == html.TextNode {
        *words += len(strings.Fields(n.Data))
    } else if n.Type == html.ElementNode && n.Data == "img" {
        *pics++
    }

    // then visit all the children using recursion

    for c := n.FirstChild; c != nil; c = c.NextSibling {
        visit(c, words, pics)
    }
}
```

Programming in Go

Matt Holiday
Christmas 2020



Structs

Structs

We've already seen a couple of aggregate types:

- slices & arrays group a sequence of the same type
- maps use one type to index a collection of another type

A **struct** is an aggregate of possibly disparate types

```
type Employee struct {
    Name    string
    Number  int
    Boss    *Employee
    Hired   time.Time
}
```

Notice that we can have a pointer to the type we're defining

Structs

In other languages it's a “record” (using database terminology)

It's parts are called “fields” and each must have a unique name

Access to the fields is with “dot” notation

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:    "Matt",
    Number:  72,
    Boss:    employees["Lamine"],
    Hired:   time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Struct initialization

With names, selected fields can be initialized

The others default to “zero”

```
employees := make(map[string]*Employee)

var matt = Employee{
    Name:    "Matt",
    Number: 72,
    Hired:   time.Date(2017, 12, 9, 16, 30, 0, 0, time.UTC),
}

employees[matt.Name] = &matt
```

Here `matt.Boss` is left to the default value `nil`

Anonymous Structs

Anonymous structs are possible:

```
var album = struct {
    title string
    artist string
    year int,
    copies int,
} {
    "The White Album",
    "The Beatles",
    1968,
    10000000,
}
```

Initialization can be done without names by setting all fields in the correct order

Anonymous Structs

But assignment can be very inconvenient:

```
var s1 struct {
    A int
    B string
}

func main() {
    s1 = struct{A int; B string}{1, "a"}

    fmt.Println(s1)
}
```

Struct compatibility

But assignment can be very inconvenient:

```
var s1 struct {
    A int
    B string
}

var s2 struct {
    A int
    B string
}

func main() {
    s1 = struct{A int; B string}{1, "a"}
    s2 = s1
    fmt.Println(s1, s2)
}
```

Struct compatibility

Two **struct** types are compatible if

- the fields have the same types and names
- in the same order
- and the same tags (*)

A **struct** may be copied or passed as a parameter in its entirety

A **struct** is comparable if all its fields are comparable

The zero value for a **struct** is “zero” for each field in turn

Struct compatibility

Anonymous structs are compatible if they follow the rules:

```
func main() {
    v1 := struct {
        X int `json:"foo"`
    }{1}

    v2 := struct {
        X int `json:"foo"`
    }{2}

    v1 = v2
    fmt.Println(v1)      // prints {2}
}
```

Struct compatibility

Types with different *user-declared* names are never compatible:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"foo"`
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = v2          // TYPE MISMATCH
    fmt.Println(v1)
}
```

Struct compatibility

Named struct types are *convertible* if they are compatible:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"foo"`
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = T1(v2)          // type conversion
    fmt.Println(v1)       // prints {2}
}
```

Struct compatibility

From Go 1.8 tag differences don't prevent type conversions:

```
type T1 struct {
    X int `json:"foo"`
}

type T2 struct {
    X int `json:"bar"` // NOTE difference
}

func main() {
    v1 := T1{1}
    v2 := T2{2}

    v1 = T1(v2)          // type conversion
    fmt.Println(v1)       // prints {2}
}
```

Passing structs

Structs are passed by value unless a pointer is used

```
var white album

func soldAnother(a album) {
    // oops
    a.copies++
}

func soldAnother(a *album) {
    // what you intended
    a.copies++
}
```

Note that “dot” notation works on pointers too, equivalent to `(*a).copies`

Struct gotcha

Here's an annoying little issue in Go:

```
employees := make(map[string]Employee) // NOTE: not *Employee

var matt Employee{
    Name:    "Matt",
    Number:  72,
    Boss:    &lamine,
    Hired:   time.Date( . . . ),
}

employees[matt.Name] = matt

employees["Matt"].Number++ // can't do this
```

A map entry is not addressable; issue #3117

Make the zero value useful

“It is usually desirable that the zero value be a natural or sensible default.

For example, in `bytes.Buffer`, the initial value of the struct is a ready-to-use empty buffer.” (*GOPL §4.4*)

```
type Buffer struct {
    buf      []byte // contents are the bytes buf[offset : len(buf)]
    off      int     // read at &buf[off], write at &buf[offset]
    lastRead readOp // last read operation, so that Unread* can work correctly.
}
```

which has a `nil` slice we can append to, and `off` starts as 0; the “zero” value for `readOp` is `opInvalid`.

Empty structs

A struct with no fields is useful; it takes up no space

```
// a set type (instead of bool)
```

```
var isPresent map[int]struct{}
```

```
// a very cheap channel type
```

```
done := make(chan struct{})
```

Struct Tags and JSON

Struct tags

Tags are a part of a `struct` definition captured by the compiler

They are available to code that uses *reflection*

```
type Response struct {
    Page int      `json:"page"`
    Words []string `json:"words,omitempty"`
}
```

Sometimes multiple tags are appropriate, separated by a space

```
type Response struct {
    Page int      `json:"page" db:"page"`
    Words []string `json:"words,omitempty" db:"words"`
}
```

Reflection in action: JSON support

The JSON package in Go uses reflection on Go objects

```
r := &Response{Page: 1, Words: []string{"up", "in", "out"}}

j, _ := json.Marshal(r)      // ignoring errs
fmt.Println(string(j))

// {"page":1,"words":["up","in","out"]}

var r2 Response

json.Unmarshal(j, &r2)      // ignoring errs
fmt.Printf("%#v\n", r2)

// main.Response{Page:1, Words:[]string{"up", "in", "out"}}
```

Reflection in action: JSON support

“omitempty” causes a nil object to be ignored by Marshal

```
r := &Response{Page: 1, Words: []string{}}

j, _ := json.Marshal(r)      // ignoring errs
fmt.Println(string(j))

// {"page":1}

var r2 Response

json.Unmarshal(j, &r2)      // ignoring errs
fmt.Printf("%#v\n", r2)

// main.Response{Page:1, Words:[]string(nil)}
```

Struct tags have many uses

Tags can also be used in conjunction with SQL queries

```
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    When string `db:"created"`
}

func PutStats(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
              VALUES (:name, :created);`
    _, err := db.NamedExec(stmt, item)

    return err
}
```

Struct tag gotcha

Only **exported** (capitalized) field names are convertible

```
import "github.com/jmoiron/sqlx"

type item struct {
    Name string `db:"name"`
    When string `db:"created"` // oops
}

func PutItem(db *sqlx.DB, item *item) error {
    stmt := `INSERT INTO items (name, created)
              VALUES (:name, :created);`

    _, err := db.NamedExec(stmt, item) // FAILS, missing when

    return err
}
```

Programming in Go

Matt Holiday
Christmas 2020



Regular Expressions & Search

Searching in strings

Use the `strings` package for simple searches

Carefully use the `regexp` package for complex searches and validation

“Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.” — Jamie Zawinski

Searching in strings

Go's regular expression syntax is a subset of what some other languages have

That avoids the performance impact of *catastrophic backtracking*

See articles by Russ Cox on regular expressions:

<https://swtch.com/~rsc/regexp/>

Simple string searches

Boolean searches:

- `strings.HasPrefix(s, substr)`
- `strings.HasSuffix(s, substr)`
- `strings.Contains(s, substr)`

Location searches:

- `strings.LastIndex(s, substr)`
- `strings.LastIndexByte(s, char)`

Search and replace:

- `strings.Replace(s, substr, replacement, count)`
- `strings.ReplaceAll(s, substr, replacement)`

Location by regex

A regular expression can match a variable number of runes
(a plain string search can't do that)

```
func main() {
    te := "aba abba abbba"
    re := regexp.MustCompile("b+")
    mm := re.FindAllString(te, -1)
    id := re.FindAllStringIndex(te, -1)

    fmt.Println(mm) // [b bb bbb]
    fmt.Println(id) // [[1 2] [5 7] [10 13]]

    for _, d := range id {
        fmt.Println(te[d[0]:d[1]]) // b bb bbb
    }
}
```

Replacement by regex

A regular expression can match a variable number of runes
(a plain string search can't do that)

```
func main() {
    te := "aba abba abbba"
    re := regexp.MustCompile("b+")
    up := re.ReplaceAllStringFunc(te, strings.ToUpper)

    fmt.Println(up)    // aBa aBBa aBBBa
}
```

Regular expressions

Syntax for repetition and character classes:

- . is any character
- .* is zero or more
- .+ is one or more
- .? is zero or one (prefer one)
- a{ n } is n repetitions of the letter “a”
- a{ n, m } is n to m repetitions of the letter “a”
- [a-z] is a **character class** (here letters a-z)
- [^a-z] is an *negated* class (here anything *except* a-z)

Regular expressions

Syntax for location:

- xy is “x” followed by “y” (a [sub]string!)
- $x|y$ is either “x” *or* “y”
- $\wedge x$ is “x” at the beginning
- $x\$\!$ is “x” at the end
- $\wedge x\$$ is “x” by itself (it’s the whole string)
- $\backslash b$ is a word boundary
- $\backslash bx\backslash b$ is the word “x” by itself (inside the string)
- (x) is a **capture group**

Regular expressions

Some built-in character classes:

- `\d` is a decimal digit
- `\w` is a *word* character ([0-9A-Za-z_])
- `\s` is *whitespace*
- `[:alpha:]` is any alphabetic character
- `[:alnum:]` is any alphanumeric character
- `[:punct:]` is any punctuation character
- `[:print:]` is any printable character
- `[:xdigit:]` is any hexadecimal character

See <https://golang.org/pkg/regexp/syntax/>

UUID validation

A UUID has the form 072665ee-a034-4cc3-a2e8-9f1822c4ebbb

So we can try to validate one by matching a regular expression

A very simple (and not quite correct) example:

```
uure := `-[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}`
ufmt := regexp.MustCompile(uure)

if !ufmt.MatchString(ustr) {
    return fmt.Errorf("%s is not a UUID", ustr)
}
```

UUID validation

What was wrong:

- hex characters could be in upper case
- RFC 4122 has specific requirements for certain characters

The format is really xxxxxxxx-xxxx-Vxxx-Wxxx-xxxxxxxxxxxx

where V is a version (1-5)

and W is a format marker (10bb — one of 8, 9, a, b)

UUID validation

```
var uure = `[[:xdigit:]]{8}-[[:xdigit:]]{4}-`+
    `[[1-5][:xdigit:]]{3}-[89abAB][[:xdigit:]]{3}-[[:xdigit:]]{12}`+
var ufmt = regexp.MustCompile(uure)

var test = []string{
    "072665ee-a034-4cc3-a2e8-9f1822c4ebbb",
    "072665ee-a034-6cc3-a2e8-9f1822c4ebbb", // wrong version
    "072665ee-a034-4cc3-72e8-9f1822c4ebbb", // wrong format
}

func main() {
    for _, t := range test {
        if !uu.MatchString(t) {
            fmt.Println(t, "fails")
        }
    }
}
```

Search and replace with capture

```
var phre = `\\((([:digit:]\\{3})\\) ([[:digit:]\\{3})-([[:digit:]\\{4})`  
var pfmt = regexp.MustCompile(phre)  
  
func main() {  
    orig := "call me at (214) 514-3232 today"  
    match := pfmt.FindStringSubmatch(orig)  
  
    fmt.Printf("%q\\n", match)  
  
    if len(match) > 3 {  
        fmt.Printf("+1 %s-%s-%s\\n", match[1], match[2], match[3])  
    }  
}  
  
// ["(214) 514-3232" "214" "514" "3232"] // match [submatch ...]  
// +1 214-514-3232
```

Search and replace with capture

```
var phre = `\\((([:digit:]\\{3})\\) ([[:digit:]\\{3})-([[:digit:]\\{4})`  
var pfmt = regexp.MustCompile(phre)  
  
func main() {  
    orig := "call me at (214) 514-3232 today"  
    match := pfmt.FindStringSubmatch(orig)  
  
    fmt.Printf("%q\n", match)  
  
    int1 := pfmt.ReplaceAllString(orig, "+1 ${1}-${2}-${3}")  
  
    fmt.Println(int1)  
}  
  
// ["(214) 514-3232" "214" "514" "3232"]  
// call me at +1 214-514-3232 today
```

URL validation with capture

```
var uure = `^((http|https)://(([a-zA-Z0-9\-\.\.]+\.[a-zA-Z]{2,4})`+
           `(?::([0-9]+))?)?/?)(([a-zA-Z0-9\-\.\_\.\\?\.,\\'\\\\\\\+&#$\=~]*))$`  
var ufmt = regexp.MustCompile(uure)  
  
var test = []string{  
    "http://matt.com/hello",  
    "http://matt.com:8080/hello/",  
    "http://matt.com:8080/hello?a=1&b=2",  
    "http://matt.com:8080/hello?a=1&b=2&c=3",  
}  
  
func main() {  
    for i, t := range test {  
        match := u.FindStringSubmatch(t)  
        fmt.Printf("%d: %q\n", i, match)  
    }  
}
```

URL validation with capture

```
var uure = `^((http|https)://(([a-zA-Z0-9\-\.\.]+\.[a-zA-Z]{2,4})`+
           `(?:([0-9]+))?)?/?)(([a-zA-Z0-9\-\.\_\.\\?\\,\\'\\\\\\\\+&\\$#\\=~]*))$`  
var ufmt = regexp.MustCompile(uure)  
var vars = regexp.MustCompile(`(\w+)=(\w+)`)  
var test = []string{  
    "http://matt.com:8080/hello?a=1&b=2",  
    "http://matt.com:8080/hello?a=1&b=2&c=3",  
}  
  
func main() {  
    for i, t := range test {  
        match := u.FindStringSubmatch(t)  
        fmt.Printf("%d: %q\n", i, match)  
        if len(match) > 4 && strings.Contains(match[4], "?") {  
            fmt.Printf("    %q\n", vars.FindAllStringSubmatch(match[4], -1))  
        }  
    }  
}
```

Programming in Go

Matt Holiday
Christmas 2020



Pointer/Value Semantics

Pointers vs values

Pointers	Values
Shared, not copied	Copied, not shared

Value semantics lead to higher integrity, particularly with concurrency
(don't share)

Pointer semantics *may* be more efficient

Pointers vs values

Common uses of pointers:

- some objects can't be copied safely (mutex)
- some objects are too large to copy efficiently
(consider pointers when size > 64 bytes)
- some methods need to change (mutate) the receiver [later!]
- when decoding protocol data into an object
(JSON, etc.; often in a variable argument list)

```
var r Response
```

```
err := json.Unmarshal(j, &r)
```

- when using a pointer to signal a “null” object

Must not copy

Any **struct** with a mutex **must** be passed by reference:

```
type Employee struct {
    mu    sync.Mutex
    Name string
    . . .
}

func do(emp *Employee) {
    emp.mu.Lock()

    defer emp.mu.Unlock()
    . . .
}
```

May copy

Any small struct under 64 bytes probably should be copied:

```
type Widget struct {
    ID      int
    Count   int
}

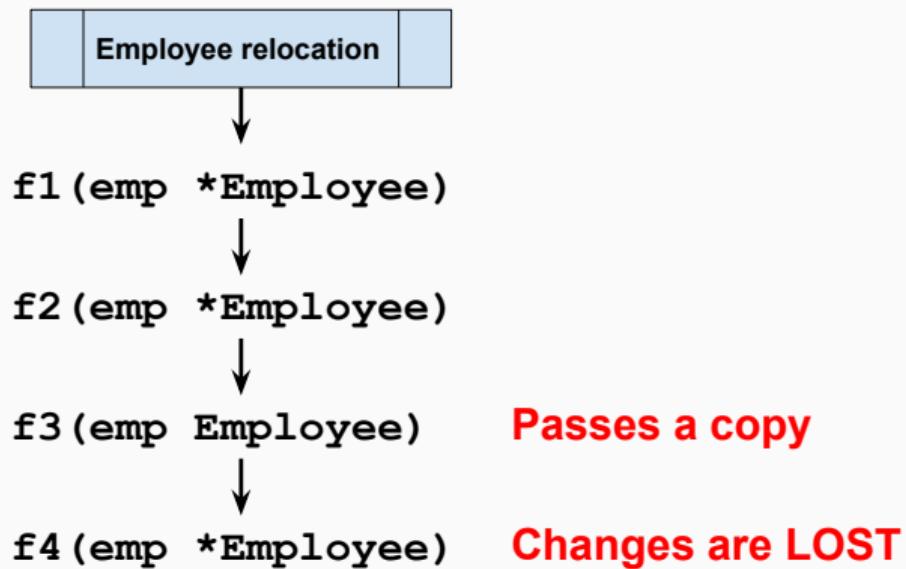
func Expend(w Widget) Widget {
    w.Count--

    return w
}
```

Note that Go routinely copies string & slice descriptors

Semantic consistency

If a thing is to be shared, then always pass a pointer
thanks to Bill Kennedy



Stack allocation

Stack allocation is more efficient

Accessing a variable directly is more efficient than following a pointer

Accessing a dense sequence of data is more efficient than sparse data
(an array is faster than a linked list, etc.)

Heap allocation

Go would prefer to allocate on the stack, but sometimes can't

- a function returns a pointer to a local object
- a local object is captured in a function closure
- a pointer to a local object is sent via a channel
- any object is assigned into an interface
- any object whose size is variable at runtime (slices)

The use of `new` has nothing to do with it

Build with the flag `-gcflags -m=2` to see the escape analysis

For loops

The value returned by `range` is always a copy

```
for i, thing := range things {  
    // thing is a copy  
    . . .  
}
```

Use the index if you need to mutate the element:

```
for i := range things {  
    things[i].which = whatever  
    . . .  
}
```

Slice safety

Anytime a function mutates a slice that's passed in, we must return a copy

```
func update(things []thing) []thing {  
    . . .  
    things = append(things, x)  
    return things  
}
```

That's because the slice's backing array may be reallocated to grow

Slice safety

Keeping a pointer to an element of a slice is risky

```
type user struct { name string; count int }

func addTo(u *user) { u.count++ }

func main() {
    users := []user{{"alice", 0}, {"bob", 0}}

    alice := &users[0]           // risky
    amy := user{"amy", 1}

    users = append(users, amy)

    addTo(alice)                // alice is likely a stale pointer
    fmt.Println(users)           // so alice's count will be 0
}
```

Capturing a slice the hard way

Each append copies a reference to `item` with its last value

```
items := [][]byte{{1, 2}, {3, 4}, {5, 6}, {7, 8}}
a := [][]byte{}

for _, item := range items {
    a = append(a, item[:])
}

fmt.Println(items) // [[1 2] [3 4] [5 6] [7 8]]
fmt.Println(a)    // [[7 8] [7 8] [7 8] [7 8]]
```

Capturing a slice the hard way

Each append copies a reference to `item` with its last value, so we need to copy each `item` into a (non-empty) slice

```
items := [][]byte{{1, 2}, {3, 4}, {5, 6}, {7, 8}}
a := [][]byte{}

for _, item := range items {
    i := make([]byte, len(item))

    copy(i, item[:]) // make unique
    a = append(a, i)
}

fmt.Println(items) // [[1 2] [3 4] [5 6] [7 8]]
fmt.Println(a)    // [[1 2] [3 4] [5 6] [7 8]]
```

Capturing the loop variable, again

Taking the address of a mutating loop variable is wrong

```
func (r OfferResolver) Changes() []ChangeResolver {
    var result []ChangeResolver

    // wrong

    for _, change := range r.d.Status.Changes {
        result = append(result, ChangeResolver{&change}) // WRONG
    }

    return result
}
```

Wrong: all the returned resolvers point to the last change in the list

Capturing the loop variable, again

Copy the loop variable *inside the loop* before taking its address

```
func (r OfferResolver) Changes() []ChangeResolver {
    var result []ChangeResolver

    for _, c := range r.d.Status.Changes {
        change := c // make unique

        result = append(result, ChangeResolver{&change})
    }

    return result
}
```

Now each resolver has its own change data allocated on the heap

Programming in Go

Matt Holiday
Christmas 2020



Networking with HTTP

Go network libraries

The Go standard library has many packages for making web servers:

That includes:

- client & server sockets
- route multiplexing
- HTTP and HTML, including HTML templates
- JSON and other data formats
- cryptographic security
- SQL database access
- compression utilities
- image generation

There are also lots of 3rd-party packages with improvements

A very short web server

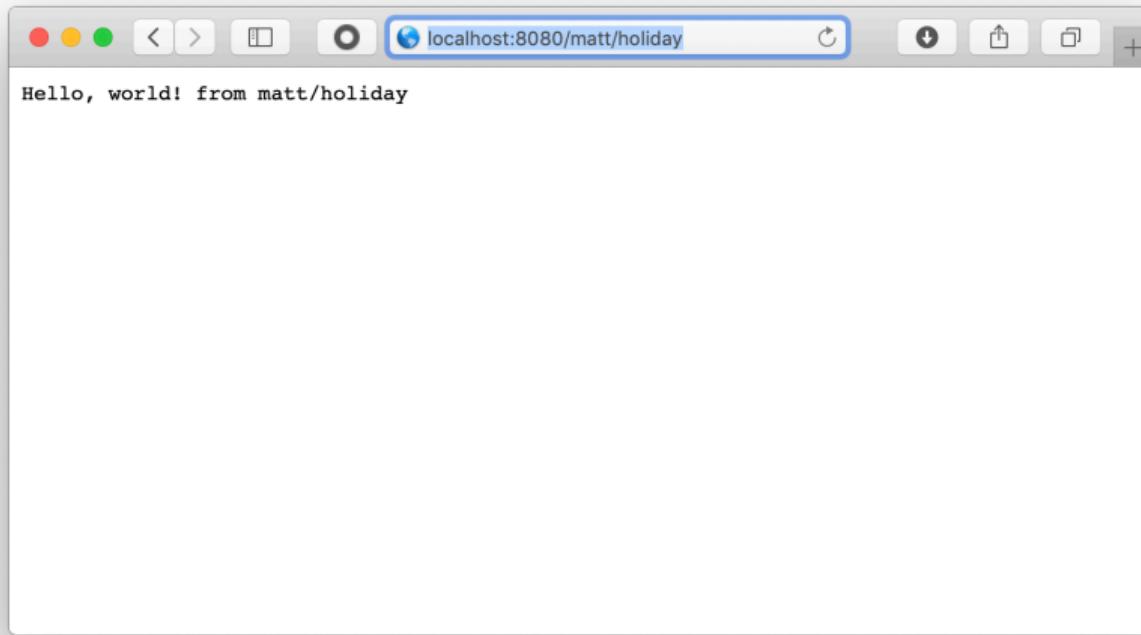
```
package main

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

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}

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

A very short web server



Go HTTP design

An HTTP handler function is an instance of an interface

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}

// The HTTP framework can call a method on a function type

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}
```

A very short web client

```
package main

import ("fmt"; "io/ioutil"; "net/http"; "os")

func main() {
    resp, _ := http.Get("http://localhost:8080/" + os.Args[1])
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        body, _ := ioutil.ReadAll(resp.Body)
        fmt.Println(string(body))
    }
}

// $ go run client.go matt/holiday
// Hello, world! from matt/holiday
```

A simple JSON REST client

```
package main

import ("fmt"; "io/ioutil"; "net/http")
const url = "https://jsonplaceholder.typicode.com"

func main() {
    resp, _ := http.Get(url + "/todos/1")
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        body, _ := ioutil.ReadAll(resp.Body)
        fmt.Println(string(body))
    }
}

// $ go run client.go
// {"userId": 1, "id": 1, "title": "delectus aut autem", "completed": false}
```

A simple JSON REST client

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

type todo struct {
    UserID      int      `json:"userID"`
    ID          int      `json:"id"`
    Title       string   `json:"title"`
    Completed   bool     `json:"completed"`
}

const base = "https://jsonplaceholder.typicode.com"
```

A simple JSON REST client

```
func main() {
    var item todo

    resp, _ := http.Get(base + "/todos/1")

    defer resp.Body.Close()

    body, _ := ioutil.ReadAll(resp.Body)

    _ := json.Unmarshal(body, &item)

    fmt.Printf("%#v\n", item)
}

// $ go run client.go
// main.todo{UserID:1, ID:1, Title:"delectus aut autem", Completed:false}
```

Serving from a template

```
package main

import (
    "encoding/json"
    "html/template"
    "io/ioutil"
    "log"
    "net/http"
)

type todo struct {
    UserID      int      `json:"userID"`
    ID          int      `json:"id"`
    Title       string   `json:"title"`
    Completed   bool     `json:"completed"`
}

const base = "https://jsonplaceholder.typicode.com/"
```

Serving from a template

```
var form = `
<h1>Todo #{{.ID}}</h1>
<div>{{printf "User %d" .UserID}}</div>
<div>{{printf "%s (completed: %t)" .Title .Completed}}</div>`

func handler(w http.ResponseWriter, r *http.Request) {
    var item todo

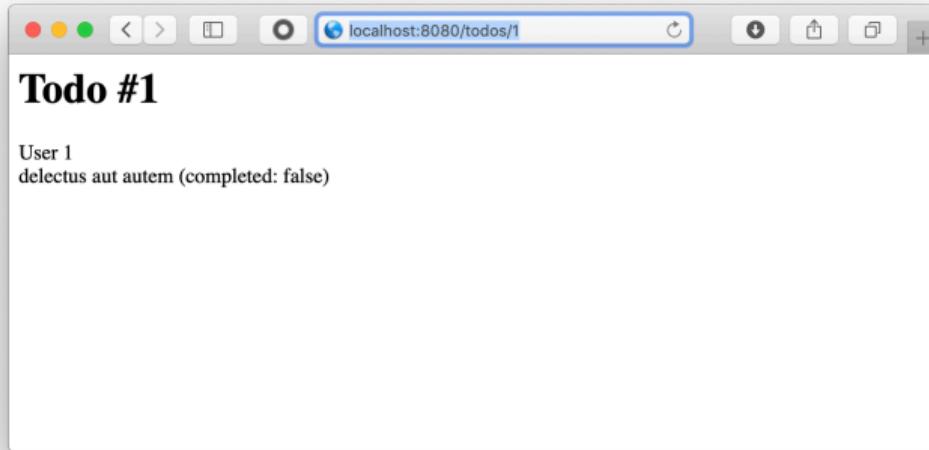
    resp, _ := http.Get(base + r.URL.Path[1:])
    defer resp.Body.Close()
    body, _ := ioutil.ReadAll(resp.Body)
    _ = json.Unmarshal(body, &item)

    tmpl := template.New("mine")

    tmpl.Parse(form)
    tmpl.Execute(w, item)
}
```

Serving from a template

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



Serving up an error

```
func handler(w http.ResponseWriter, r *http.Request) {
    var item todo

    resp, _ := http.Get(base + r.URL.Path[1:])
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        http.NotFound(w, r)
        return
    }

    body, _ := ioutil.ReadAll(resp.Body)
    _ = json.Unmarshal(body, &item)

    tmpl := template.New("mine")
    tmpl.Parse(form)
    tmpl.Execute(w, item)
}
```

Serving up a wiki

See the Golang article [Writing Web Applications](#)

The tutorial includes:

- creating a data structure with load and save methods
- using the net/http package to build web applications
- using the html/template package to process HTML templates
- using the regexp package to validate user input
- using closures

Programming in Go

Matt Holiday
Christmas 2020

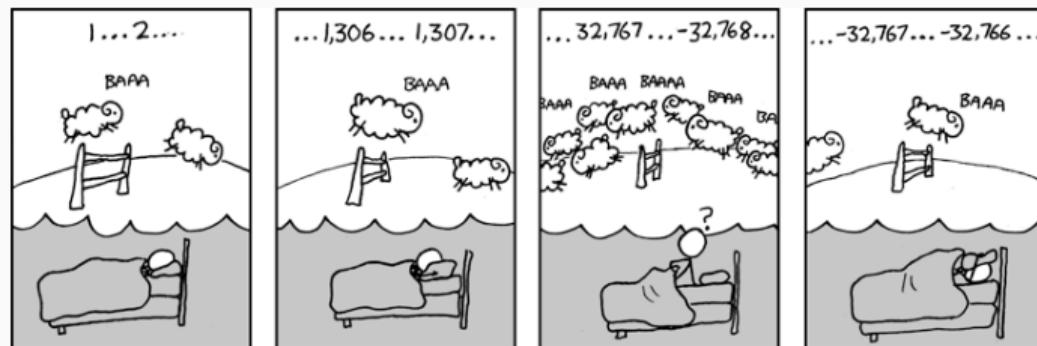


Homework

Homework #3

Exercise 4.12 from *GOPL*: fetching from the web

The popular web comic “xkcd” has a JSON interface. For example, a request to <https://xkcd.com/571/info.0.json> produces a detailed description of comic 571, one of many favorites. Download each URL (once!) and build an offline index. Write a tool `xkcd` that, using this index, prints the *URL, date, and title* of each comic whose *title or transcript* matches a *list of search terms* provided on the command line.



Homework #3

What the raw data looks like:

```
{  
    "month":        "4",  
    "num":          571,  
    "link":         "",  
    "year":         "2009",  
    . . .  
    "transcript":  "[[Someone is in bed, . . . long int." ,  
    "img":          "https://imgs.xkcd.com/comics/cant_sleep.png",  
    "title":        "Can't Sleep",  
    "day":          "20"  
}
```

Homework #3

Sample output:

```
$ go run xkcd-load.go xkcd.json
skipping 404: got 404
skipping 2403: got 404
skipping 2404: got 404
read 2401 comics
```

```
$ go run xkcd-find.go xkcd.json someone bed sleep
read 2318 comics
https://xkcd.com/571/ 4/20/2009 "Can't Sleep"
found 1 comics
```

Find the solution at: matt4biz/go-class-exer-4.12

First program: downloader

The first program downloads comic metadata

1. We will read until we get two 404 responses in a row
2. Each request will generate a JSON object as a string
3. We will bracket them with [and] and a comma between
(so no comma before the first object)
4. The result will be a file with a JSON array of metadata objects,
so we won't need to decode
5. We will optionally take a filename from the command line for output

First program: downloader

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
)
```

First program: downloader

This function gets the data for a single comic strip

```
func getOne(i int) []byte {
    url := fmt.Sprintf("https://xkcd.com/%d/info.0.json", i)
    resp, err := http.Get(url)

    if err != nil {
        fmt.Fprintf(os.Stderr, "stopped reading: %s\n", err)
        os.Exit(-1)
    }

    defer resp.Body.Close()
```

First program: downloader

```
if resp.StatusCode != http.StatusOK {
    // easter egg: #404 returns HTTP 404 - not found
    fmt.Fprintf(os.Stderr, "skipping %d: got %d\n",
                i, resp.StatusCode)
    return nil
}

body, err := ioutil.ReadAll(resp.Body)

if err != nil {
    fmt.Fprintf(os.Stderr, "bad body: %s\n", err)
    os.Exit(-1)
}

return body
}
```

First program: downloader

```
func main() {
    var (
        output     io.WriteCloser = os.Stdout
        err        error
        cnt, fails int
        data       []byte
    )

    if len(os.Args) > 0 {
        output, err = os.Create(os.Args[1])

        if err != nil {
            fmt.Fprintln(os.Stderr, err)
            os.Exit(-1)
        }

        defer output.Close()
    }
}
```

First program: downloader

```
// the output will be in the form of a JSON array,  
// so add the brackets before and after  
  
fmt.Fprint(output, "[")  
defer fmt.Fprint(output, "]")  
  
for i := 1; fails < 2; i++ { // stop on 2 404s in a row  
    if data = getOne(i); data == nil {  
        fails++  
        continue  
    }  
  
    if cnt > 0 {  
        fmt.Fprint(output, ",") // 0B1  
    }  
    . . .
```

First program: downloader

```
    . . .

_, err = io.Copy(output, bytes.NewBuffer(data))

if err != nil {
    fmt.Fprintf(os.Stderr, "stopped: %s", err)
    os.Exit(-1)
}

fails = 0
cnt++
}

fmt.Fprintf(os.Stderr, "read %d comics\n", cnt)

// $ go run ./xkcd-load.go xkcd.json
// read 2318 comics
}
```

Second program: searcher

The second program reads in and searches all comics

1. We will require a filename from the command line for input
2. We will read in and decode to a slice of objects
3. We will take multiple search terms from the command line
4. We will select comics that match all words by doing a quadratic search (nested loops)
5. We will compare all strings in lower case

Second program: searcher

```
// { "month":      "4",
//     "num":        571,
//     "year":       "2009",
//     "transcript": "[[Someone is in bed, . . . long int." ,
//     "img":        "https://imgs.xkcd.com/comics/cant_sleep.png",
//     "title":      "Can't Sleep",
//     "day":        "20"
// }

type xkcd struct {
    Num      int      `json:"num"`
    Day      string   `json:"day"`
    Month    string   `json:"month"`
    Year     string   `json:"year"`
    Title    string   `json:"title"`
    Transcript string `json:"transcript"`
}
```

Second program: searcher

```
func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "no file given")
        os.Exit(-1)
    }

    fn := os.Args[1]

    if len(os.Args) < 3 {
        fmt.Fprintln(os.Stderr, "no search term")
        os.Exit(-1)
    }
}
```

Second program: searcher

```
var items []xkcd
var terms []string
var input io.ReadCloser
var cnt int
var err error

if input, err = os.Open(fn); err != nil {
    fmt.Fprintf(os.Stderr, "invalid file: %s", err)
    os.Exit(-1)
}

// read the file in one big step
err = json.NewDecoder(input).Decode(&items)

if err != nil {
    fmt.Fprintf(os.Stderr, "decode failed: %s\n", err)
    os.Exit(-1)
}
```

Second program: searcher

```
fmt.Fprintf(os.Stderr, "read %d comics\n", len(items))

for _, t := range os.Args[2:] {
    terms = append(terms, strings.ToLower(t))
}

outer:
for _, item := range items {
    title := strings.ToLower(item.Title)
    transcript := strings.ToLower(item.Transcript)

    for _, term := range terms {
        if !strings.Contains(title, term) &&
           !strings.Contains(transcript, term) {
            continue outer
        }
    }
    . . .
}
```

Second program: searcher

```
    . . .

    fmt.Printf("https://xkcd.com/%d/ %s/%s/%s\n",
        item.Num, item.Month, item.Day, item.Year)
    cnt++
}

fmt.Fprintf(os.Stderr, "found %d comics\n", cnt)

// $ go run ./xkcd-find.go xkcd.json someone bed sleep
// read 2318 comics
// https://xkcd.com/571/ 4/20/2009
// found 1 comics
}
```

Programming in Go

Matt Holiday
Christmas 2020



Object-Oriented Programming

Object-oriented programming

For many people, the essential elements of object-oriented programming have been:

- abstraction
- encapsulation
- polymorphism
- inheritance

Sometimes those last two items are combined or confused

Go's approach to OO programming is similar but different

Abstraction

Abstraction: decoupling behavior from the implementation details

The Unix file system API is a great example of effective abstraction

Roughly five basic functions hide all the messy details:

- open
- close
- read
- write
- ioctl

Many different operating system things can be treated like files

Encapsulation

Encapsulation: hiding implementation details from misuse

It's hard to maintain an abstraction if the details are exposed:

- the internals may be manipulated in ways contrary to the concept behind the abstraction
- users of the abstraction may come to depend on the internal details — but those might change

Encapsulation usually means controlling the visibility of names (“private” variables)

Polymorphism

Polymorphism literally means “many shapes” — multiple types behind a single interface

Three main types are recognized:

- ad-hoc: typically found in function/operator overloading
- parametric: commonly known as “generic programming”
- subtype: subclasses substituting for superclasses

“Protocol-oriented” programming uses explicit interface types, now supported in many popular languages (an ad-hoc method)

In this case, *behavior is completely separate from implementation*, which is good for abstraction

Inheritance

Inheritance has conflicting meanings:

- substitution (subtype) polymorphism
- structural sharing of implementation details

In theory, inheritance should always imply subtyping:
the subclass should be a “kind of” the superclass

See the [Liskov substitution principle](#)

Theories about substitution can be pretty messy

Why would inheritance be bad?

It injects a dependence on the superclass into the subclass:

- what if the superclass changes behavior?
- what if the abstract concept is leaky?

Not having inheritance means better encapsulation & isolation

“Interfaces will force you to think in term of communication between objects”
— Nicolò Pignatelli in [Inheritance is evil](#)

See also [Composition over inheritance](#) and [Inheritance tax](#) (Pragmatic)

One more view

“Object-oriented programming to me means only messaging,
local retention and protection and hiding of state-process,
and extreme late-binding of all things.” — Alan Kay

He wrote this to

- de-emphasize inheritance hierarchies as a key part of OOP
- emphasize the idea of self-contained objects sending messages to each other
- emphasize polymorphism in behavior

OO in Go

Go offers four main supports for OO programming:

- encapsulation using the package for visibility control
- abstraction & polymorphism using interface types
- enhanced *composition* to provide structure sharing

Go does not offer inheritance or substitutability based on types

Substitutability is based only on **interfaces**: purely a function of abstract **behavior**

See also [Go for Gophers](#)

Classes in Go

Classes in Go

Not having classes can be liberating!

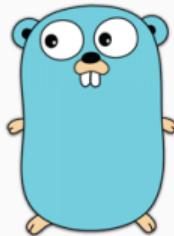
Go allows defining methods on any user-defined type, rather than only a “class”

Go allows any object to implement the method(s) of an interface, not just a “subclass”

Let's get away from defining OOP in terms of a particular language's features!

Programming in Go

Matt Holiday
Christmas 2020



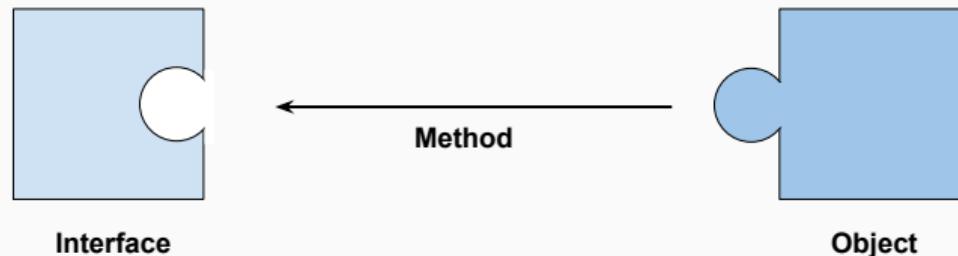
Methods and Interfaces

Why have methods?

An **interface** specifies *abstract* behavior in terms of **methods**

```
type Stringer interface { // in "fmt"
    String() string
}
```

Concrete types offer methods that *satisfy* the interface



Methods are type-bound functions

A **method** is a special type of function (syntax from Oberon-2)

It has a **receiver** parameter *before* the function name parameter

```
type IntSlice []int

func (is IntSlice) String() string {
    strs []string

    for _, v := range is {
        strs = append(strs, strconv.Itoa(v))
    }

    return "[" + strings.Join(strs, ";") + "]"
}
```

Why have methods?

Only **methods** may be used to satisfy an **interface**

```
func main() {
    var v IntSlice = []int{1, 2, 3}
    var s fmt.Stringer = v

    for i, x := range v {
        fmt.Printf("%d: %d\n", i, x)
    }

    fmt.Printf("%T %[1]v\n", s)
    fmt.Printf("%T %[1]v\n", v) // Uses String() method (if available)
}
```

An `IntSlice` value **“is a”** `fmt.Stringer` because it implements the `String()` method

Why interfaces?

Without interfaces, we'd have to write (many) functions
for (many) concrete types, possibly coupled to them

```
func OutputToFile(f *File, . . .) { . . . }
func OutputToBuffer(b *Buffer, . . .) { . . . }
func OutputToSocket(s *Socket, . . .) { . . . }
```

Better — we want to define our function in terms of *abstract behavior*

```
type Writer interface {
    Write([]byte) (int, error)
}

func OutputTo(w io.Writer, . . .) { . . . }
```

Why interfaces?

An interface specifies required behavior as a **method set**

Any type that implements that method set satisfies the interface:

```
type Stringer interface { // in "fmt"
    String() string
}

func (is IntSlice) String() string {
    . .
}
```

This is known as *structural* typing (“duck” typing)

No type will declare itself to implement `ReadWrite` explicitly

Not just structs

A method may be defined on any **user-declared** (named) type*

That means methods can't be declared on `int`, but

```
type MyInt int  
  
func (i MyInt) String() string {  
    . . .  
}
```

The same method name may be bound to different types

* Some rules and restrictions apply, see package insert for details

Receivers

A method may take a *pointer* or *value* receiver, but not both

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

func (p Point) Offset(x, y float64) Point {
    return Point{p.x+x, p.y+y}
}

func (p *Point) Move(x, y float64) {
    p.x += x
    p.y += y
}
```

Taking a pointer allows the method to change the receiver (original object)

Interface variables

A variable of interface type can refer to any object that satisfies it

```
func Copy(w Writer, r Reader) (int, error) { // in "io"
    . . .
}

f1, err := os.Open("input.txt")
f2, err := os.Create("output.txt")

n, err := io.Copy(f2, f1)
```

Here `w` and `r` are references ultimately to files

But it could be a `File` and a `bytes.Buffer` source; it wouldn't care — all it needs is the specific behaviors (write & read)

Interface example

```
type ByteCounter int

func (b *ByteCounter) Write(p []byte) (int, error) {
    *b += ByteCounter(len(p)) // conversion required
    return len(p), nil
}

var c ByteCounter

f, _ := os.Open("input.txt")
n, _ := io.Copy(&c, f)           // &c required

fmt.Println(n == int64(c))      // true
```

Lots of types are **Writers** and can be written/copied to;
see also Francesc Campoy [Interfaces in Go \(2019\)](#)

Interfaces and substitution

All the methods must be present to satisfy the interface

```
var w    io.Writer
var rwc io.ReadWriteCloser

w = os.Stdout          // OK: *os.File has Write method
w = new(bytes.Buffer)  // OK: *bytes.Buffer has Write method
w = time.Second        // ERROR: no Write method

rwc = os.Stdout         // OK: *os.File has all 3 methods
rwc = new(bytes.Buffer) // ERROR: no Close method

w = rwc                // OK: io.ReadWriteCloser has Write
rwc = w                 // ERROR: no Close method
```

Which is why it pays to keep interfaces small

Interface satisfiability

The **receiver** must be of the right type (pointer or value)

```
type IntSet struct { /* ... */ }

func (*IntSet) String() string

var _ = IntSet{}.String() // ERROR: String needs *IntSet (l-value)

var s IntSet
var _ = s.String()        // OK: s is a variable; &s used automatically

var _ fmt.Stringer = &s  // OK
var _ fmt.Stringer = s  // ERROR: no String method
```

We'll come back and talk about pointer vs value receivers in more detail

Interface composition

`io.ReadWriter` is actually defined by Go as two interfaces

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

Small interfaces with **composition** where needed are more flexible

Interface declarations

All methods for a given type must be declared in the same package where the type is declared

This allows a package importing the type to know all the methods at compile time

But we can always extend the type in a new package through embedding:

```
type Bigger struct {
    my.Big           // get all Big methods via promotion
}

func (b Bigger) DoIt() { // and add one more method here
    . . .
}
```

Interfaces in practice

1. Let **consumers** define interfaces
(what *minimal* behavior do they require?)
2. Keep interface declarations small
("The bigger the interface, the weaker the abstraction")
3. Compose one-method interfaces into larger interfaces (if needed)
4. Avoid coupling interfaces to particular types/implementations
5. Accept interfaces, return concrete types (if possible *)
6. Re-use standard interfaces wherever possible

* Returning **error** is a good example of an exception to this rule

Programming in Go

Matt Holiday
Christmas 2020



Composition, not Inheritance

Composition

The fields of an **embedded struct** are *promoted* to the level of the embedding structure

```
type Pair struct {
    Path string
    Hash string
}

type PairWithLength struct {
    Pair
    Length int
}

p1 := PairWithLength{Pair{/usr}, "0xfdfe"}, 121}

fmt.Println(p1.Path, p1.Length) // not p1.x.Path
```

Composition

The *methods* of an embedded struct are also promoted

Those methods **can't** see fields of the *embedding struct*

```
func (p Pair) String() string {
    return fmt.Sprintf("Hash of %v is %v", p.Path, p.Hash)
}

p1 := PairWithLength{Pair{/usr}, "0xfdfe"}, 121}

// Pair.String() doesn't have visibility to p1.Length

fmt.Println(p1) // prints "Hash of /usr is 0xfdfe"
```

Composition

The *embedding* structure may declare the same methods and so override the promoted methods

```
p1 := PairWithLength{Pair{/usr}, "0xfdfe"}, 121}

fmt.Println(p1) // uses Pair.String()

// now define the String() method

func (p PairWithLength) String() string {
    return fmt.Sprintf("Length of %v is %v with hash %v",
                       p.Path, p.Length, p.Hash)
}

fmt.Println(p1) // Length of /usr is 121 with hash 0xfdfe
```

Composition is not inheritance

A `PairWithLength` “**has a**” `Pair` but it isn’t one and **is not substitutable** for `Pair`

```
func Filename(p Pair) string {
    return filepath.Base(p.Path)
}

p1 := PairWithLength{Pair{/usr", "0xfdfe"}, 121}

a := Filename(p1) // NOT ALLOWED even though p1.Path exists
```

The only substitution is through interface types!

Composition is not inheritance

We can make an interface that `PairWithLength` will satisfy with a method promoted from `Pair`

```
func (p Pair) Filename() string {
    return p.Path
}

interface Filenamer {
    Filename() string
}

// this works because Pair's method is promoted

var fn Filenamer = PairWithLength{Pair{/usr", "0xfdfe"}, 121}

name := fn.Filename()
```

Composition with pointer types

A struct can embed a pointer to another type; promotion of its fields and methods works the same way

```
type Fizgig struct {
    *PairWithLength
    Broken bool
}

fg := Fizgig{
    &PairWithLength{Pair{/usr, "0xfdfe"}, 121},
    false,
}

fmt.Println(fg)

// Length of /usr is 121 with hash 0xfdfe
```

Sorting

Sortable interface

sort.Interface is defined as

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int

    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool

    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

and sort.Sort as

```
func Sort(data Interface)
```

Sortable built-ins

Slices of strings can be sorted using `StringSlice`

```
// defined in the sort package
// type StringSlice []string

entries := []string{"charlie", "able", "dog", "baker"}

sort.Sort(sort.StringSlice(entries))

fmt.Println(entries) // [able baker charlie dog]
```

Sorting example

Implement `sort.Interface` to make a type sortable:

```
type Organ struct {
    Name   string
    Weight int
}

type Organs []Organ

func (s Organs) Len() int      { return len(s) }
func (s Organs) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
```

From Andrew Gerrand's [Go for Gophers](#)

Sorting example

Implement `sort.Interface` to make a type sortable:

```
type ByName struct{ Organs }

func (s ByName) Less(i, j int) bool {
    return s.Organs[i].Name < s.Organs[j].Name
}

type ByWeight struct{ Organs }

func (s ByWeight) Less(i, j int) bool {
    return s.Organs[i].Weight < s.Organs[j].Weight
}
```

Here we use *struct composition* which promotes the `Organs` methods

Sorting example

Make a struct of the correct type on the fly to sort:

```
s := []Organ{  
    {"brain", 1340}, {"heart", 290},  
    {"liver", 1494}, {"pancreas", 131},  
    {"spleen", 162},  
}  
  
sort.Sort(ByWeight{s})      // pancreas first  
fmt.Println(s)  
  
sort.Sort(ByName{s})       // brain first  
fmt.Println(s)
```

```
[{pancreas 131} {spleen 162} {heart 290} {brain 1340} {liver 1494}]  
[{brain 1340} {heart 290} {liver 1494} {pancreas 131} {spleen 162}]
```

Sorting in reverse

Use `sort.Reverse` which is defined as:

```
type reverse struct {
    // This embedded Interface permits Reverse to use the
    // methods of another Interface implementation.
    Interface
}

// Less returns the opposite of the embedded implementation's Less method.
func (r reverse) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}

// Reverse returns the reverse order for data.
func Reverse(data Interface) Interface {
    return &reverse{data}
}
```

Sorting in reverse

Let's use `StringSlice` again:

```
// defined in the sort package
// type StringSlice []string

entries := []string{"charlie", "able", "dog", "baker"}

sort.Sort(sort.Reverse(sort.StringSlice(entries)))

fmt.Println(entries) // [dog charlie baker able]
```

Make Nil Useful

Make the nil value useful

```
type StringStack struct {
    data []string          // "zero" value ready-to-use
}

func (s *StringStack) Push(x string) {
    s.data = append(s.data, x)
}

func (s *StringStack) Pop() string {
    if l := len(s.data); l > 0 {
        t := s.data[l-1]
        s.data = s.data[:l-1]
        return t
    }

    panic("pop from empty stack")
}
```

Nil as a receiver value

Nothing in Go prevents calling a method with a `nil` receiver

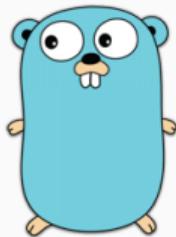
```
type IntList struct {
    Value int
    Tail  *IntList
}

// Sum returns the sum of the list elements.
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }

    return list.Value + list.Tail.Sum()
}
```

Programming in Go

Matt Holiday
Christmas 2020



Details, Details

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to type information so the correct actual method can be identified

```
var r io.Reader      // nil until initialized
var b *bytes.Buffer // ditto

r = b                // r is no longer nil!
                      // but it has a nil pointer to a Buffer
```

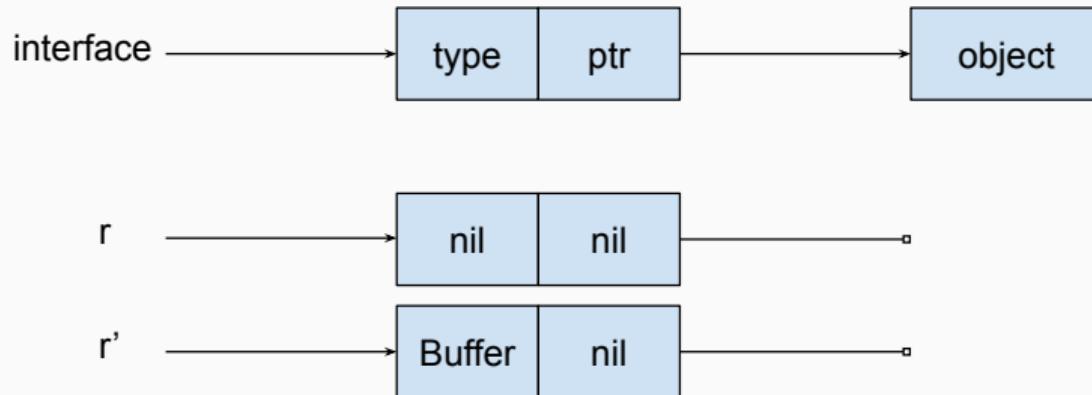
This may confuse; an interface variable is `nil` only if both parts are

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to type information so the correct actual method can be identified



Error is really an interface

We called `error` a special type, but it's really an interface

```
type error interface {
    func Error() string
}
```

We can compare it to `nil` unless we make a mistake

The mistake is to store a nil pointer to a concrete type in the `error` variable

Error is really an interface

```
type errFoo struct {
    err  error
    path string
}

func (e errFoo) Error() string {
    return fmt.Sprintf("%s: %s", e.path, e.err)
}

func XYZ(a int) *errFoo { return nil }

func main() {
    var err error = XYZ(1) // BAD: interface gets a nil concrete ptr
    if err != nil {
        fmt.Println("oops")
    }
}
```

Pointer vs value receivers

A method can be defined on a pointer to a type

```
type Point struct {
    x,y float32
}

func (p *Point) Add(x, y float32) {
    p.x, p.y = p.x + x, p.y + y
}

func (p Point) OffsetOf(p1 Point) (x float32, y float32) {
    x, y = p.x - p1.x, p.y - p1.y
    return
}
```

The same method name may **not** be bound to both T and *T

Pointer vs value receivers

Pointer methods may be called on non-pointers and vice versa

Go will automatically use * or & as needed

```
p1 := new(Point)      // *Point, at (0,0)
p2 := Point{1, 1}

p1.OffsetOf(p2)        // same as (*p1).OffsetOf(p2)

p2.Add(3, 4)           // same as (&p2).Add(3, 4)
```

Except & may only be applied to objects that are *addressable*

Pointer vs value receivers

Compatibility between objects and receiver types

	Pointer	L-Value	R-Value
pointer receiver	OK	OK &	Not OK
value receiver	OK *	OK	OK

A method requiring a pointer receiver may only be called on an addressable object

```
var p Point
```

```
p.Add(1, 2)           // OK, &p  
Point{1, 1}.Add(2, 3) // Not OK, can't take address
```

Consistency in receiver types

If one method of a type takes a pointer receiver, then *all* its methods should take pointers*

And in general objects of that type are probably not safe to copy

```
type Buffer struct {
    buf     []byte
    off     int
}

func (b *Buffer) ReadString(delim byte) (string, error) {
    . . .
}
```

*Well, except when they shouldn't for other reasons

Currying functions

Currying takes a function and reduces its argument count by one
(one argument gets bound, and a new function is returned)

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

func AddToA(a int) func(int) int {
    return func(b int) int {
        return Add(a, b)
    }
}

addTo1 := AddToA(1)

fmt.Println(Add(1,2) == addTo1(2)) // true
```

Method values

A selected method may be passed similar to a closure;
the receiver is closed over at that point

```
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance      // this is a method value

fmt.Println(distanceFromP(q))    // and can be called later
```

Reference and value semantics

A method value with a value receiver copies the receiver

If it has a pointer receiver, it copies a pointer to the receiver

```
func (p *Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance
p = Point{3, 4}

fmt.Println(distanceFromP(q)) // uses "new" value of p
```

Interfaces in Practice

Interfaces in practice

1. Let **consumers** define interfaces
(what *minimal* behavior do they require?)
2. Re-use standard interfaces wherever possible
3. Keep interface declarations small
("The bigger the interface, the weaker the abstraction")
4. Compose one-method interfaces into larger interfaces (if needed)
5. Avoid coupling interfaces to particular types/implementations
6. Accept interfaces, but return concrete types
(let the consumer of the return type decide how to use it)

Interfaces vs concrete values

“Be liberal in what you accept, be conservative in what you return”

- Put the least restriction on what parameters you accept
(the *minimal* interface)

Don’t require `ReadWriteCloser` if you only need to read

- Avoid restricting the use of your return type (the concrete value you return might fit with many interfaces!)

Returning `*os.File` is less restrictive than returning `io.ReadWriteCloser` because files have other useful methods

Returning `error` is a good example of an exception to this rule

Empty interfaces

The `interface{}` type has no methods

So it is satisfied by anything!

Empty interfaces are commonly used; they're how the formatted I/O routines can print any type

```
func fmt.Printf(f string, args ...interface{})
```

Reflection is needed to determine what the concrete type is

We'll talk about that later

Programming in Go

Matt Holiday
Christmas 2020



Homework

Interfaces in HTTP

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}

// handler matches type HandlerFunc and so interface Handler
// so the HTTP framework can call ServeHTTP on it

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}
```

Homework #4

Exercise 7.11 from *GOPL*: web front-end for a database

Add additional handlers [to the database example in §7.7, which is program gopl.io/ch7/http4] so that clients can create, read, update, and delete database entries. For example, a request of the form

/update?item=socks&price=6

will update the price of an item in the inventory and report an error if the item does not exist or if the price is invalid.

(We will *ignore* the race conditions for the purpose of this exercise.)

See my solution at: <https://github.com/matt4biz/go-class-exer-7.11>

Homework #4

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "strconv"
)

type dollars float32 // DO NOT do this in real life!

func (d dollars) String() string {
    return fmt.Sprintf("%.2f", d)
}

type database map[string]dollars
```

Homework #4

```
func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) add(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")

    if _, ok := db[item]; ok {
        msg := fmt.Sprintf("duplicate item: %q\n", item)

        http.Error(w, msg, http.StatusBadRequest) // 400
        return
    }
}
```

Homework #4

```
if f64, err := strconv.ParseFloat(price, 32); err != nil {
    msg := fmt.Sprintf("invalid price: %q\n", price)

    http.Error(w, msg, http.StatusBadRequest) // 400
} else {
    db[item] = dollars(f64)

    fmt.Fprintf(w, "added %s with price %s\n", item, dollars(f64))
}
}

func (db database) update(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")
```

Homework #4

```
if _, ok := db[item]; !ok {
    fmt.Sprintf("no such item: %q\n", item)

    http.Error(w, msg, http.StatusNotFound) // 404
    return
}

if f64, err := strconv.ParseFloat(price, 32); err != nil {
    msg := fmt.Sprintf("invalid price: %q\n", price)

    http.Error(w, msg, http.StatusBadRequest) // 400
} else {
    db[item] = dollars(f64)

    fmt.Fprintf(w, "new price %s for %s\n", dollars(f64), item)
}
```

Homework #4

```
func (db database) fetch(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")

    if _, ok := db[item]; !ok {
        fmt.Sprintf("no such item: %q\n", item)

        http.Error(w, msg, http.StatusNotFound) // 404
        return
    }

    fmt.Fprintf(w, "item %s has price %s\n", item, db[item])
}
```

Homework #4

```
func (db database) drop(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")

    if _, ok := db[item]; !ok {
        fmt.Sprintf("no such item: %q\n", item)

        http.Error(w, msg, http.StatusNotFound) // 404
        return
    }

    delete(db, item)
    fmt.Fprintf(w, "dropped %s\n", item)
}
```

Homework #4

```
func main() {
    db := database{
        "shoes": 50,
        "socks": 5,
    }

    // all these handlers are method values closing over db
    // (each is cast to a HandlerFunc)

    http.HandleFunc("/list", db.list)      // func(ResponseWriter, *Request)
    http.HandleFunc("/create", db.add)
    http.HandleFunc("/update", db.update)
    http.HandleFunc("/delete", db.drop)
    http.HandleFunc("/read", db.fetch)

    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

```
$ curl localhost:8080/list
shoes: $50.00
socks: $5.00

$ curl localhost:8080/create?item=ties\&price=13
added ties with price $13.00

$ curl localhost:8080/list
shoes: $50.00
socks: $5.00
ties: $13.00

$ curl localhost:8080/read?item=ties
item ties has price $13.00

$ curl localhost:8080/update?item=r
no such item: "r"

$ curl localhost:8080/update?item=ties\&price=A
invalid price: "A"

$ curl localhost:8080/update?item=ties\&price=12
new price $12.00 for ties

$ curl localhost:8080/list
shoes: $50.00
socks: $5.00
ties: $12.00

$ curl localhost:8080/delete?item=ties
dropped ties

$ curl localhost:8080/list
shoes: $50.00
socks: $5.00
```

Programming in Go

Matt Holiday
Christmas 2020



What is Concurrency?

Some definitions of concurrency

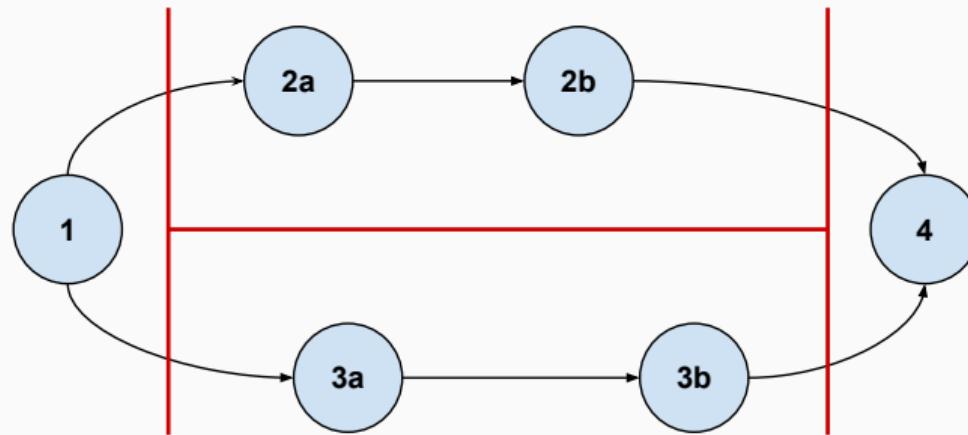
“Execution happens in some non-deterministic order”

“Undefined out-of-order execution”

“Non-sequential execution”

“Parts of a program execute out-of-order or in partial order”

Partial order



- part 1 happens before parts of 2 or 3
- both 2 and 3 complete before part 4
- the parts of 2 and 3 are ordered among themselves

Non-deterministic

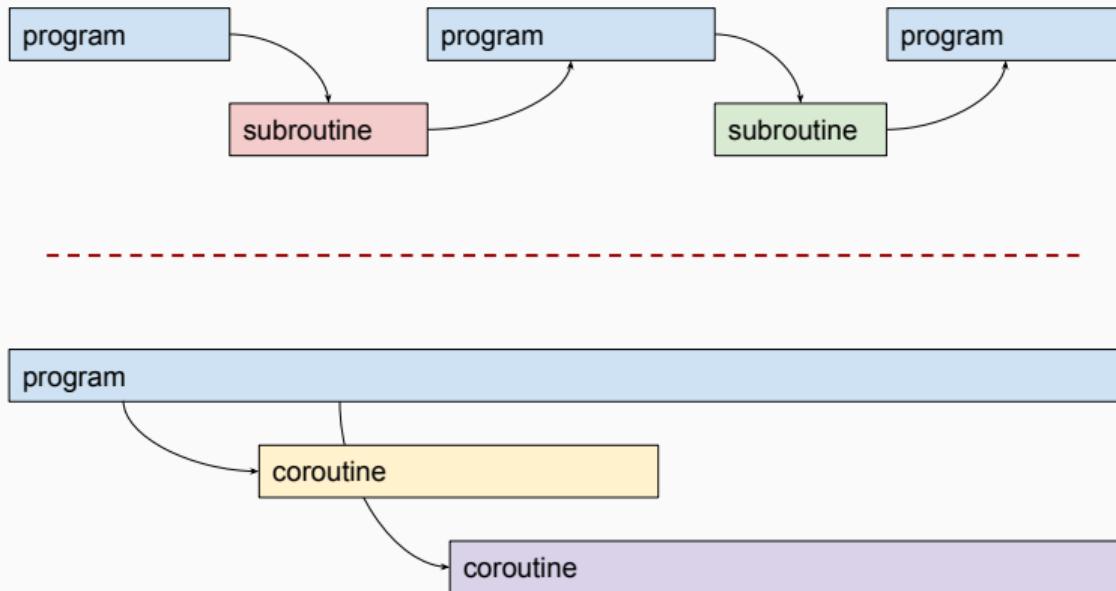
“Different behaviors on different runs, even with the same input”

1. {1, 2a, 2b, 3a, 3b, 4}
2. {1, 2a, 3a, 2b, 3b, 4}
3. {1, 2a, 3a, 3b, 2b, 4}
4. {1, 3a, 3b, 2a, 2b, 4}
5. {1, 3a, 2a, 3b, 2b, 4}

Note we don't necessarily mean different results, but a different trace of execution

Execute independently

Subroutines are subordinate, while coroutines are co-equal



Concurrency

So let's try a new definition of concurrency:

Parts of the program may execute independently in some non-deterministic (partial) order

Parallelism

Parts of a program execute independently at the same time

You can have concurrency with a single-core processor
(think interrupt handling in the operating system)

Parallelism can happen only on a multi-core processor

Concurrency doesn't make the program faster, parallelism does

Concurrency vs Parallelism

Concurrency is about *dealing with* things happening out-of-order

Parallelism is about things actually happening *at the same time*

A single program won't have parallelism without concurrency

We need concurrency to allow parts of the program to execute independently

And that's where the fun begins . . .

Race condition

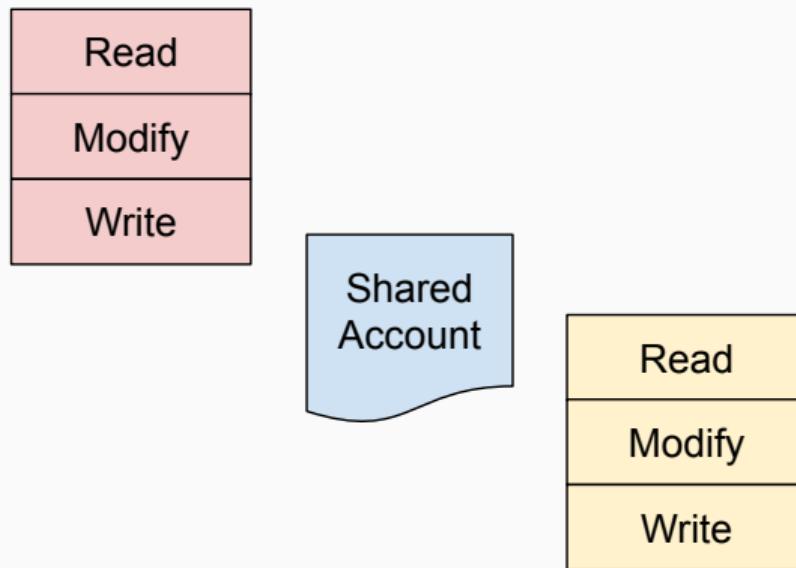
“System behavior depends on the (*non-deterministic*) sequence or timing of parts of the program executing independently, where some possible behaviors (*orders of execution*) produce invalid results”

What if interleaving parts of 2 and 3 is wrong?

1. {1, 2a, 2b, 3a, 3b, 4}
2. {1, 2a, 3a, 2b, 3b, 4}
3. {1, 2a, 3a, 3b, 2b, 4}
4. {1, 3a, 3b, 2a, 2b, 4}
5. {1, 3a, 2a, 3b, 2b, 4}

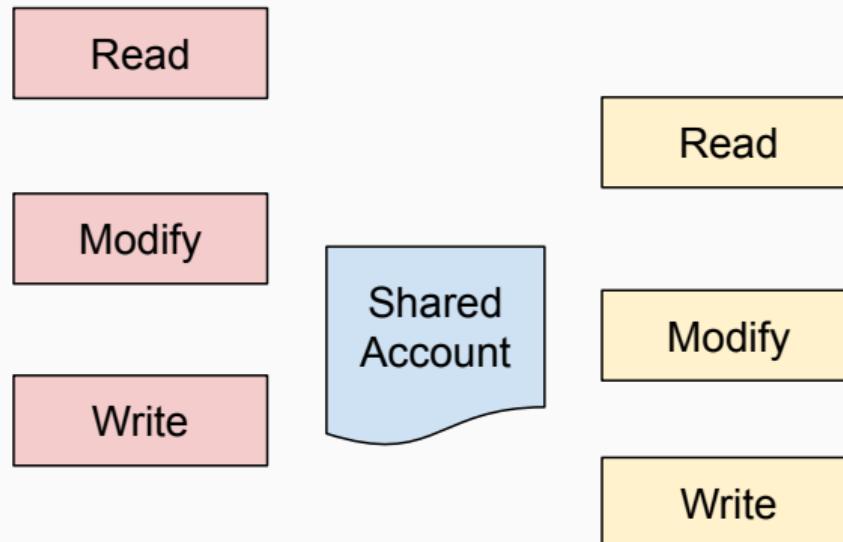
Race condition example

Two deposits to a bank account



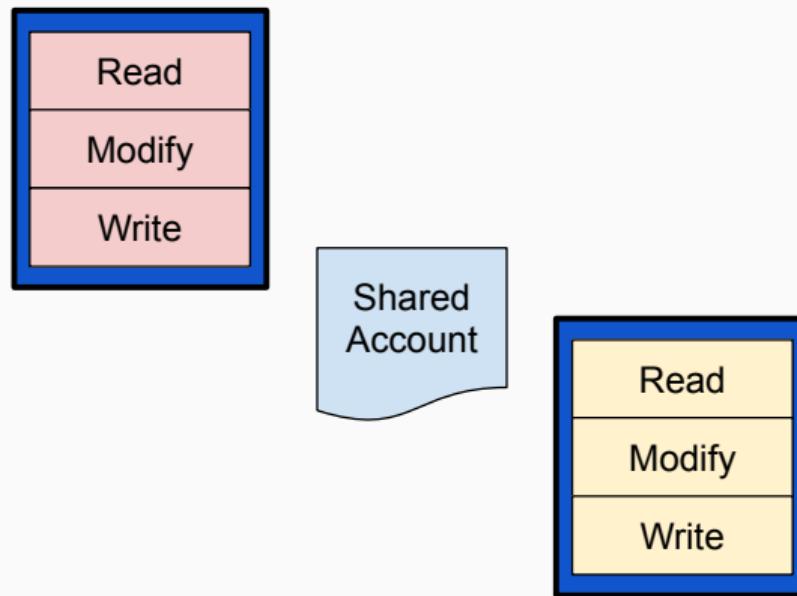
Race condition example fails

Parts of the two deposits are interleaved

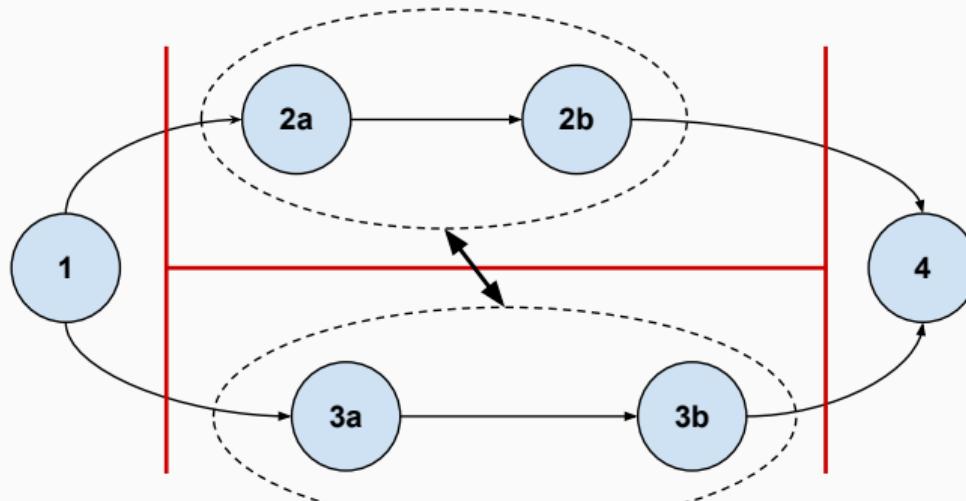


Race condition example fixed

We must actively prevent the parts interleaving



Race condition example fixed



Some ways to solve race conditions

Race conditions involve independent parts of the program changing things that are shared

Solutions making sure operations produce a consistent state to any shared data

- don't share anything
- make the shared things read-only
- allow only one writer to the shared things
- make the read-modify-write operations atomic

In the last case, we're adding more (sequential) order to our operations

Programming in Go

Matt Holiday
Christmas 2020



Share memory by communicating

Channels

A channel is a one-way communications pipe



- things go in one end, come out the other
- in the same order they went in
- until the channel is closed
- **multiple readers & writers can share it safely**

Sequential process

Looking at a single independent part of the program,
it appears to be sequential

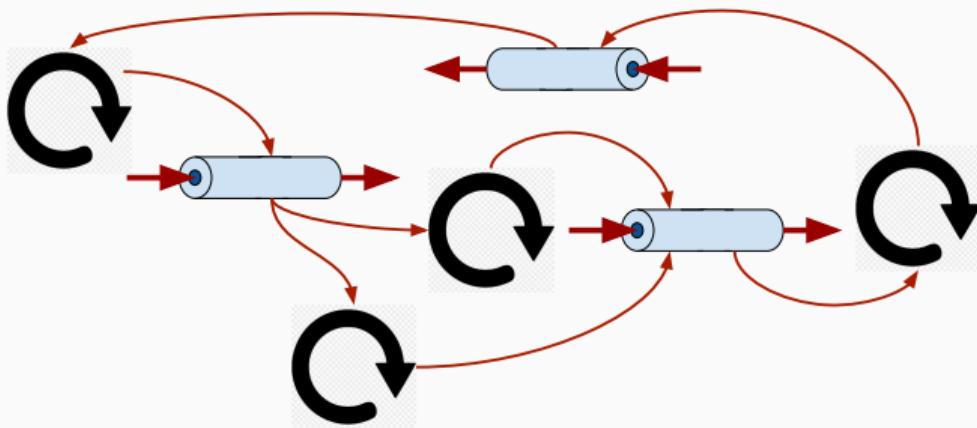
```
for {
    read()
    process()
    write()
}
```



This is perfectly natural if we think of reading & writing files or network sockets

Communicating sequential processes

Now put the parts together with channels to communicate



- each part is independent
- all they share are the channels between them
- **the parts can run in parallel as the hardware allows**

Communicating sequential processes

Concurrency is **always** hard
(the human brain didn't evolve for this, sorry :-)

CSP provides a model for thinking about it that makes it **less hard**
(take the program apart and make the pieces talk to each other)

“Go doesn’t force developers to embrace the asynchronous ways of event-driven programming. ... That lets you **write asynchronous code in a synchronous style**. As people, we’re much better suited to writing about things in a synchronous style.”

— Andrew Gerrand

Goroutines

A goroutine is a unit of **independent execution** (coroutine)

It's easy to start a goroutine: put `go` in front of a *function call*

The trick is knowing how the goroutine will stop:

- you have a well-defined loop terminating condition, or
- you signal completion through a channel or context, or
- you let it run until the program stops

But you need to make sure it doesn't get blocked by mistake

Channels

A channel is like a one-way socket or a Unix pipe
(except it allows multiple readers & writers)

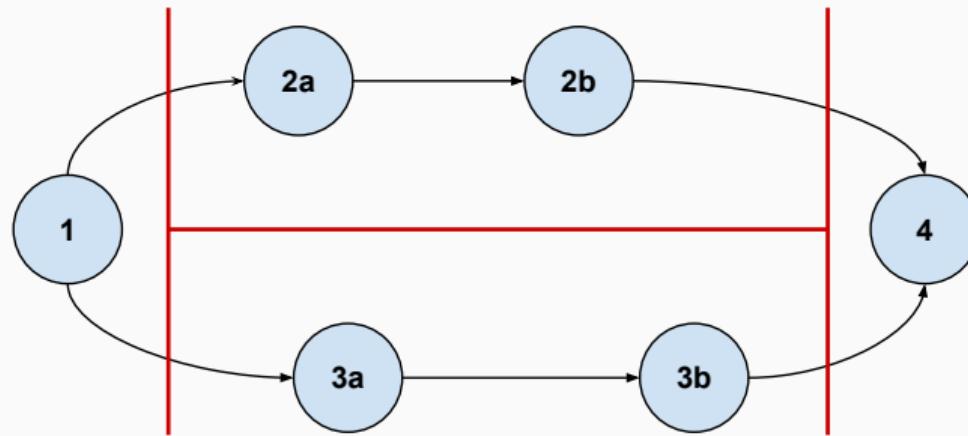
It's a method of synchronization as well as communication

We know that a send (write) always **happens before** a receive (read)

It's also a vehicle for *transferring ownership* of data, so that only one goroutine at a time is writing the data (avoid race conditions)

“Don’t communicate by sharing memory; instead, **share memory by communicating.**” — Rob Pike

Partial order



- part 1 **happens before** parts of 2 or 3
- both 2 and 3 **happen before** part 4
- the parts of 2 and 3 are ordered among themselves

Concurrency Example 1: Parallel Get

```
func main() {
    results := make(chan result) // channel for results
    list := []string{"https://amazon.com", "https://google.com",
                     "https://nytimes.com", "https://wsj.com",
    }
    for _, url := range list {
        go get(url, results) // start a CSP process
    }
    for range list { // read from the channel
        r := <-results
        if r.err != nil {
            log.Printf("%-20s %s\n", r.url, r.latency)
        } else {
            log.Printf("%-20s %s\n", r.url, r.err)
        }
    }
}
```

Concurrency Example 1: Parallel Get

```
type result struct {
    url      string
    err      error
    latency time.Duration
}

func get(url string, ch chan<- result) {
    start := time.Now()

    if resp, err := http.Get(url); err != nil {
        ch <- result{url, err, 0}      // error response
    } else {
        t := time.Since(start).Round(time.Millisecond)
        ch <- result{url, nil, t}      // normal response
        resp.Body.Close()
    }
}
```

Concurrency Example 2: Stream of IDs

```
var nextID = 0

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", nextID)

    // unsafe - data race

    nextID++
}

func main() {
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}

// simple HTTP server example from Francesc Campoy
```

Concurrency Example 2: Stream of IDs

```
var nextID = make(chan int)

func handler(w http.ResponseWriter, q *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v</h1>", <-nextID)
}

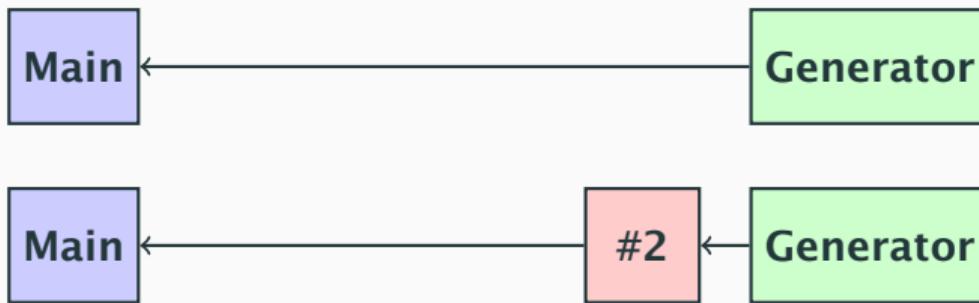
func counter() {
    for i := 0; ; i++ {
        nextID <- i
    }
}

func main() {
    go counter()
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

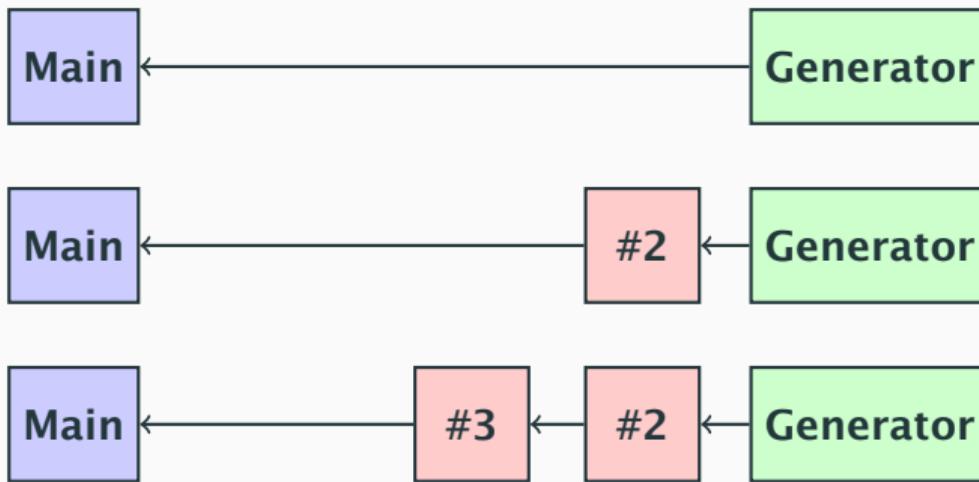
Concurrency Example 3: Prime Sieve



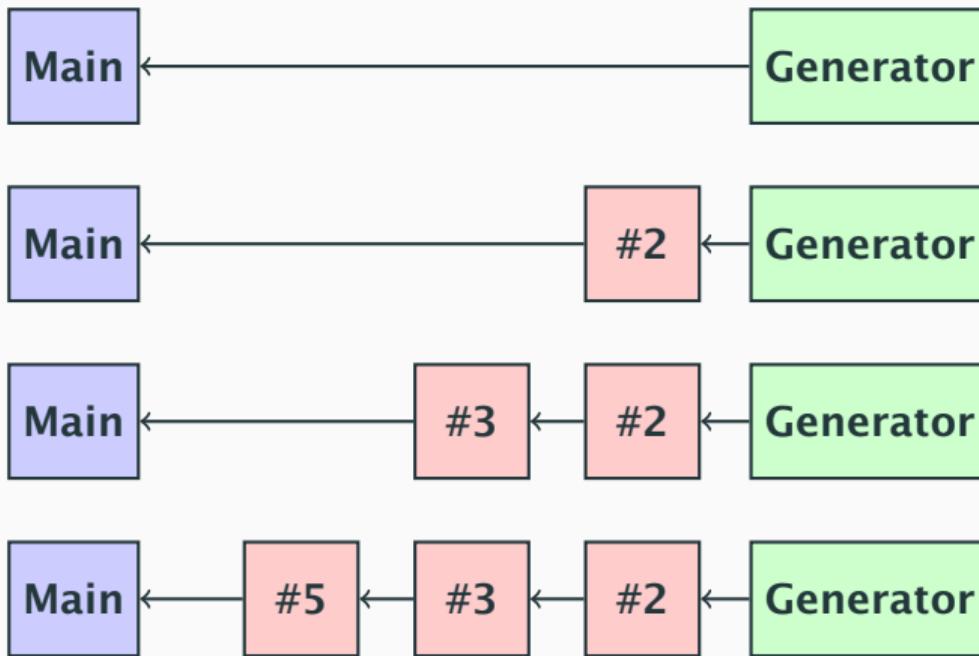
Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve



Concurrency Example 3: Prime Sieve

```
// Doug McIlroy (1968) via Tony Hoare (1978)
// code example from the Go language spec

func generate(limit int, ch chan<- int) {
    for i := 2; i < limit ; i++ {
        ch <- i
    }
    close(ch)
}

func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {
        if i % prime != 0 {
            dst <- i
        }
    }
    close(dst)
}
```

Concurrency Example 3: Prime Sieve

```
func sieve(limit int) {
    ch := make(chan int)
    go generate(limit, ch)
    for {
        prime, ok := <-ch
        if !ok {
            break
        }
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
        fmt.Println(prime, " ")
    }
}

func main() {
    sieve(100) // 2 3 5 7 11 13 17 19 23 29 31 37 41 43 ...
}
```

Programming in Go

Matt Holiday
Christmas 2020



Select

Select

`select` allows any “ready” alternative to proceed among

- a channel we can read from
- a channel we can write to
- a `default` action that’s always ready

Most often `select` runs in a loop so we keep trying

We can put a timeout or “done” channel into the `select`

- ♥ We can compose channels as synchronization primitives!
- ♠ Traditional primitives (mutex, condition variable) can’t be composed

Select example: read two channels

```
func main() {
    chans := []chan int{ make(chan int), make(chan int) }
    for i := range chans {
        go func(i int, ch chan int) {
            for {
                time.Sleep(time.Duration(i) * time.Second); ch <- i
            }
        }(i+1, chans[i])
    }
    for i := 0; i < 12; i++ {
        select {
        case m0 := <-chans[0]:
            fmt.Println("received", m0)
        case m1 := <-chans[1]:
            fmt.Println("received", m1)
        }
    }
}
```

Select example: read two channels

```
2009/11/10 23:00:01 received 1
2009/11/10 23:00:02 received 2
2009/11/10 23:00:02 received 1
2009/11/10 23:00:03 received 1
2009/11/10 23:00:04 received 2
2009/11/10 23:00:04 received 1
2009/11/10 23:00:05 received 1
2009/11/10 23:00:06 received 1
2009/11/10 23:00:06 received 2
2009/11/10 23:00:07 received 1
2009/11/10 23:00:08 received 1
2009/11/10 23:00:08 received 2
```

Simple example with timeout

```
func main() {
    stopper := time.After(5 * time.Second)
    . . .

    for _, url := range list {
        go get(url, results)           // start a CSP process
    }

    for range list {                  // read from the channel
        select {
        case r := <-results:
            log.Printf("%-20s %s\n", r.url, r.latency)
        case <-stopper:
            log.Fatal("timeout")
        }
    }
}
```

Select with a periodic timer

```
const tickRate = 2 * time.Second

func main() {
    log.Println("start")
    ticker := time.NewTicker(tickRate).C // periodic
    stopper := time.After(5 * tickRate) // one shot

loop:
    for {
        select {
        case <-ticker:
            log.Println("tick")
        case <-stopper:
            break loop
        }
    }
    log.Println("finish")
}
```

Select with a periodic timer

```
2009/11/10 23:00:00 start
2009/11/10 23:00:02 tick
2009/11/10 23:00:04 tick
2009/11/10 23:00:06 tick
2009/11/10 23:00:08 tick
2009/11/10 23:00:10 tick
2009/11/10 23:00:10 finish
```

Select example: default

In a `select` block, the `default` case is always ready and will be chosen if no other case is:

```
func sendOrDrop(data []byte) {
    select {
        case ch <- data;
            // sent ok; do nothing

        default:
            log.Printf("overflow: drop %d bytes", len(data))
    }
}
```

Don't use `default` inside a loop — the `select` will busy wait and waste CPU

Select example: running a subprocess

```
func start(name, args ...string) error {
    cmd := exec.Command(name, args...)
    stderr, _ := cmd.StderrPipe()           // ignoring errors
    _ := cmd.Start()                      // ditto :-
    done := make(chan string)

    go scrape(stderr, done)

    select {
    case result := <-done:
        if result == "running" {
            return nil
        }
        return fmt.Errorf("failed: %s", result)
    case <-time.After(20 * time.Second):
        return fmt.Errorf("timeout")
    }
}
```

Select example: running a subprocess

```
func scrape(in io.ReadCloser, done ch<- string) {
    defer in.Close()
    defer close(done)

    for {
        d := bufferedRead(in)    // details omitted

        if strings.Contains(d, "Exception") {
            done <- strings.TrimSuffix(d, "\n")
            return
        }

        if strings.Contains(d, "server is now running") {
            done <- "running"
            return
        }
    }
}
```

Programming in Go

Matt Holiday
Christmas 2020



Context

Cancellation and timeouts

The Context package offers a common method to cancel requests

- explicit cancellation
- implicit cancellation based on a timeout or deadline

A context may also carry request-specific values, such as a trace ID

Many network or database requests, for example, take a context for cancellation

Cancellation and timeouts

A context offers two controls:

- a channel that closes when the cancellation occurs
- an error that's readable once the channel closes

The error value tells you whether the request was cancelled or timed out

We often use the channel from `Done()` in a `select` block

Cancellation and timeouts

Contexts form an **immutable** tree structure
(goroutine-safe; changes to a context do not affect its ancestors)

Cancellation or timeout applies to the current context and its **subtree**

Ditto for a value

A subtree may be created with a shorter timeout (but not longer)

Context as a tree structure

It's a tree of **immutable** nodes which can be extended



Context example

The Context value should always be the first parameter

```
// First runs a set of queries and returns the result from the
// the first to respond, canceling the others.
func First(ctx context.Context, urls []string) (*Result, error) {
    c := make(chanResult, len(urls))           // buffered to avoid orphans
    ctx, cancel := context.WithCancel(ctx)

    defer cancel() // cancel the other queries when we're done

    search := func(url string) {
        c <- runQuery(ctx, url)
    }

    . . .
}
```

Context example

Don't call Err() until the channel from Done() closes

```
    . . .
for _, url := range urls {
    go search(url)
}

select {
case r := <- c:
    return &r, nil
case <-ctx.Done():
    return nil, ctx.Err()
}
}
```

Values

Context values should be data specific to a request, such as:

- a trace ID or start time (for latency calculation)
- security or authorization data

Avoid using the context to carry “optional” parameters

Use a package-specific, private context key type (not string) to avoid collisions

Value example

```
type contextKey int

const TraceKey contextKey = 1

// AddTrace is HTTP middleware to insert a trace ID into the request.
func AddTrace(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        if traceID := r.Header.Get("X-Cloud-Trace-Context"); traceID != "" {
            ctx = context.WithValue(ctx, TraceKey, traceID)
        }

        next.ServeHTTP(w, r.WithContext(ctx))
    })
}
```

Value example

```
type contextKey int

const TraceKey contextKey = 1

// ContextLog makes a log with the trace ID as a prefix.
func ContextLog(ctx context.Context, f string, args ...interface{}) {
    // reflection -- to be discussed

    traceID, ok := ctx.Value(TraceKey).(string)

    if ok && traceID != "" {
        f = traceID + ": " + f
    }

    log.Printf(f, args...)
}
```

Parallel get with timeout

```
func main() {
    results := make(chan result) // channel for results
    list := []string{"https://amazon.com", . . .}
    ctx, cancel := context.WithTimeout(context.Background, 3*time.Second)
    defer cancel()
    for _, url := range list {
        go get(ctx, url, results) // start a CSP process
    }
    for range list {           // read from the channel
        r := <-results
        if r.err != nil {
            log.Printf("%-20s %s\n", r.url, r.latency)
        } else {
            log.Printf("%-20s %s\n", r.url, r.err)
        }
    }
}
```

Parallel get with timeout

```
func get(ctx context.Context, url string, ch chan<- result) {
    start := time.Now()
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)

    if err != nil {
        ch <- result{url, err, 0}      // error response
        return
    }

    if resp, err := http.DefaultClient.Do(req); err != nil {
        ch <- result{url, err, 0}      // error response
    } else {
        t := time.Since(start).Round(time.Millisecond)
        ch <- result{url, nil, t}      // normal response
        resp.Body.Close()
    }
}
```

Server with variable delay

```
package main

import ("flag"; "fmt"; "log"; "net/http"; "time")

var delay int

func handler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(time.Duration(delay) * time.Second)
    fmt.Fprintf(w, "hello")
}

func main() {
    flag.IntVar(&delay, "delay", 60, "delay all responses")
    flag.Parse()
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8081", nil))
}
```

Programming in Go

Matt Holiday
Christmas 2020



Channels & Synchronization

Channel state

Channels **block** unless ready to read or write

A channel is ready to write if

- it has buffer space, or
- at least one reader is ready to read (rendezvous)

A channel is ready to read if

- it has unread data in its buffer, or
- at least one writer is ready to write (rendezvous), or
- it is closed

Channel state

Channels are unidirectional, but have two ends
(which can be passed separately as parameters)

- An end for writing & closing

```
func get(url string, ch chan<- result) {           // write-only end
    ...
}
```

- An end for reading

```
func collect(ch <-chan result) map[string]int { // read-only end
    ...
}
```

Closed channels

Closing a channel causes it to return the “zero” value

We can receive a second value: *is the channel closed?*

```
func main() {
    ch := make(chan int, 1)

    ch <- 1

    b, ok := <-ch
    fmt.Println(b, ok)      // 1 true

    close(ch)

    c, ok := <-ch
    fmt.Println(c, ok)      // 0 false
}
```

Closed channels

A channel can only be closed once (else it will panic)

One of the main issues in working with goroutines is **ending** them

- An unbuffered channel requires a reader and writer
(a writer blocked on a channel with no reader will “leak”)
- Closing a channel is often a **signal** that work is done
- Only **one** goroutine can close a channel (not many)
- We may need some way to coordinate closing a channel or stopping goroutines
(beyond the channel itself)

Nil channels

Reading or writing a channel that is `nil` always blocks *

But a `nil` channel in a `select` block is *ignored*

This can be a powerful tool:

- Use a channel to get input
- Suspend it by changing the channel variable to `nil`
- You can even un-suspend it again
- But **close** the channel if there really is no more input (EOF)

Channel state reference

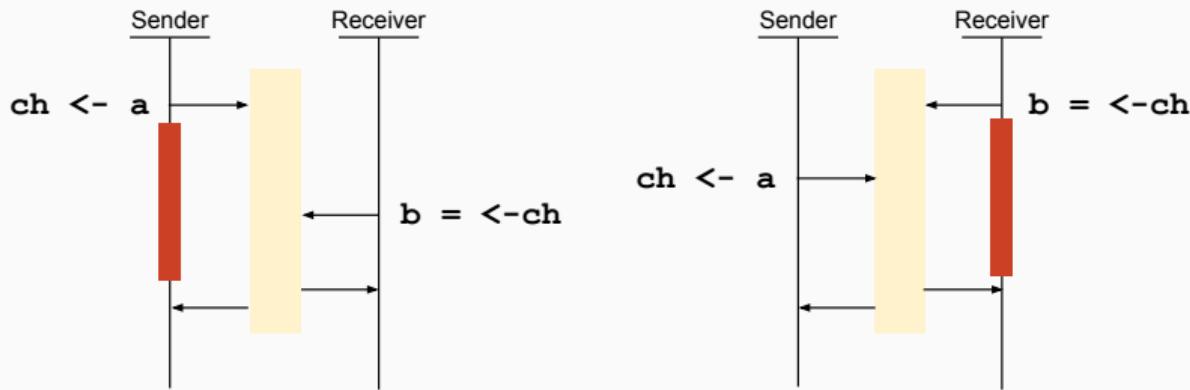
State	Receive	Send	Close
Nil	Block*	Block*	Panic
Empty	Block	Write	Close
Partly Full	Read	Write	Readable until empty
Full	Read	Block	
Closed	Default Value**		Panic
Receive-only	OK		Compile Error
Send-only	Compile Error		OK

* `select` ignores a nil channel since it would always block

** Reading a closed channel returns (`<default-value>`, `!ok`)

Rendezvous

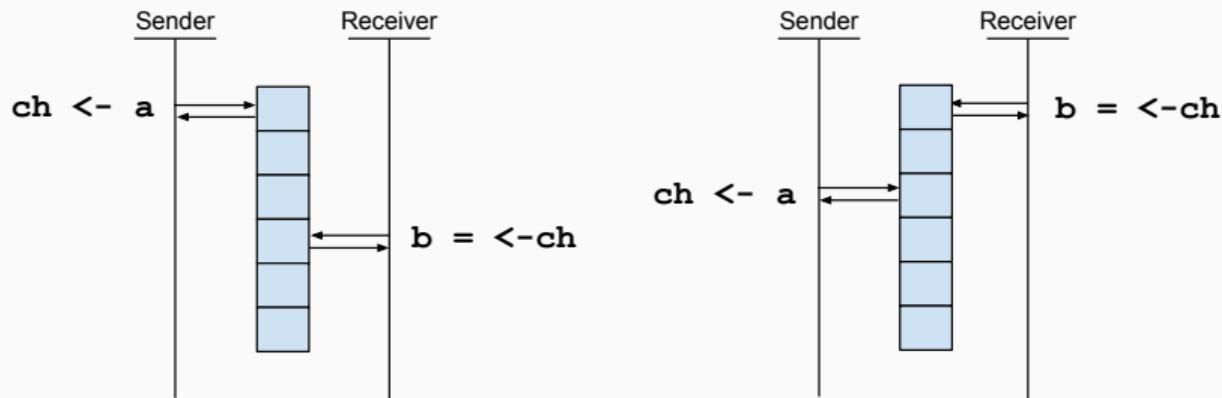
By default, channels are unbuffered (rendezvous model)



- the sender blocks until the receiver is ready (and vice versa)
- the send always happens before the receive
- the receive always *returns* before the send
- **the sender & receiver are synchronized**

Buffering

Buffering allows the sender to send without waiting



- the sender deposits its item and returns immediately
- the sender blocks only if the buffer is full
- the receiver blocks only if the buffer is empty
- **the sender & receiver run independently**

Buffering

Buffering allows the sender to send without waiting

```
func main() {
    // make a channel with buffer that holds two items
    messages := make(chan string, 2)

    // now we can send twice without getting blocked
    messages <- "buffered"
    messages <- "channel"

    // and then receive both as usual
    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

With buffer size 1 (or no buffer at all), it will **deadlock!**

Rendezvous vs Buffering

```
package main

import (
    "fmt"
    "time"
)

type T struct {
    i byte
    t bool
}

func send(i int, ch chan<- *T) {
    t := &T{i: byte(i)}
    ch <- t
    t.t = true // UNSAFE AT ANY SPEED
}
```

Rendezvous vs Buffering

```
func main() {
    vs, ch := make([]T, 5), make(chan *T) // change to [5]

    for i := range vs {
        go send(i, ch)
    }

    time.Sleep(1*time.Second)           // all goroutines started

    for i := range vs {                // copy quickly!
        vs[i] = *<-ch
    }

    for _, v := range vs {            // print later
        fmt.Println(v)
    }
}
```

Buffering

Common uses of buffered channels:

- avoid goroutine leaks (from an abandoned channel)
- avoid rendezvous pauses (performance improvement)

Don't buffer until it's needed: *buffering may hide a race condition*

Some testing may be required to find the right number of slots!

Special uses of buffered channels:

- counting semaphore pattern

Counting semaphores

A **counting semaphore** limits work in progress (or occupancy)

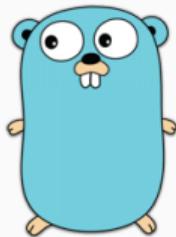
Once it's "full" only one unit of work can enter for each one that leaves

We model this with a buffered channel:

- attempt to send (write) before starting work
- the send will block if the buffer is full (occupancy is at max)
- receive (read) when the work is done to free up a space in the buffer
(this allows the next worker to start)

Programming in Go

Matt Holiday
Christmas 2020



File Walk Example

What's the problem?

I want to find duplicate files based on their **content**

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

f088913 2

/Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
/Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

f088913 2

/Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
/Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf

It takes nearly **5 minutes** to comb through my Dropbox folder
(2017 core i7 quad-core MacBook Pro)

Code at <https://github.com/matt4biz/go-class-walk>

Sequential Approach

How it works: Declarations

```
package main

import (
    "crypto/md5"
    "fmt"
    "io"
    "log"
    "os"
    "path/filepath"
)

type pair struct {
    hash, path string
}

type fileList []string
type results map[string]fileList
```

How it works: Hashing

```
func hashFile(path string) pair {
    file, err := os.Open(path)

    if err != nil {
        log.Fatal(err)
    }

    defer file.Close()

    hash := md5.New() // fast & good enough

    if _, err := io.Copy(hash, file); err != nil {
        log.Fatal(err)
    }

    return pair{fmt.Sprintf("%x", hash.Sum(nil)), path}
}
```

How it works: Searching

```
func searchTree(dir string) (results, error) {
    hashes := make(results)

    err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                    err error) error {
        // ignore the error parm for now

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            h := hashFile(p)
            hashes[h.hash] = append(hashes[h.hash], h.path)
        }

        return nil
    })

    return hashes, err
}
```

How it works: Output

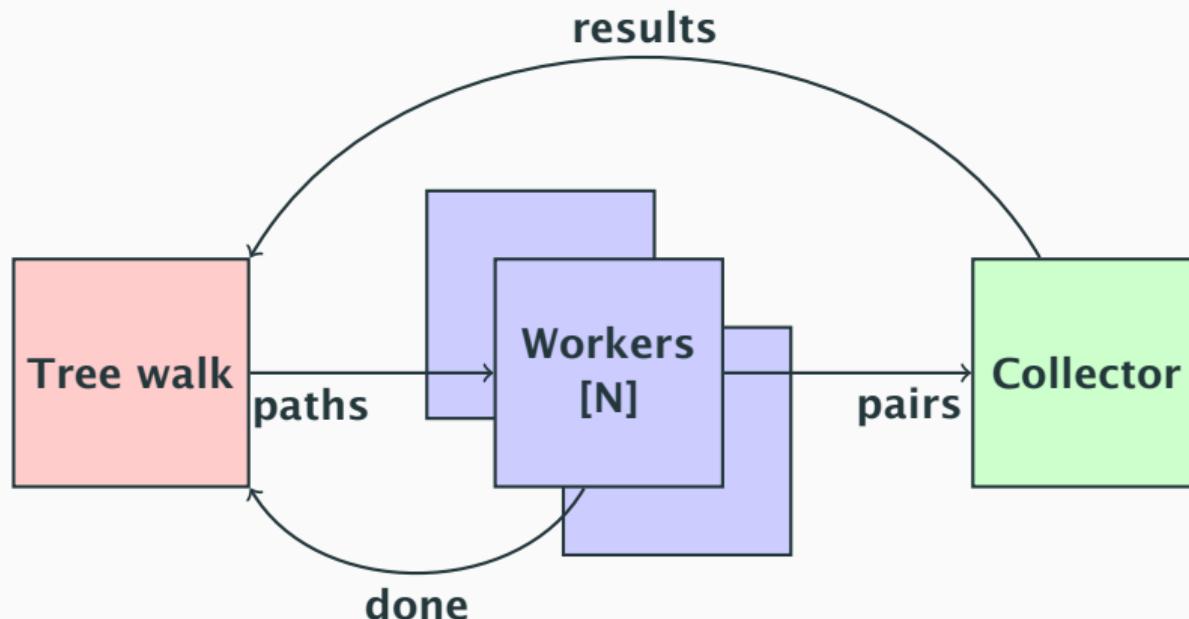
```
func main() {
    if len(os.Args) < 2 {
        log.Fatal("Missing parameter, provide dir name!")
    }
    if hashes, err := searchTree(os.Args[1]); err == nil {
        for hash, files := range hashes {
            if len(files) > 1 {
                // we will use just 7 chars like git
                fmt.Println(hash[len(hash)-7:], len(files))

                    for _, file := range files {
                        fmt.Println("  ", file)
                    }
            }
        }
    }
}
```

Concurrent Approach #1

A concurrent approach (like map-reduce)

Use a fixed pool of goroutines and a collector and channels



How it works: Collecting the hashes

```
func collectHashes(pairs <-chan pair, result chan<- results) {
    hashes := make(results)

    for p := range pairs {
        hashes[p.hash] = append(hashes[p.hash], p.path)
    }

    result <- hashes
}
```

How it works: Replacing the processor

```
func processFiles(paths <-chan string, pairs chan<- pair,  
                  done chan<- bool) {  
    for path := range paths {  
        pairs <- hashFile(path)  
    }  
  
    done <- true  
}
```

How it works: Replacing the tree walk

```
workers := 2 * runtime.GOMAXPROCS(0)

paths := make(chan string)
pairs := make(chan pair)
done := make(chan bool)
result := make(chan results)

for i := 0; i < workers; i++ {
    go processFiles(paths, pairs, done)
}

// we need another goroutine so we don't block here

go collectHashes(pairs, result)

...
```

How it works: Replacing the tree walk

```
err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                err error) error {
    // again, ignore the error passed in

    if fi.Mode().IsRegular() && fi.Size() > 0 {
        paths <- p
    }

    return nil
})

if err != nil {
    log.Fatal(err)
}

// we must close the paths channel so the workers stop
close(paths)
. . .
```

How it works: Replacing the tree walk

```
 . . .

// wait for all the workers to be done

for i := 0; i < workers; i++ {
    <-done
}

// by closing pairs we signal that all the hashes
// have been collected; we have to do it here AFTER
// all the workers are done

close(pairs)

hashes := <-result

return hashes
```

Evaluation #1

56.11s in the version shown above

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs` (buffering `pairs` keeps the workers working)

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs` (buffering `pairs` keeps the workers working)

51.36s with twice as many workers (32)

Concurrent Approach #2

Another concurrent approach

Add a goroutine for each directory in the tree

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

How it works: Parallel tree walk

```
 . . .
wg := new(sync.WaitGroup)

// multi-threaded walk of the directory tree; we need a
// waitGroup because we don't know how many to wait for

wg.Add(1)
err := walkDir(dir, paths, wg)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(paths)

. . .
```

How it works: Parallel tree walk

```
func walkDir(dir string, paths chan<- string,
            wg *sync.WaitGroup) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        // ignore dir itself to avoid an infinite loop!
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, paths)
            return filepath.SkipDir
        }

        .
        .
        .
    }
}
```

How it works: Parallel tree walk

```
    . . .

    if fi.Mode().IsRegular() && fi.Size() > 0 {
        paths <- p
    }

    return nil
}

return filepath.Walk(dir, visit)
}
```

Evaluation #2

51.14s in the basic version

Evaluation #2

51.14s in the basic version

50.03 adding buffers on all channels to/from workers

Evaluation #2

51.14s in the basic version

50.03 adding buffers on all channels to/from workers

48.75 with twice as many workers

Concurrent Approach #3

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

Without some controls, we'll run out of threads!

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

What could go wrong?

Without some controls, we'll run out of threads!

GOMAXPROCS doesn't limit threads blocked on syscalls (all our disk I/O)

We'll limit the number of **active** goroutines instead (the ones making syscalls)

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking
(with no intervening reads)

Channels as counting semaphores

A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking
(with no intervening reads)

The buffer provides a fixed upper bound (unlike a WaitGroup)

Channels as counting semaphores

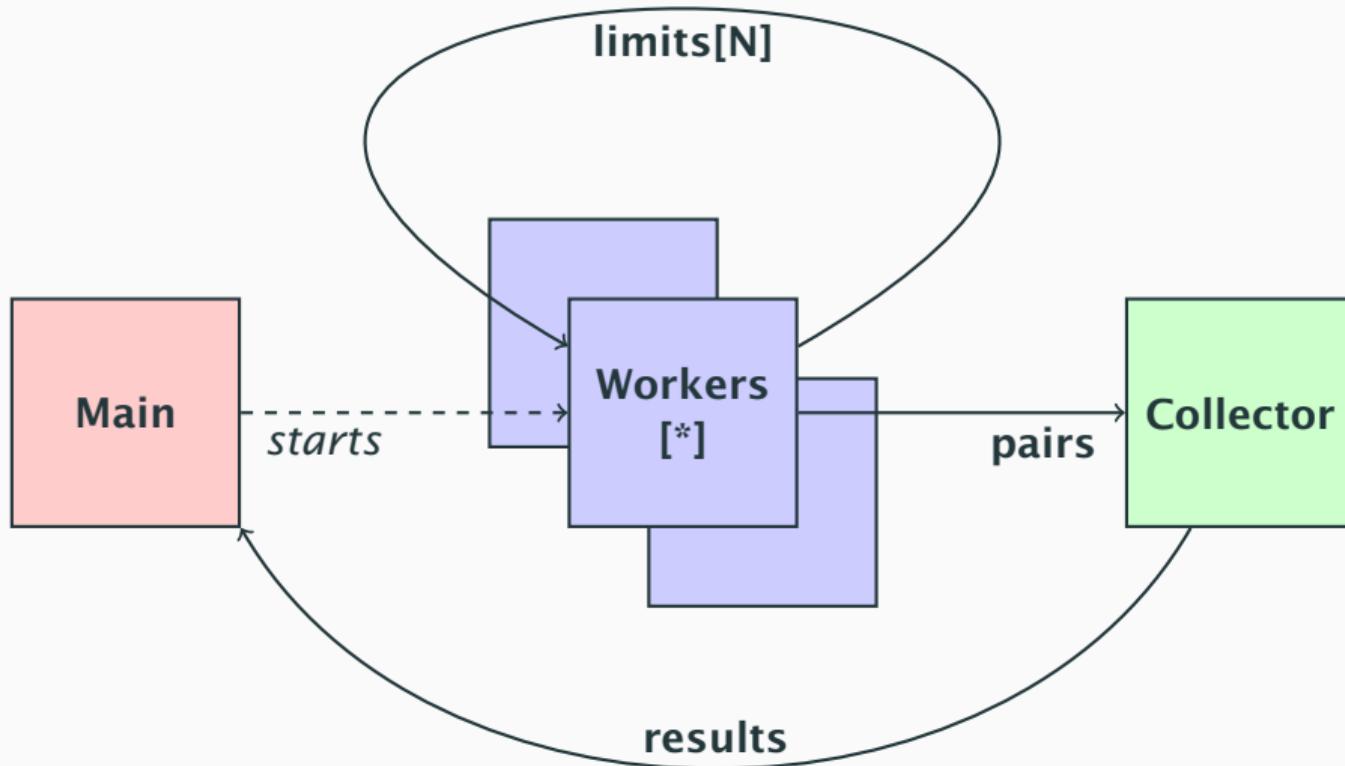
A goroutine can't proceed without *sending* on the channel

A channel with buffer size N can accept N sends without blocking
(with no intervening reads)

The buffer provides a fixed upper bound (unlike a WaitGroup)

One goroutine can start for each one that finishes
(each *reads* from the channel when done)

What that looks like



How it works: Limiting goroutines

```
// we don't need a channel for paths or to signal done but
// we need a buffered channel to act as a counting semaphore

wg := new(sync.WaitGroup)
limits := make(chan bool, workers)
pairs := make(chan pair, workers)
result := make(chan results)

go collect(pairs, result)

...
```

How it works: Limiting goroutines

```
.

.

wg.Add(1)
err := walkDir(dir, pairs, wg, limits)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(pairs)

hashes := <-result

return hashes
```

How it works: Modified processing

```
func processFile(path string, pairs chan<- pair,  
                 wg *sync.WaitGroup, limits chan bool) {  
    defer wg.Done()  
  
    limits <- true  
  
    defer func() {  
        <-limits  
    }()  
  
    pairs <- hashFile(path)  
}
```

How it works: Modified tree walk

```
func walkDir(dir string, pairs chan<- pair, wg *sync.WaitGroup,
            limits chan bool) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, pairs, wg, limits)
            return filepath.SkipDir
        }

        . . .
    }
}
```

How it works: Modified tree walk

```
    . . .

    if fi.Mode().IsRegular() && fi.Size() > 0 {
        wg.Add(1)
        go processFile(p, pairs, wg, limits)
    }

    return nil
}

limits <- true

defer func() {
    <-limits
}()

return filepath.Walk(dir, visit)
}
```

Evaluation #3

46.93s using 32 workers was the best time

Evaluation #3

46.93s using 32 workers was the best time

Increasing the `limits` buffer makes the time grow longer due to disk contention

Evaluation #3

46.93s using 32 workers was the best time

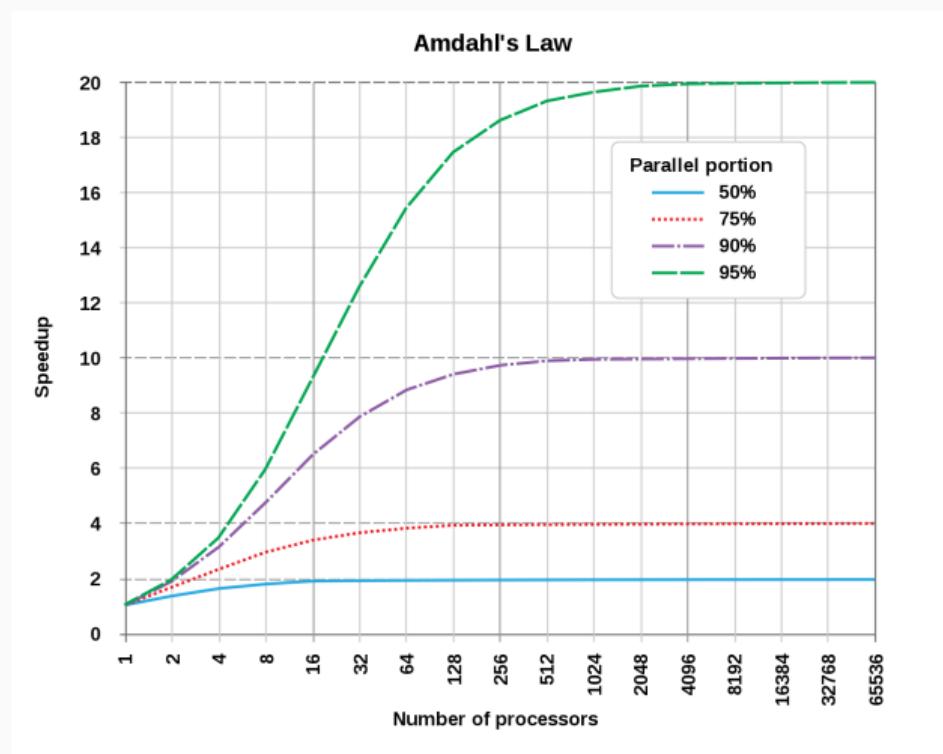
Increasing the `limits` buffer makes the time grow longer due to disk contention

Amdahl's law: speedup is limited by the part (not) parallelized

$$S = \frac{1}{1 - p + (p/s)}$$

Here we've managed about $S = 6.25$ on $s = 8$ processors, or about $p = 96\%$ parallel

Evaluation #3



Conclusions

We don't need to limit *goroutines*

We need to limit *contention* for shared resources

Programming in Go

Matt Holiday
Christmas 2020



Conventional Synchronization

Conventional synchronization

Package `sync`, for example:

- `Mutex`
- `Once`
- `Pool`
- `RWMutex`
- `WaitGroup`

Package `sync/atomic` for atomic scalar reads & writes

We saw a use of `WaitGroup` in the “file walk” example

Mutual exclusion

What if multiple goroutines must read & write some data?

We must make sure only **one** of them can do so at any instant
(in the so-called “critical section”)

We accomplish this with some type of lock:

- acquire the lock before accessing the data
- any other goroutine will **block** waiting to get the lock
- release the lock when done

Mutexes in action

```
type SafeMap struct {
    sync.Mutex          // not safe to copy
    m map[string]int
}

// so methods must take a pointer, not a value
func (s *SafeMap) Incr(key string) {
    s.Lock()
    defer s.Unlock()

    // only one goroutine can execute this
    // code at the same time, guaranteed
    s.m[key]++
}
```

Using `defer` is a good habit — avoid mistakes

RWMutexes in action

Sometimes we need to prefer readers to (infrequent) writers

```
type InfoClient struct {
    mu      sync.RWMutex
    token   string
    tokenTime time.Time
    TTL     time.Duration
}

func (i *InfoClient) CheckToken() (string, time.Duration) {
    i.mu.RLock()
    defer i.mu.RUnlock()

    return i.token, i.TTL - time.Since(i.tokenTime)
}
```

RWMutexes in action

```
func (i *InfoClient) ReplaceToken(ctx context.Context) (string, error) {
    token, ttl, err := i.getAccessToken(ctx)

    if err != nil {
        return "", err
    }

    i.mu.Lock()

    defer i.mu.Unlock()

    i.token = token
    i.tokenTime = time.Now()
    i.TTL = time.Duration(ttl) * time.Second

    return token, nil
}
```

Atomic primitives

```
func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            n++                         // DATA RACE
            w.Done()
        }()
    }

    w.Wait()
    return int(n)
}
```

Atomic primitives

```
func do() int {
    var n int64
    var w sync.WaitGroup

    for i := 0; i < 1000; i++ {
        w.Add(1)

        go func() {
            atomic.AddInt64(&n, 1) // fixed
            w.Done()
        }()
    }

    w.Wait()
    return int(n)
}
```

Only-once execution

A `sync.Once` object allows us to ensure a function runs only once
(only the first call to `Do` will call the function passed in)

```
var once sync.Once
var x    *singleton

func initialize() {
    x = NewSingleton()
}

func handle(w http.ResponseWriter, r *http.Request) {
    once.Do(initialize)
    . .
}

}
```

Checking `x == nil` in the handler is **unsafe!**

Pool

A Pool provides for efficient & safe reuse of objects, but it's a container of interface{}

```
var bufPool = sync.Pool{
    New: func() interface{} {
        return new(bytes.Buffer)
    },
}

func Log(w io.Writer, key, val string) {
    b := bufPool.Get().(*bytes.Buffer)      // more reflection
    b.Reset()
    // write to it
    w.Write(b.Bytes())
    bufPool.Put(b)
}
```

Wait, there's more!

Other primitives:

- Condition variable
- Map (safe container; uses `interface{}`)
- WaitGroup

Programming in Go

Matt Holiday
Christmas 2020



Homework

Homework #5

Exercise 7.11 from *GOPL*: web front-end for a database

Except now we will

- Write a program to generate traffic against our first solution
- Show running the server with `-race`
- Solve the race conditions

See my solution at: <https://github.com/matt4biz/go-class-exer-7.11>

Homework #5

1. Change the DB type

```
type database struct {
    mu sync.Mutex
    db map[string]int
}
```

2. Use a pointer receiver in all methods
3. Lock the mutex (and defer unlock) in all methods

```
package main

import ("fmt"; "net/http"; "os"; "testing"; "time")

type sku struct {
    item  string
    price string
}

var items = []sku{
    {"shoes", "46"}, {"socks", "6"}, {"sandals", "27"}, {"clogs", "36"}, {"pants", "30"}, {"shorts", "20"}, }
}
```

```
func doQuery(cmd, parms string) {
    resp, err := http.Get("http://localhost:8080/" + cmd + "?" + parms)

    if err == nil {
        defer resp.Body.Close()
        fmt.Fprintf(os.Stderr, "got %s = %d (no err)\n", parms, resp.StatusCode)
    } else if resp != nil {
        defer resp.Body.Close()
        fmt.Fprintf(os.Stderr, "got %s = %d (%v)\n", parms, resp.StatusCode, err)
    } else {
        fmt.Fprintf(os.Stderr, "got err %v\n", err)
    }
}
```

```
func runAdds() {
    for {
        for _, s := range items {
            doQuery("create", "item="+s.item+"&price="+s.price)
        }
    }
}

func runUpdates() {
    for {
        for _, s := range items {
            doQuery("update", "item="+s.item+"&price="+s.price)
        }
    }
}
```

```
func runDrops() {
    for {
        for _, s := range items {
            doQuery("create", "item="+s.item)
        }
    }
}

func TestServer(t *testing.T) {
    go runServer()      // code from old main
    go runAdds()
    go runDrops()
    go runUpdates()

    time.Sleep(5 * time.Second)
}
```

Programming in Go

Matt Holiday
Christmas 2020



Concurrency Gotchas

Concurrency problems

#1: race conditions, where unprotected read & writes overlap

- must be some data that is written to
- could be a read-modify-write operation
- and two goroutines can do it at the same time

#2: deadlock, when no goroutine can make progress

- goroutines could all be blocked on empty channels
- goroutines could all be blocked waiting on a mutex
- GC could be prevented from running (busy loop)

Go detects *some* deadlocks automatically; with `-race` it can find *some* data races

Concurrency problems

#3: goroutine leak

- goroutine hangs on a empty or blocked channel
- not deadlock: other goroutines make progress
- often found by looking at pprof output

When you start a goroutine, **always know how/when it will end**

#4: channel errors

- trying to send on a closed channel
- trying to send or receive on a nil channel
- closing a nil channel
- closing a channel twice

Concurrency problems

#5: other errors

- closure capture
- misuse of `Mutex`
- misuse of `WaitGroup`
- misuse of `select`

A good taxonomy of Go concurrency errors may be found in this paper:
<https://cseweb.ucsd.edu/~yiying/GoStudy-ASPLOS19.pdf>

Many of the errors are basic & should easily be found by review;
maybe we'll get static analysis tools to help find them

Gotchas 1: Data race

We've already seen this, but here it is again

```
var nextID = 0

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "<h1>You got %v<h1>", nextID)

    // unsafe - data race
    nextID++
}

func main() {
    http.HandleFunc("/", handler)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

Gotchas 2: Deadlock

Go can usually detect when no goroutine is able to make progress; here the main goroutine is blocked on a channel it can never read

```
func main() {
    ch := make(chan bool)

    go func(ok bool) {
        fmt.Println("STARTED")

        if ok {
            ch <- ok
        }
    }(false)

    <-ch
    fmt.Println("DONE")
}
```

Gotchas 2: Deadlock

Locking a mutex and then failing to unlock it afterwards;
the fix is to use defer at the point of locking

```
var m sync.Mutex
done := make(chan bool)

go func() {
    m.Lock() // not unlocked!
}()

go func() {
    time.Sleep(1)
    m.Lock()
    defer m.Unlock()
    done <- true
}()
<-done
```

Gotchas 2: Deadlock

Locking mutexes in the wrong order will often result in deadlock;
the fix is **always** to lock them in the same order everywhere

```
var m1, m2 sync.Mutex
done := make(chan bool)
go func() {
    m1.Lock(); defer m1.Unlock()
    time.Sleep(1)
    m2.Lock(); defer m2.Unlock()
    done <- true
}()
go func() {
    m2.Lock(); defer m2.Unlock()
    time.Sleep(1)
    m1.Lock(); defer m1.Unlock()
    done <- true
}()
<-done; <-done
```

Gotchas 3: Goroutine leak

In this example, a timeout leaves the goroutine hanging forever; the correct solution is to make a buffered channel

```
func finishReq(timeout time.Duration) *obj {
    ch := make(chan obj)

    go func() {
        . . .          // work that takes too long
        ch <- fn()   // blocking send
    }()
}

select {
case rslt := <-ch:
    return rslt
case <-time.After(timeout):
    return nil
}
}
```

Gotchas 4: Incorrect use of WaitGroup

Always, always, always call Add before go or Wait

```
func walkDir(dir string, pairs chan<- pair, ...) {
    wg.Add(1)                                // BIG MISTAKE
    defer wg.Done()

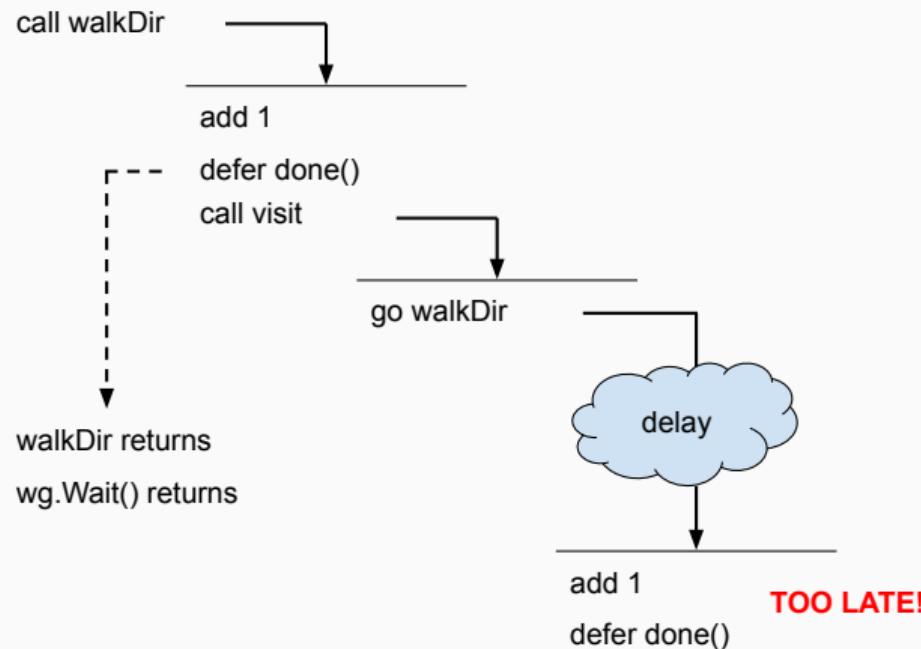
    visit := func(p string, fi os.FileInfo, ...) {
        if fi.Mode().IsDir() && p != dir {
            go walkDir(p, pairs, wg, limits)
        }
    }

    err := walkDir(dir, paths, wg)
    wg.Wait()
```

Adding too late may cause Wait to return too soon

Gotchas 4: Incorrect use of WaitGroup

Adding *inside* the goroutine may fail (too late)



Gotchas 4: Incorrect use of WaitGroup

Always, always, always call Add before go or Wait

```
func walkDir(dir string, pairs chan<- pair, ...) {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, ...) {
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)                                // RIGHT
            go walkDir(p, pairs, wg, limits)
        }
    }

    wg.Add(1)                                // RIGHT
    err := walkDir(dir, paths, wg)

    wg.Wait()
```

Gotchas 5: Closure capture

A goroutine closure shouldn't capture a **mutating** variable

```
for i := 0; i < 10; i++ {    // WRONG
    go func() {
        fmt.Println(i)
    }()
}
```

Instead, **pass the variable's value as a parameter**

```
for i := 0; i < 10; i++ {    // RIGHT
    go func(i int) {
        fmt.Println(i)
    }(i)
}
```

Select problems

`select` can be challenging and lead to mistakes:

- `default` is always active
- a `nil` channel is always ignored
- a full channel (for send) is skipped over
- a “done” channel is *just another channel*
- **available channels are selected at random**

Anatomy of a select mistake: #1

Mistake #1: skipping a full channel to default and losing a message

```
for {
    x := socket.Read()

    select {
        case output <- x:
            . . .

        default:
            return
    }
}
```

The code was written assuming we'd skip `output` only if it was set to `nil`

We also skip if `output` is full, and lose this and future messages

Anatomy of a select mistake: #2

Mistake #2: reading a “done” channel and aborting when input is backed up on another channel — that input is lost

```
for {
    select {
        case x := <- input:
            . . .

        case <- done:
            return
    }
}
```

There's no guarantee we read all of `input` before reading `done`

Better: use `done` only for an error abort; close `input` on EOF

Some thoughts

Four considerations when using concurrency:

1. Don't start a goroutine without knowing how it will stop
2. Acquire locks/semaphores as late as possible; release them in the reverse order
3. Don't wait for non-parallel work that you could do yourself

```
func do() int {  
    ch := make(chan int)  
  
    go func() { ch <- 1 }()  
  
    return <-ch  
}
```

4. Simplify! Review! Test!

Programming in Go

Matt Holiday
Christmas 2020



Odds and Ends

Enumerated types

There are no real enumerated types in Go

You can make an almost-enum type using a named type and constants:

```
type shoe int

const (
    tennis shoe = iota
    dress
    sandal
    clog
)
```

`iota` starts at 0 in each `const` block and increments once on each line;
here 0, 1, 2, ...

Enumerated types

Traditional flags are easy:

```
type Flags uint

const (
    FlagUp Flags = 1 << iota // is up
    FlagBroadcast           // supports broadcast access
    FlagLoopback             // is a loopback interface
    FlagPointToPoint         // is a point-to-point link
    FlagMulticast            // supports multicast access
)
```

These flags take on the values in a power-of-two sequence: 0x01, 0x02, 0x04, etc.

That makes them easy to combine, e.g. FlagUp | FlagLoopback

Enumerated types

Go also supports more complex iota expressions:

```
type ByteSize int64

const (
    _ = iota           // ignore first value
    KiB ByteSize = 1 << (10 * iota) // 2^10
    MiB                      // 2^20 (1 << 10*2)
    GiB
    TiB
    PiB
    EiB
)
```

So EiB is set to $2^{60} = 1152921504606846976 \approx 10^{19}$

Variable argument lists

What if we don't know how many parameters a function needs?

```
fmt.Printf("%#v\n", myMap)  
fmt.Printf("%s: %s\n", type, quantity)  
a := sum(1, 2, 3)  
b := sum(1, 2, 3, 4, 5)
```

All the formatted printing code uses variable argument lists

Variable argument lists

We use a special operator ... before the parameter type

```
func sum(nums ...int) int {  
    var total int  
  
    for _, num := range nums {  
        total += num  
    }  
  
    fmt.Printf("+/%v=%d\n", nums, total)  
    return total  
}  
  
// prints +/[1 2 3 4 5] = 15
```

Only the **last** parameter may have this operator

Variable argument lists

Since the parameter looks like a slice, we can pass a slice

```
func main() {
    fmt.Println(add())
    fmt.Println(add(11))
    fmt.Println(add(1, 2, 3, 4))

    s := []int{1, 2, 3}
    fmt.Println(add(s...))
}

// prints 0, 11, 10, 6
```

The special operator `...` after the actual parameter “unpacks” it into the variable argument list

Sized integers

Sometimes we need to handle low-level protocols (TCP/IP, etc.)

```
type TCPFields struct {
    SrcPort      uint16
    DstPort      uint16
    SeqNum       uint32
    AckNum       uint32
    DataOffset   uint8
    Flags        uint8
    WindowSize   uint16
    Checksum     uint16
    UrgentPtr    uint16
}
```

So we need to work with integers that have a particular size and/or are unsigned

Bitwise operators

We can mask off bits inside a byte or word

```
package main
import "fmt"

func main() {
    a, b := uint16(65535), uint16(281)

    fmt.Printf("%016b %#04[1]x\n", a)          // 1111111111111111 0xfffff
    fmt.Printf("%016b %#04[1]x\n", a &^ 0b1111) // 1111111111110000 0xffff0
    fmt.Printf("%016b %#04[1]x\n", a & 0b1111)  // 0000000000001111 0x000f

    fmt.Printf("%016b %#04[1]x\n", b)          // 0000000100011001 0x0119
    fmt.Printf("%016b %#04[1]x\n", ^b)         // 1111111011100110 0xeee6
    fmt.Printf("%016b %#04[1]x\n", b | 0b1111) // 0000000100011111 0x011f
    fmt.Printf("%016b %#04[1]x\n", b ^ 0b1111) // 0000000100010110 0x0116
}
```

Bitwise operators

We can combine the TCP declaration and an enumerated type:

```
// Flags that may be set in a TCP segment.  
const (  
    TCPFlagFin = 1 << iota  
    TCPFlagSyn  
    TCPFlagRst  
    TCPFlagPsh  
    TCPFlagAck  
    TCPFlagUrg  
)  
  
// true if both flags are set  
synAck := tcpHeader.Flags & (TCPFlagSyn|TCPFlagAck) == (TCPFlagSyn|TCPFlagAck)
```

Checking for bit flags this way is pretty common in low-level code

Bitwise operators

We can (logical) shift bits inside a byte or word

```
package main
import "fmt"

func main() {
    a, b, c := uint16(1024), uint16(255), uint16(0xff00)

    fmt.Printf("%016b %#04[1]x\n", a)          // 0000010000000000 0x0400
    fmt.Printf("%016b %#04[1]x\n", a << 3)      // 0010000000000000 0x2000
    fmt.Printf("%016b %#04[1]x\n", a << 13)     // 0000000000000000 0x0000

    fmt.Printf("%016b %#04[1]x\n", b)          // 0000000011111111 0x00ff
    fmt.Printf("%016b %#04[1]x\n", b << 2)      // 0000000111111100 0x03fc
    fmt.Printf("%016b %#04[1]x\n", b >> 2)     // 0000000000111111 0x003f
    fmt.Printf("%016b %#04[1]x\n", c >> 2)     // 0011111110000000 0x3fc0
}
```

Sized integers

The 32-bit values get truncated; high bit set \Rightarrow negative

```
package main
import "fmt"

func main() {
    var a, b uint32 = 66000, 2000000
    m, n := int16(a), int16(b) // 464, -31616
    fmt.Printf("%032b %016b %4d\n", a, uint16(m), m)
    fmt.Printf("%032b %016b %4d\n", b, uint16(n), n)
}
```

000000000000000010000000111010000	0000000111010000	464
0000000000001111010000100100000000	1000010010000000	-31616

Sized integers

Arithmetic with sized integers may overflow

```
package main
import "fmt"

func main() {
    a, b := uint16(8), uint16(128)      // compare to uint8
    x, y := uint16(a*a), uint16(b*b)

    fmt.Printf("%5d %#04[1]x\n", a)      //     8 0x0008 //     8 0x0008
    fmt.Printf("%5d %#04[1]x\n", x)      //    64 0x0040 //    64 0x0040
    fmt.Printf("%5d %#04[1]x\n", b)      //   128 0x0080 //   128 0x0080
    fmt.Printf("%5d %#04[1]x\n", y)      // 16384 0x4000 //     0 0x0000
}
```

The last multiplication doesn't fit (high bits disappear to the left)

Signed integers

There's one more negative number (e.g., -128, ..., -1, 0, 1, ..., 127)

```
package main
import "fmt"

func main() {
    a := int8(-128)      // try -127, -1 for comparison
    b, c := -a, a/-1
    d, e := a+1, a-1
    fmt.Printf("%4d %#02x\n", a, uint8(a)) // -128 0x80 // -127 0x81 // -1 0xff
    fmt.Printf("%4d %#02x\n", b, uint8(b)) // -128 0x80 // 127 0x7f // 1 0x01
    fmt.Printf("%4d %#02x\n", c, uint8(c)) // -128 0x80 // 127 0x7f // 1 0x01
    fmt.Printf("%4d %#02x\n", d, uint8(d)) // -127 0x81 // -126 0x82 // 0 0x00
    fmt.Printf("%4d %#02x\n", e, uint8(e)) // 127 0x7f // -128 0x80 // -2 0xfe
}
```

Weird things happen when we do 8-bit math with -128, be careful

Goto considered harmful *

Every once in a long while, goto is simply easier to understand

```
readFormat:  
    err = binary.Read(buf, binary.BigEndian, &header.format)  
  
    if err != nil {  
        return &header, nil, HeaderReadFailed.from(pos, err)  
    }  
  
    if header.format == junkID {  
        . . .           // find size & consume WAVE junk header  
        goto readFormat  
    }  
  
    if header.format != fmtID {  
        return &header, nil, InvalidChunkType  
    }
```

Programming in Go

Matt Holiday
Christmas 2020



Customizing Errors

Simple Errors

Most of the time, errors are just strings

```
func (h HAL9000) OpenPodBayDoors() error {
    if h.kill {
        return fmt.Errorf("I'm sorry %s, I can't do that", h.victim)
    }
    .
    .
}
```

Error types

Errors in Go are objects satisfying the `error` interface:

```
type error interface {
    Error() string
}
```

Any *concrete* type with `Error()` can represent an error

```
type Fizgig struct{}

func (f Fizgig) Error() string {
    return "Your fizgig is bent"
}
```

A custom error type

We're going to build out a custom error type

```
type errKind int

const (
    _           errKind = iota // so we don't start at 0
    noHeader
    cantReadHeader
    invalidHdrType
    ...
}

type WaveError struct {
    kind  errKind
    value int
    err   error
}
```

A custom error type

We use different formats depending on the situation

```
func (e WaveError) Error() string {
    switch e.kind {
    case noHeader:
        return "no header (file too short?)"

    case cantReadHeader:
        return fmt.Sprintf("can't read header[%d]: %s", e.value, e.err.Error())

    case invalidHdrType:
        return "invalid header type"

    case invalidChkLength:
        return fmt.Sprintf("invalid chunk length: %d", e.value)
    ...
}
```

A custom error type

We have a couple of helper methods to generate errors

```
// with returns an error with a particular value (e.g., header type)
func (e WaveError) with(val int) WaveError {
    e1 := e
    e1.value = val
    return e1
}

// from returns an error with a particular location and
// underlying error (e.g., from the standard library)
func (e WaveError) from(pos int, err error) WaveError {
    e1 := e
    e1.value = pos
    e1.err = err
    return e1
}
```

A custom error type

And we have some prototype errors we can return or customize

```
var (
    HeaderMissing      = WaveError{kind: noHeader}
    HeaderReadFailed   = WaveError{kind: cantReadHeader}
    InvalidHeaderType  = WaveError{kind: invalidHdrType}
    InvalidChunkLength = WaveError{kind: invalidChkLength}
    InvalidChunkType   = WaveError{kind: invalidChkType}
    InvalidDataLength   = WaveError{kind: invalidLength}

    ...
)
```

The custom error type in use

Here's an example of those errors in use

```
func DecodeHeader(b []byte) (*Header, []byte, error) {
    var err error
    var pos int

    header := Header{TotalLength: uint32(len(b))}
    buf := bytes.NewReader(b)

    if len(b) < headerSize {
        return &header, nil, HeaderMissing
    }

    if err = binary.Read(buf, binary.BigEndian, &header.riff); err != nil {
        return &header, nil, HeaderReadFailed.from(pos, err)
    }

    . . .
```

Wrapped errors

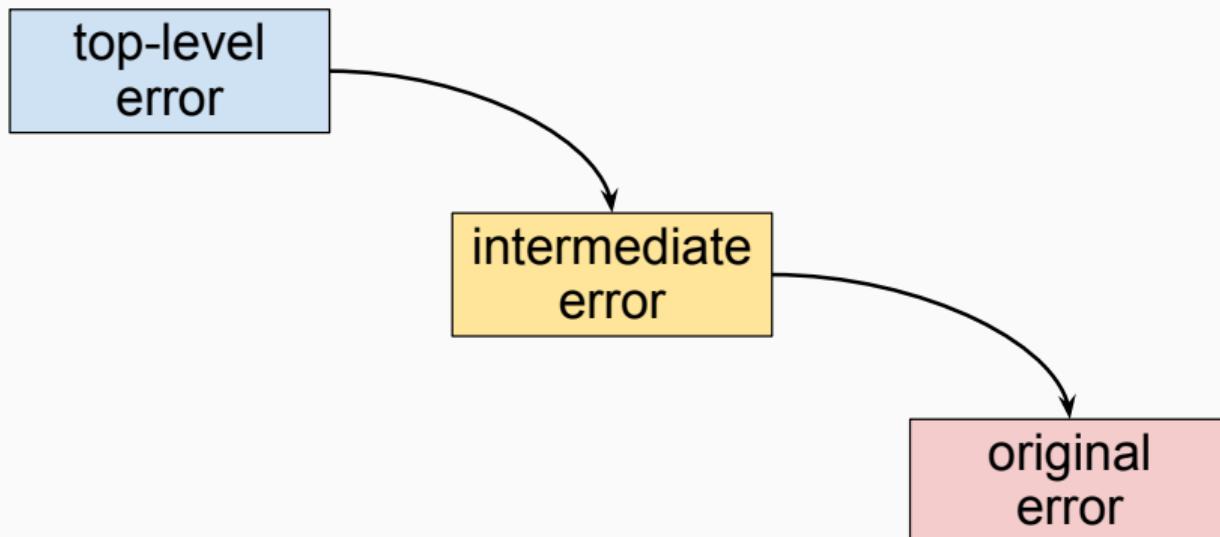
Starting with Go 1.13, we can wrap one error in another

```
func (h HAL9009) OpenPodBayDoors() error {
    . .
    if h.err != nil {
        return fmt.Errorf("I'm sorry %s, I can't: %w", h.victim, h.err)
    }
    . .
}
```

The easiest way to do that is to use the `%w` format verb with `fmt.Errorf()`

Wrapped errors

Wrapping errors gives us an error chain we can unravel



Wrapped errors

Custom error types may now unwrap their internal errors

```
type WaveError struct {
    kind  errKind
    value int
    err   error
}

func (w *WaveError) Unwrap() error {
    return w.err
}
```

errors.Is

We can check whether an error has another error in its chain

`errors.Is` compares with an error **variable**, not a type

```
    . . .
    if audio, err = DecodeWaveFile(fn); err != nil {
        if errors.Is(err, os.ErrPermission) {
            // let's report a security violation
        }
    }
    . . .
```

Customized tests

We can provide the `Is()` method for our custom type
(only useful if we export our error *variables* also)

```
type WaveError struct {
    kind errKind
    ...
}

func (w *WaveError) Is(t error) bool {
    e, ok := t.(*WaveError) // reflection again

    if !ok {
        return false
    }

    return e.errKind == w.errKind
}
```

`errors.As`

We can get an error of an underlying type if it's in the chain

`errors.As` looks for an error **type**, not a value

```
    . . .

if audio, err = DecodeWaveFile(fn); err != nil {
    var e os.PathError // a struct

    if errors.As(err, &e) {
        // let's pass back just the underlying file error

        return e
    }
}
```

A Philosophy of Error Handling

Errors in Go

When it comes to errors, you may fall into one of these camps:

1. you hate constantly writing if/else blocks
2. you think writing if/else blocks makes things clearer
3. **you don't care because you're too busy writing code**



Normal errors

Normal errors result from *input* or *external conditions*
(for example, a “file not found” error)

Go code handles this case by returning the `error` type

```
// Not exactly os.Open, but shows the basic logic

func Open(name string, flag int, perm FileMode) (*File, error) {
    r, e := syscall.Open(name, flag|syscall.O_CLOEXEC, syscallMode(perm))

    if e != nil {
        return nil, &PathError{"open", name, e}
    }

    return newFile(uintptr(r), name, kindOpenFile), nil
}
```

Abnormal errors

Abnormal errors result from *invalid program logic*
(for example, a nil pointer)

For program logic errors, Go code does a **panic**

```
func (d *digest) checkSum() [Size]byte {
    // finish writing the checksum
    . . .

    if d.nx != 0 {           // panic if there's data left over
        panic("d.nx != 0")
    }
    . . .
```

When your program has a logic bug



“Fail hard, fail fast”

When your program has a logic bug

If your server crashes, it will get immediate attention

- logs are often noisy
- so proactive log searches for “problems” are rare

We want evidence of the failure as close as possible in time and space to the original defect in the code

- connect the crash to logs that explain the context
- traceback from the point closest to the broken logic

In a distributed system, *crash failures* are the safest type to handle

- it's better to die than to be a zombie or babble or corrupt the DB
- not crashing may lead to *Byzantine failures*

When should we panic?

Only when the error was caused by our own programming defect, e.g.

- we can't walk a data structure we built
- we have an off-by-one bug encoding bytes

In other words,

*panic should be used when our assumptions of
our own programming design or logic are wrong*

These cases might use an “assert” in other programming languages

When should we panic?

A *B-tree* data structure satisfies several invariants:

1. every path from the root to a leaf has the same length
2. if a node has n children, it contains $n - 1$ keys
3. every node (except the root) is at least half full
4. the root has at least two children if it is not a leaf
5. subnode keys fall between the keys of the parent node that lie on either side of the subnode pointer

If any of these is ever false, the B-tree methods should **panic!**

Exception handling

Exception handling was popularized to allow “graceful degradation” of safety-critical systems (e.g., Ada and flight control software)

Ironically, most safety-critical systems are built without using exceptions!

Exception handling introduces invisible control paths through code

So code with exceptions is harder to analyze (automatically or by eye)

Exception handling

Officially, Go doesn't support exception handling as in other languages

Practically, it does — in the form of `panic` & `recover`

`panic` in a function will still cause deferred function calls to run

Then it will stop only if it finds a valid `recover` call in a `defer` as it unwinds the stack

Panic and recover

Recovery from `panic` only works inside `defer`

```
func abc() {
    panic("omg")
}

func main() {
    defer func() {
        if p := recover(); p != nil {
            // what else can you do?

            fmt.Println("recover:", p)
        }
    }()
    abc()
}
```

Define errors out of existence

Error (edge) cases are one of the primary sources of complexity

The best way to deal with many errors is to make them impossible

Design your abstractions so that most (or all) operations are safe:

- reading from a nil map
- appending to a nil slice
- deleting a non-existent item from a map
- taking the length of an uninitialized string

Try to reduce edge cases that are hard to test or debug (or even think about!)

Proactively Prevent Problems™

Every piece of data in your software should start life in a valid state

Every transformation should leave it in a valid state

- break large programs into small pieces you can understand
- hide information to reduce the chance of corruption
- avoid clever code and side effects
- avoid unsafe operations
- assert your invariants
- never ignore errors
- **test, test, test**

Never, ever accept input from a user (or environment) without validation

Error handling culture

“Go programmers think about the failure case first.

We solve the ‘what if ...’ case first. This leads to programs where failures are handled at the point of writing, rather than the point they occur in production. The verbosity of

```
if err != nil {  
    return err  
}
```

is outweighed by the value of deliberately handling each failure condition at the point at which it occurs. Key to this is the cultural value of handling each and every error explicitly.” — Dave Cheney

Programming in Go

Matt Holiday
Christmas 2020



Reflection

Type assertion

“interface{} says nothing” since it has no methods

It's a “generic” thing, but sometimes we need its “real” type

We can extract a specific type with a *type assertion* (a/k/a “downcasting”)

This has the form `value.(T)` for some type T

```
var w io.Writer = os.Stdout  
  
f := w.(*os.File)          // success: f == os.Stdout  
  
c := w.(*bytes.Buffer)    // panic: interface holds *os.File, not *bytes.Buffer
```

Type assertion

If we use the two-result version, we can avoid panic

```
var w io.Writer = os.Stdout  
  
f, ok := w.(*os.File)          // success: ok, f == os.Stdout  
  
b, ok := w.(*bytes.Buffer)    // failure: !ok, b == nil
```

Switching on type

We can also use type assertion in a `switch` statement
(matching a *type* not a *value*)

```
func Println(args ...interface{}) {
    buf := make([]byte, 0, 80)

    for arg := range args {
        switch a := arg.(type) {
            case string:                      // concrete type
                buf = append(buf, a...)
            case Stringer:                   // interface
                buf = append(buf, a.String()...)
            ...
        }
    }
}
```

Here the switch variable `a` has a specific type if the case has a *single* type

Deep equality

We can use the `reflect` package in UTs to check equality

```
want := struct{
    a: "a string",
    b: []int{1, 2, 3}      // not comparable with ==
}

got := gotGetIt( . . . )

if !reflect.DeepEqual(got, want) {
    t.Errorf("bad response: got=%#v, want=%#v", got, want)
}
```

You can use github.com/kylelemon/godebug/pretty to show a deep diff

Hard JSON

Not all JSON messages are well-behaved

What if some keys depend on others in the message?

```
{  
  "item": "album",  
  "album": {"title": "Dark Side of the Moon"}  
}
```

```
{  
  "item": "song",  
  "song": {"title": "Bella Donna", "artist": "Stevie Nicks"}  
}
```

Custom JSON decoding

We'll make a wrapper and a custom decoder

```
type response struct {
    Item    string `json:"item"`
    Album   string
    Title   string
    Artist  string
}

type respWrapper struct {
    response
}
```

We need `respWrapper` because it must have a *separate* unmarshal method from the `response` type (see below)

Custom JSON decoding

```
func (r *respWrapper) UnmarshalJSON(b []byte) (err error) {
    var raw map[string]interface{}

    err = json.Unmarshal(b, &r.response) // ignore error handling
    err = json.Unmarshal(b, &raw)

    switch r.Item {
    case "album":
        inner, ok := raw["album"].(map[string]interface{})

        if ok {
            if album, ok := inner["title"].(string); ok {
                r.Album = album
            }
        }
    }

    . . .
}
```

Custom JSON decoding

```
case "song":  
    inner, ok := raw["song"].(map[string]interface{})  
  
    if ok {  
        if title, ok := inner["title"].(string); ok {  
            r.Title = title  
        }  
  
        if artist, ok := inner["artist"].(string); ok {  
            r.Artist = artist  
        }  
    }  
  
    return err  
}
```

Custom JSON decoding

```
func main() {
    var resp1, resp2 respWrapper
    var err error

    if err = json.Unmarshal([]byte(j1), &resp1); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%#v\n", resp1.response)

    if err = json.Unmarshal([]byte(j2), &resp2); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%#v\n", resp2.response)
}
```

Custom JSON decoding

```
var j1 = `{
  "item": "album",
  "album": {"title": "Dark Side of the Moon"}
}`

var j2 = `{
  "item": "song",
  "song": {"title": "Bella Donna", "artist": "Stevie Nicks"}
}`

// main.response{Item:"album", Album:"Dark Side of the Moon",
//               Title:"", Artist:""}
//
// main.response{Item:"song", Album:"", Title:"Bella Donna",
//               Artist:"Stevie Nicks"}
```

Testing JSON

We want to know if a known fragment of JSON is contained in a larger unknown piece

```
{"id": "Z"} in? {"id": "Z", "part": "fizgig", "qty": 2}
```

All done with reflection from a generic map

```
func matchNum(key string, exp float64, data map[string]interface{}) bool {
    if v, ok := data[key]; ok {
        if val, ok := v.(float64); ok && val == exp {
            return true
        }
    }
    return false
}
```

Testing JSON

```
func matchString(key, exp string, data map[string]interface{}) bool {
    // is it in the map?

    if v, ok := data[key]; ok {
        // is it a string, and does it match?

        if val, ok := v.(string); ok && strings.EqualFold(val, exp) {
            return true
        }
    }

    return false
}
```

Testing JSON

```
func contains(exp, data map[string]interface{}) error {
    for k, v := range exp {
        switch x := v.(type) {

            case float64:
                if !matchNum(k, x, data) {
                    return fmt.Errorf("%s unmatched (%d)", k, int(x))
                }

            case string:
                if !matchString(k, x, data) {
                    return fmt.Errorf("%s unmatched (%s)", k, x)
                }

            ...
        }
    }
}
```

Testing JSON

```
    . . .

case map[string]interface{}:
    if val, ok := data[k]; !ok {
        return fmt.Errorf("%s missing in data", k)
    } else if unk, ok := val.(map[string]interface{}); ok {
        if err := contains(x, unk); err != nil {
            return fmt.Errorf("%s unmatched (%+v): %s", k, x, err)
        }
    } else {
        return fmt.Errorf("%s wrong in data (%#v)", k, val)
    }
}

return nil
}
```

Testing JSON

```
func CheckData(want, got []byte) error {
    var w, g map[string]interface{}

    if err := json.Unmarshal(want, &w); err != nil {
        return err
    }

    if err := json.Unmarshal(got, &g); err != nil {
        return err
    }

    return contains(w, g)
}
```

Testing JSON

Run the tests and analyze the code coverage

```
// go test -v
// go test ./... -cover
// go test ./... -coverprofile=c.out -covermode=count
// go tool cover -html=c.out

var unknown = `{
    "id": 1,
    "name": "bob",
    "addr": {
        "street": "Lazy Lane",
        "city": "Exit",
        "zip": "99999"
    },
    "extra": 21.1
}`
```

Testing JSON

```
func TestContains(t *testing.T) {
    var known = []string{
        `{"id": 1}`,
        `{"extra": 21.1}`,
        `{"name": "bob"} `,
        `{"addr": {"street": "Lazy Lane", "city": "Exit"}}`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err != nil {
            t.Errorf("invalid: %s (%s)\n", k, err)
        }
    }
}
```

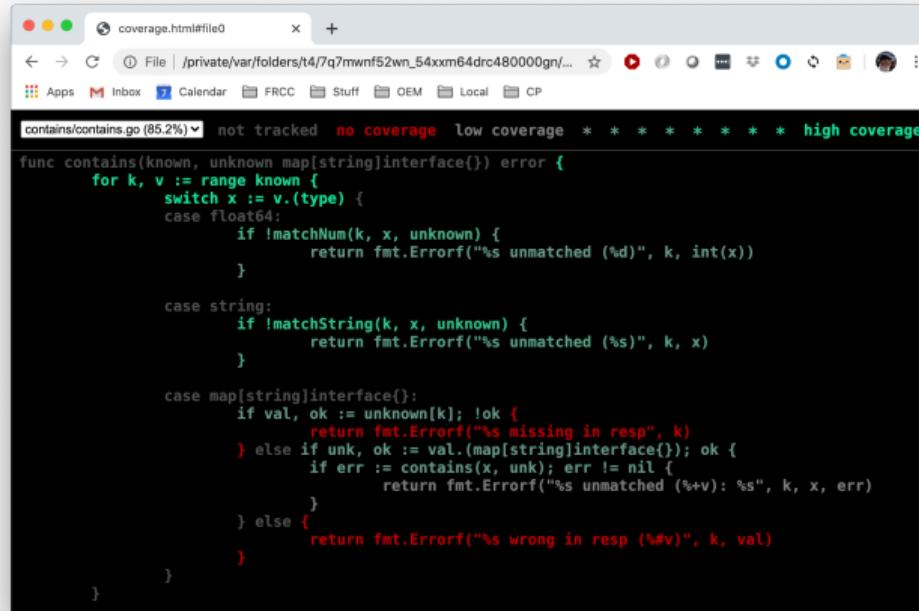
Testing JSON

```
func TestNotContains(t *testing.T) {
    var known = []string{
        `{"id": 2}`,
        `{"pid": 2}`,
        `{"name": "bobby"} `,
        `{"first": "bob"} `,
        `{"addr": {"street": "Lazy Lane", "city": "Alpha"} }`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err == nil {
            t.Errorf("false positive: %s\n", k)
        } else {
            t.Log(err)
        }
    }
}
```

Testing JSON

go test has options to help visualize code coverage



coverage.html#file0

contains/contains.go (85.2%) not tracked no coverage low coverage * * * * * * * * * high coverage

```
func contains(known, unknown map[string]interface{}) error {
    for k, v := range known {
        switch x := v.(type) {
        case float64:
            if !matchNum(k, x, unknown) {
                return fmt.Errorf("%s unmatched (%d)", k, int(x))
            }
        case string:
            if !matchString(k, x, unknown) {
                return fmt.Errorf("%s unmatched (%s)", k, x)
            }
        case map[string]interface{}:
            if val, ok := unknown[k]; !ok {
                return fmt.Errorf("%s missing in resp", k)
            } else if unk, ok := val.(map[string]interface{}); ok {
                if err := contains(x, unk); err != nil {
                    return fmt.Errorf("%s unmatched (%s+v): %s", k, x, err)
                }
            } else {
                return fmt.Errorf("%s wrong in resp (%#v)", k, val)
            }
        }
    }
}
```

Programming in Go

Matt Holiday
Christmas 2020



Mechanical Sympathy

Mechanical sympathy

“The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.” — Henry Petroski

We got similar *perceived* performance 30 years ago with

- 100 times less CPU
- 100 times less memory
- 100 times less disk space



Performance in the cloud

We've made a deliberate choice to accept some overhead

We have to trade off performance against other things:

- choice of architecture
- quality, reliability, scalability
- cost of development & ownership

We need to optimize where we can, given those choices

We still want **simplicity, readability & maintainability of code**

Optimization

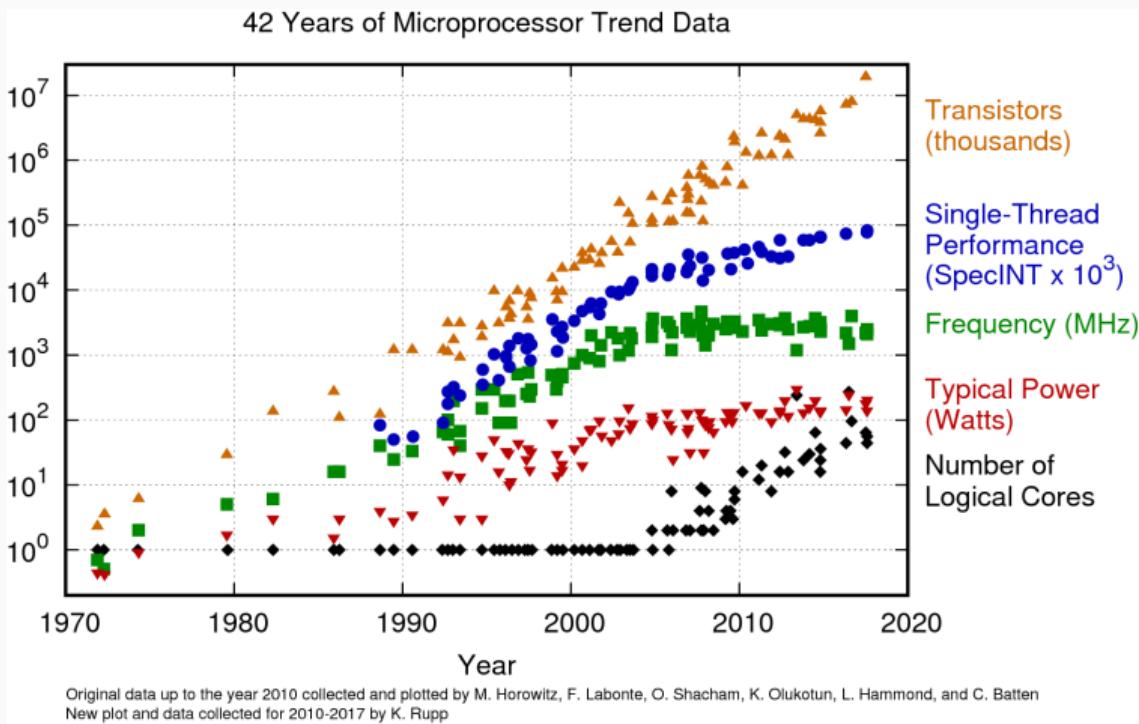
Top-down refinement:

Architecture	latency, cost of communication
Design	algorithms, concurrency, layers
Implementation	programming language, memory use

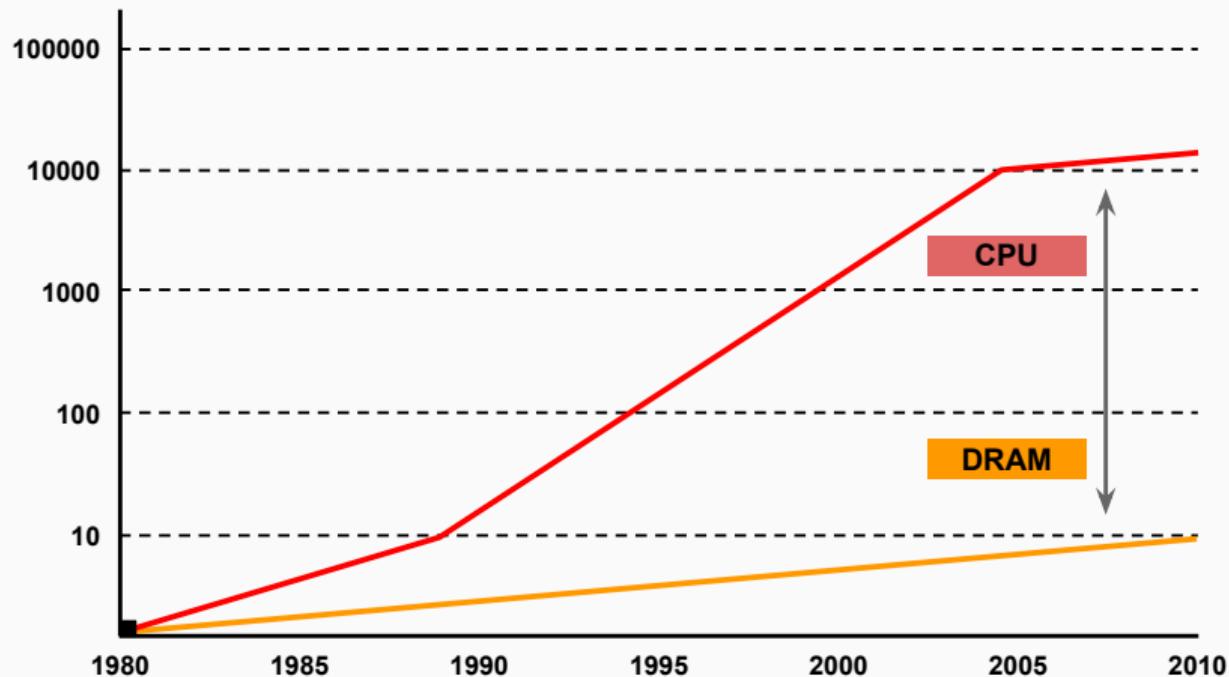
Mechanical sympathy plays a role in our *implementation*

Interpreted languages may cost 10x more to operate due to their inefficiency

CPU performance



Memory performance



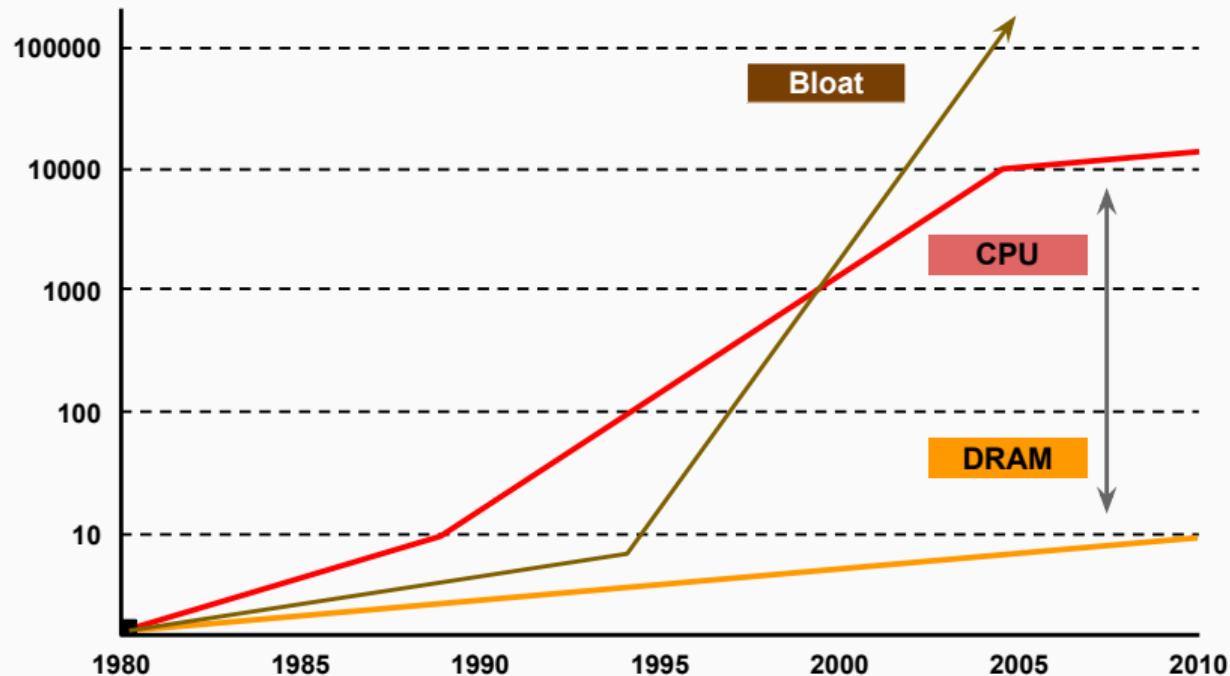
Mechanical sympathy

Some unfortunate realities:

- CPUs aren't getting faster any more
- the gap between memory and CPU isn't shrinking
- software gets slower more quickly than CPUs get faster

Software development costs exceed hardware costs

Software bloatation



Mechanical sympathy

Two competing realities

- maintain or improve the performance of software
- control the cost of developing software

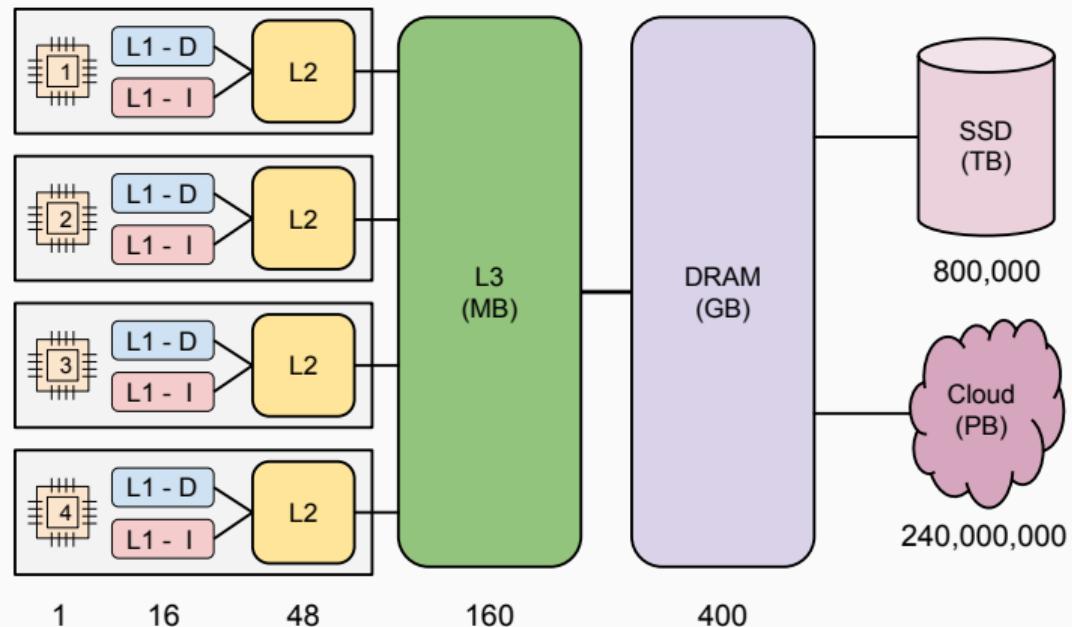
The only way to do that is

- make software simpler
- **make software that works *with* the machine**, not against it

Make software suck less

Memory hierarchy

As memory capacity increases, access latency also increases



Memory caching

“Computational” cost is often dominated by **memory access cost**

Caching takes advantage of access patterns to keep frequently-used code and data “close” to the CPU to reduce access time

Caching imposes some costs of its own

- Memory access by the **cache line**, typically 64 bytes
- **Cache coherency** to manage cache line ownership

Locality

Locality in space:

access to one thing implies access to another nearby

Locality in time:

access implies we're likely to access it again soon

Caching hardware and performance benchmarks usually favor large-scale “number crunching” problems where the software makes optimal use of the cache

Cache efficiency

Caching is effective when we use (and reuse) entire cache lines

Caching is effective when we access memory in predictable patterns
(but only sequential access is predictable)

We get our best performance when we

- **keep things in contiguous memory**
- **access them sequentially**

Cache efficiency

Things that make the cache **less efficient**:

- synchronization between CPUs
- copying blocks of data around in memory
- non-sequential access patterns (calling functions, chasing pointers)

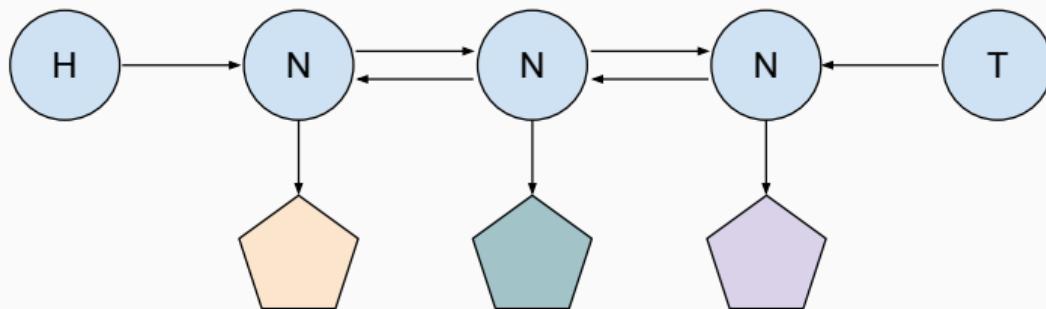
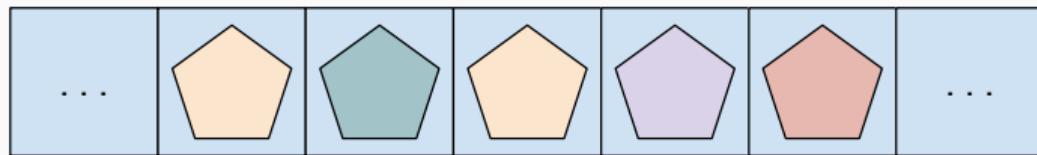
A little copying is better than a lot of pointer chasing!

Things that make the cache **more efficient**:

- keeping code or data in cache longer
- keeping data together (so all of a cache line is used)
- processing memory in sequential order (code or data)

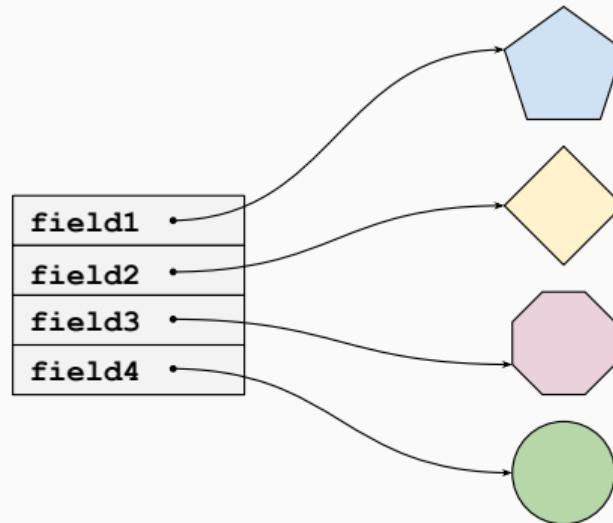
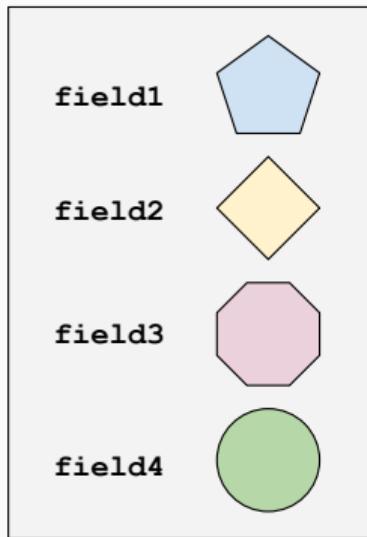
Access patterns

A slice of objects beats a list with pointers



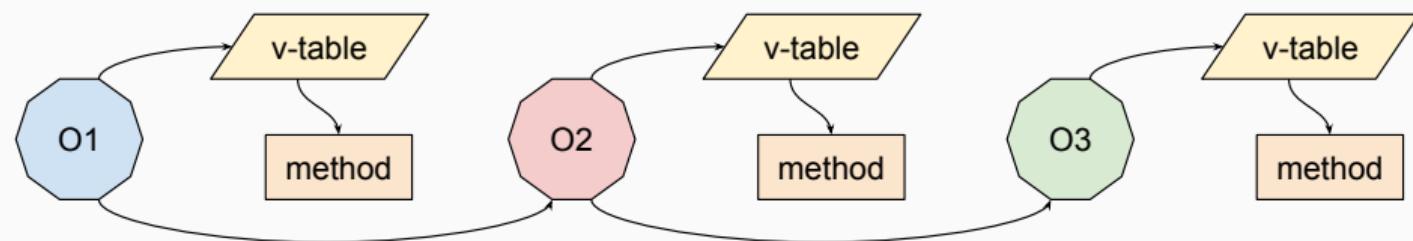
Access patterns

A struct with contiguous fields beats a class with pointers



Access patterns

Calling lots short methods via dynamic dispatch is very expensive



The cost of calling a function should be proportional to the work it does
(short inline functions vs longer methods with late binding)

Synchronization costs

Synchronization has two costs:

- the actual cost to synchronize (lock & unlock)
- the impact of contention if we create a “hot spot”

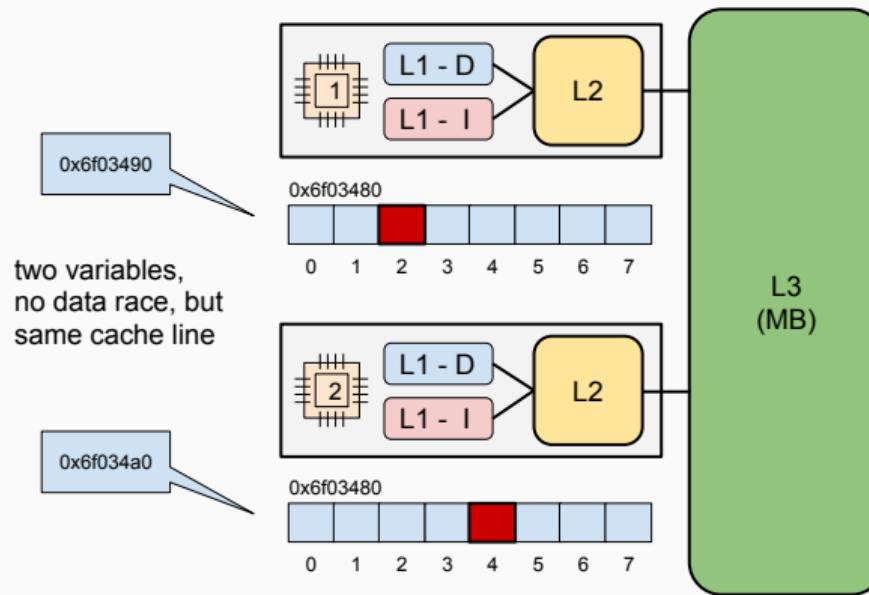
In the worst case, synchronization can make the program sequential

Amdahl's Law:

total speedup is limited by the fraction of the program that runs sequentially

Synchronization costs

False sharing: cores fight over a cache line for *different* variables



Other costs

There are other hidden costs:

- disk access
- garbage collection
- virtual memory & its cache
- context switching between processes

The only one you can really control is GC:

- reduce unnecessary allocations
- reduce embedded pointers in objects
- paradoxically, you may want a larger heap

Optimization

Go (and the Go philosophy) encourages good design:
you can choose

- to allocate contiguously
- to copy or not copy
- to allocate on the stack or heap (sometimes)
- to be synchronous or asynchronous
- to avoid unnecessary abstraction layers
- to avoid short / fowarding methods

Go doesn't get between you and the machine

Good code in Go doesn't hide the costs involved

Optimization

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about *small* efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%.”

— Don Knuth

Optimization

“There are only three optimizations:

1. Do less
2. Do it less often
3. Do it faster

The largest gains come from 1, but we spend all our time on 3.”

— Michael Fromberger

Programming in Go

Matt Holiday
Christmas 2020



Benchmarking

Go benchmarks

Go has standard tools and conventions for running benchmarks

Benchmarks live in test files ending with `_test.go`

You run benchmarks with `go test -bench`

Go only runs the `BenchmarkXXX` functions

Simple example function

```
func Fib(n int, recursive bool) int {
    switch n {
    case 0:
        return 0
    case 1:
        return 1
    default:
        if recursive {
            return Fib(n-1, r) + Fib(n-2, r)
        }
        a, b := 0, 1
        for i := 1; i < n; i++ {
            a, b = b, a+b
        }
        return b
    }
}
```

Simple benchmark tests

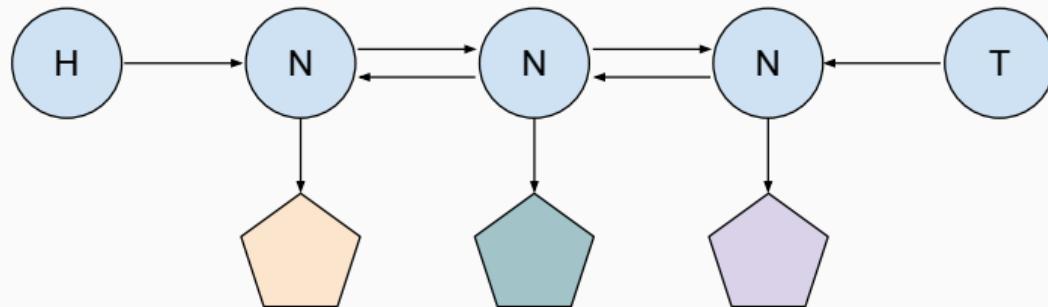
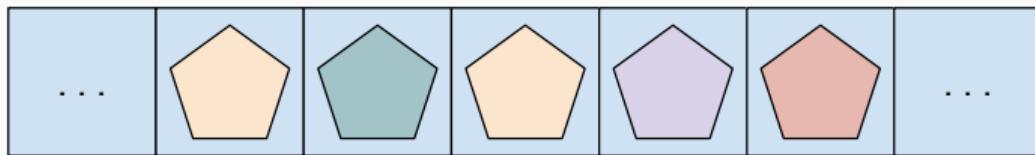
```
func BenchmarkFib20T(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(20, true) // run the Fib function b.N times
    }
}

func BenchmarkFib20F(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(20, false) // run the Fib function b.N times
    }
}
```

```
$ go test -bench=. ./fib_test.go
goos: darwin
goarch: amd64
BenchmarkFib20T-16          20851          59634 ns/op
BenchmarkFib20F-16          94855990        12.6 ns/op
```

List vs Slice example

A slice of objects beats a list with pointers



List vs Slice example

```
package main
import "testing"

type node struct {
    v int           // value in the list node
    t *node
}

func insert(i int, h *node) *node {
    t := &node{i, nil}

    if h != nil {
        h.t = t
    }

    return t
}
```

List vs Slice example

```
func mkList(n int) *node {
    var h, t *node

    h = insert(0, h)
    t = insert(1, h)
    for i := 2; i < n; i++ {
        t = insert(i, t)
    }
    return h
}

func sumList(n *node) (i int) {
    for h := n; h != nil; h = h.t {
        i += h.v
    }
    return
}
```

List vs Slice example

```
func mkSlice(n int) []int {
    r := make([]int, n)

    for i := 0; i < n; i++ {
        r[i] = i
    }

    return r
}

func sumSlice(l []int) (i int) {
    for _, v := range l {
        i += v
    }

    return
}
```

List vs Slice example

```
func BenchmarkList(b *testing.B) {
    for n := 0; n < b.N; n++ {
        l := mkList(1200)
        sumList(l)
    }
}

func BenchmarkSlice(b *testing.B) {
    for n := 0; n < b.N; n++ {
        l := mkSlice(1200)
        sumSlice(l)
    }
}
```

```
$ go test -bench=. ./list_test.go
BenchmarkList-16          35452          33904 ns/op
BenchmarkSlice-16         769028          1555 ns/op
```

List vs Slice example

In this example, we'll separate out the cost of making the list

```
func BenchmarkList(b *testing.B) {
    l := mkList(1200); b.ResetTimer()
    for n := 0; n < b.N; n++ {
        sumList(l)
    }
}
```

```
func BenchmarkSlice(b *testing.B) {
    l := mkSlice(1200); b.ResetTimer()
    for n := 0; n < b.N; n++ {
        sumSlice(l)
    }
}
```

BenchmarkList-16	885607	1243 ns/op
BenchmarkSlice-16	2910057	414 ns/op

List vs Slice example

In this version, we keep the value separate from the list node

```
type node struct {
    v *int
    t *node
}

func insert(i int, n *node) *node {
    t := &node{&i, nil}

    if n != nil {
        n.t = t
    }

    return t
}
```

List vs Slice example

In this version, we keep the value separate from the list node

```
func sumList(n *node) (i int) {
    for h := n; h != nil; h = h.t {
        i += *h.v
    }
    return
}
```

BenchmarkList-16	707344	1596 ns/op
BenchmarkSlice-16	2883547	417 ns/op

What if we go back and include the cost of building the list?

BenchmarkList-16	24858	48690 ns/op
BenchmarkSlice-16	662131	1518 ns/op

List vs Slice example

We can extend the benchmark to include memory allocations using the `-benchmem` flag

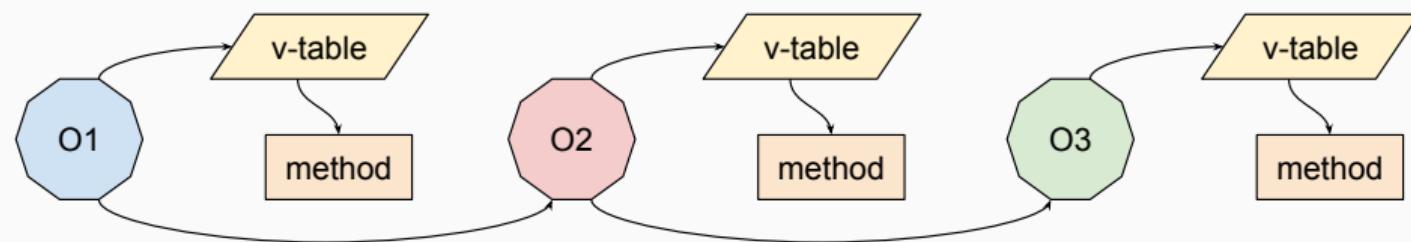
This version is with the list building included (since that's where the allocation takes place)

We don't just allocate more, we do it in smaller chunks

```
$ go test -bench=. -benchmem ./list_test.go
goos: darwin
goarch: amd64
BenchmarkList-16    23977  49766 ns/op  28800 B/op  2400 allocs/op
BenchmarkSlice-16  741272   1515 ns/op   9728 B/op       1 allocs/op
```

Forwarding example

Calling lots short methods via dynamic dispatch is very expensive



The cost of calling a function should be proportional to the work it does
(short inline functions vs longer methods with late binding)

Forwarding example

```
package forward
import ("math/rand"; "testing")

const defaultChars = "01234567 . . . mnopqrstuvwxyz"

func randString(length int, charset string) string {
    b := make([]byte, length)
    for i := range b {
        b[i] = charset[rand.Intn(len(charset))]
    }
    return string(b)
}

type forwarder interface {
    forward(string) int
}
```

Forwarding example

```
type thing1 struct {
    t forwarder
}

func (t1 *thing1) forward(s string) int {
    return t1.t.forward(s)
}

type thing2 struct {
    t forwarder
}

func (t2 *thing2) forward(s string) int {
    return t2.t.forward(s)
}

type thing3 struct {}
```

Forwarding example

```
func (t3 *thing3) forward(s string) int {
    return len(s)
}

func length(s string) int {
    return len(s)
}

func BenchmarkDirect(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h[i] = length(r)
    }
}
```

Forwarding example

```
func BenchmarkForward(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)

    var t3 forwarder = &thing3{}
    var t2 forwarder = &thing2{t3}
    var t1 forwarder = &thing1{t2}

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h[i] = t1.forward(r)
    }
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8      1000000000          0.627 ns/op
BenchmarkForward-8     189176289          6.35 ns/op
```

Forwarding example

Let's make one small change

```
//go:noinline
func length(s string) int {
    return Len(s)
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8      624469180          1.87 ns/op
BenchmarkForward-8     189169784          6.35 ns/op
```

Presumably that makes both function calls happen out of line

Forwarding example

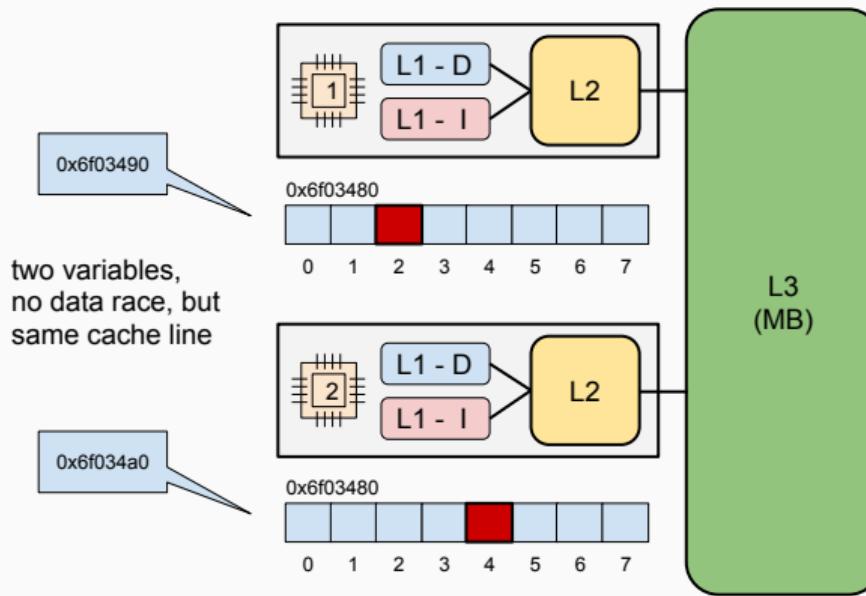
Let's make one other change, not so small

```
func BenchmarkForward(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        var t3 forwarder = &thing3{}
        var t2 forwarder = &thing2{t3}
        var t1 forwarder = &thing1{t2}
        h[i] = t1.forward(r)
    }
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8      623482398          1.88 ns/op
BenchmarkForward-8     23797018          49.2 ns/op
```

False sharing example

False sharing: cores fight over a cache line for *different* variables



False sharing example

We're going to let CPUs clobber each other's cache

```
import (
    "sync"
    "testing"
)

const (
    nworker = 8
    buffer  = 1024
)

var wg sync.WaitGroup
```

False sharing example

```
func run() (total int) {
    cnt := make([]uint64, nworker)    // one cache line
    in := make([]chan int, nworker)
    for i := 0; i < nworker; i++ {
        in[i] = make(chan int, buffer)
        go fill(10000, in[i])
    }
    for i := 0; i < nworker; i++ {
        wg.Add(1)
        go count(&cnt[i], in[i])
    }
    wg.Wait()
    for _, v := range cnt {
        total += int(v)
    }
    return
}
```

False sharing example

```
func count(cnt *uint64, in <-chan int) {
    // false sharing

    for i := range in {
        *cnt += uint64(i)
    }

    wg.Done()
}

func fill(n int, in chan<- int) {
    for i := 0; i < n; i++ {
        in <- i
    }

    close(in)
}
```

False sharing example

```
func BenchmarkShare(b *testing.B) {
    for i := 0; i < b.N; i++ {
        run()
    }
}
```

We're going to run with different numbers of cores

```
$ go test -bench=. -benchtime=10s -cpu=2,4,8 ./share_test.go
goos: darwin
goarch: amd64
BenchmarkShare-2          3630      3280637 ns/op
BenchmarkShare-4          4291      2832584 ns/op
BenchmarkShare-8          4910      2629852 ns/op
```

And watch it blow up due to false sharing

False sharing example

Now we're going to write back a local total instead

```
func count(cnt *uint64, in <-chan int) {
    var total int

    for i := range in {
        total += i
    }
    *cnt = uint64(total)
    wg.Done()
}
```

We actually see real improvement with more cores

BenchmarkShare-2	5526	2206546 ns/op
BenchmarkShare-4	9950	1241392 ns/op
BenchmarkShare-8	13222	998438 ns/op

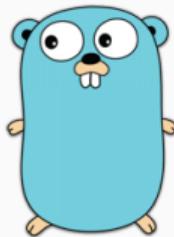
A few things to consider

Here are some concerns about CPU/memory benchmarking:

- is the data / code available in cache?
- did you hit a garbage collection?
- did virtual memory have to page in/out?
- did branch prediction work the same way?
- did the compiler remove code via optimization?
(are there side effects in the code?)
- are you running in parallel? how many cores?
- are those cores physical or virtual?
- are you sharing a core with anything else?
- what other processes are sharing the machine?

Programming in Go

Matt Holiday
Christmas 2020



Profiling

Web proxy example

We're going to write a proxy for the “todo” JSON demo server

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"

    _ "net/http/pprof"
)

const url = "https://jsonplaceholder.typicode.com/todos/"
```

Web proxy example

```
type todo struct {
    UserID      int      `json:"userID"`
    ID         int      `json:"id"`
    Title      string   `json:"title"`
    Completed  bool     `json:"completed"`
}

var mark = map[bool]string{
    false: " ",
    true:  "x",
}

func handler(w http.ResponseWriter, r *http.Request) {
    var item todo

    req, _ := http.NewRequest("GET", url+r.URL.Path[1:], nil)
    . . .
```

Web proxy example

```
 . . .
tr := &http.Transport{}
cli := &http.Client{Transport: tr}
resp, err := cli.Do(req)

if err != nil {
    http.Error(w, err.Error(), http.StatusBadGateway)
}

if resp.StatusCode != http.StatusOK {
    http.NotFound(w, r)
    return
}

body, _ := ioutil.ReadAll(resp.Body)
. . .
```

Web proxy example

```
    . . .

if err := json.Unmarshal(body, &item); err != nil {
    http.Error(w, err.Error(),
        http.StatusInternalServerError)
    return
}

fmt.Fprintf(w, "[%s] %d - %s\n", mark[item.Completed],
    item.ID, item.Title)
}

func main() {
    http.HandleFunc("/", handler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Running it

We can now open `http://localhost:8080/debug/pprof`
which is the default route

If we look at goroutines, we'll see 4 goroutines initially, but as
we exercise the server, that number will grow

```
$ curl http://localhost:8080/1
[ ] 1 - delectus aut autem

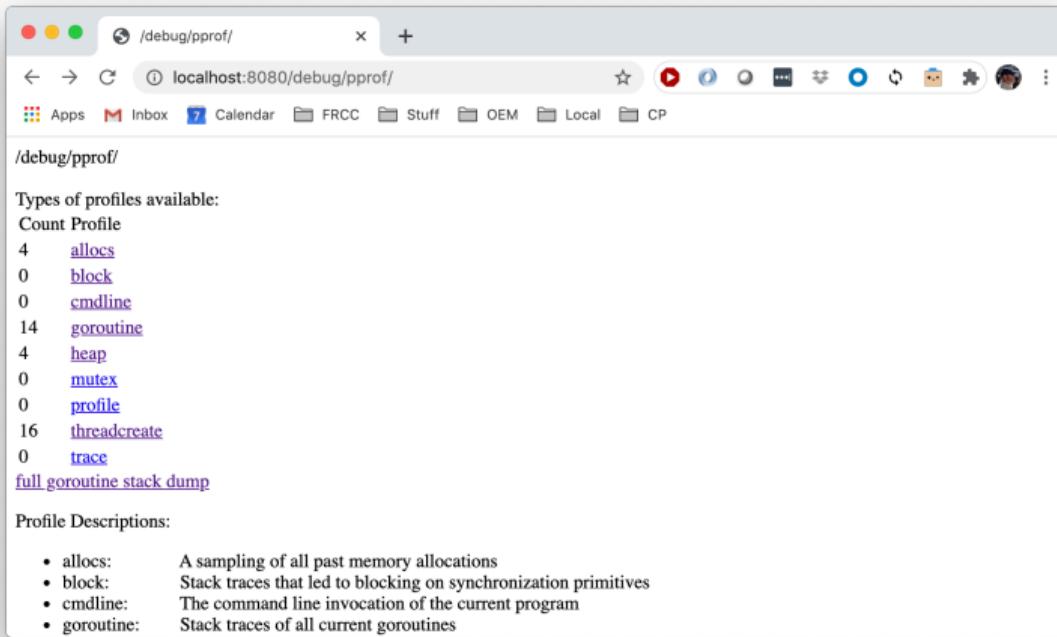
$ curl http://localhost:8080/8
[x] 8 - quo adipisci enim quam ut ab

$ curl http://localhost:8080/14
[x] 14 - repellendus sunt dolores architecto voluptatum

$ curl http://localhost:8080/1133
404 page not found
```

Main pprof page

This is what including pprof provides for us



The screenshot shows a web browser window with the title bar reading "/debug/pprof/" and the URL "localhost:8080/debug/pprof/". The page content is as follows:

```
/debug/pprof/  
  
Types of profiles available:  
Count Profile  
4    allocs  
0    block  
0    cmdline  
14   goroutine  
4    heap  
0    mutex  
0    profile  
16   threadcreate  
0    trace  
full goroutine stack dump  
  
Profile Descriptions:  


- allocs: A sampling of all past memory allocations
- block: Stack traces that led to blocking on synchronization primitives
- cmdline: The command line invocation of the current program
- goroutine: Stack traces of all current goroutines

```

Goroutine page

In this view we see a list of (hanging) goroutines

```
goroutine profile: total 14
10 @ 0x10326f0 0x102d6da 0x102cc55 0x10c2f95 0x10c3f6b 0x10c3f4d 0x11b165f 0x11c3e48 0x121c8b0 0x10e9174 0x121caf
0x121b054 0x121f2c1 0x121f2cc 0x116873a 0x106f9a7 0x12a0717 0x12a06ca 0x12a0f81 0x12c141e 0x12c0b83 0x105fa71
# 0x102cca4 internal/poll.runtime_pollWait+0x54
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/runtime/netpoll.go:184
# 0x10c2f94 internal/poll.(*pollDesc).wait+0x44
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/internal/poll/fd_poll_runtime.go:87
# 0x10c3f6a internal/poll.(*pollDesc).waitRead+0x2a
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/internal/poll/fd_poll_runtime.go:92
# 0x10c3f4c internal/poll.(*FD).Read+0x20c
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/internal/poll/fd_unix.go:169
# 0x11b165e net.(*netFD).Read+0x4e
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/net/fd_unix.go:202
# 0x11c3e47 net.(*conn).Read+0x67
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/net/net.go:184
# 0x121c8af crypto/tls.(*atLeastReader).Read+0x5f
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/crypto/tls/conn.go:780
# 0x10e9173 bytes.(*Buffer).ReadFrom+0xb3
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/bytes/buffer.go:204
# 0x121caf8 crypto/tls.(*Conn).readFromUntil+0xeb
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/crypto/tls/conn.go:802
# 0x121b053 crypto/tls.(*Conn).readRecordOrCCS+0x123
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/crypto/tls/conn.go:609
# 0x121f2c0 crypto/tls.(*Conn).readRecord+0x160
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/crypto/tls/conn.go:577
# 0x121f2cb crypto/tls.(*Conn).Read+0x16b
/usr/local/Cellar/go@1.13/1.13.12/libexec/src/crypto/tls/conn.go:1255
# 0x1168739 bufio.(*Reader).Read+0x269
```

Fixing it

We'll see that all the goroutines are handling HTTP sockets

And the allocations happen where we read from the response body
but forgot to close it

```
// we need this to recover the socket

defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {
    http.NotFound(w, r)
    return
}

body, _ := ioutil.ReadAll(resp.Body)
```

Prometheus metrics

We can add a Prometheus client which exposes metrics
(they can be scraped by hitting /metrics)

```
import ()  
    "github.com/prometheus/client_golang/prometheus"  
    "github.com/prometheus/client_golang/promhttp"  
)  
  
var queries = prometheus.NewCounter(prometheus.CounterOpts{  
    Name: "all_queries",  
    Help: "How many queries we've received.",  
})
```

Prometheus metrics

We add a line of code to the end of our handler

```
    . . .
    fmt.Fprintf(w, "[%s] %d - %s\n", mark[item.Completed], item.ID, item.Title)
    queries.Inc()
}
```

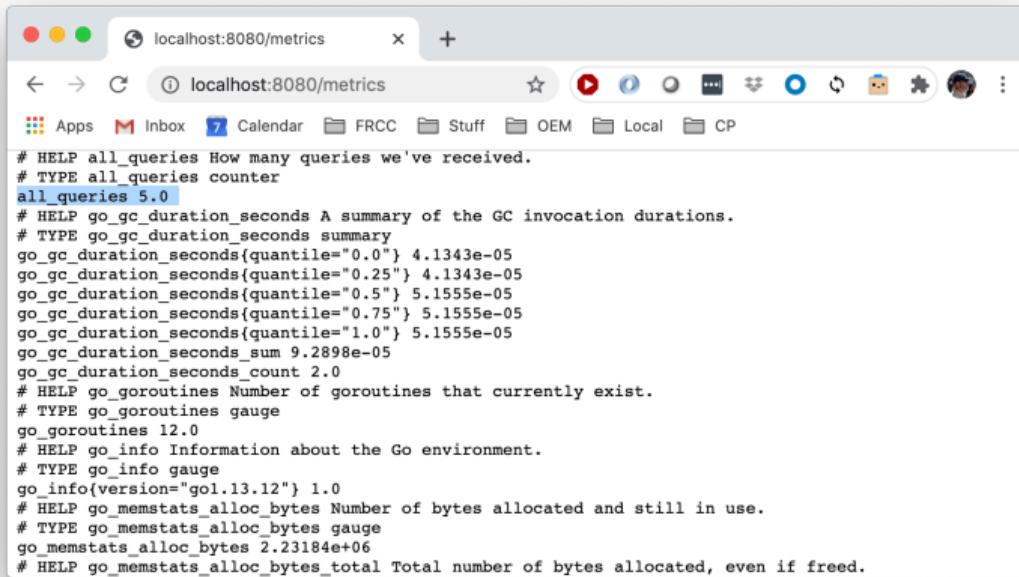
And an extra route in our main function

```
func main() {
    prometheus.MustRegister(queries)

    http.HandleFunc("/", handler)
    http.Handle("/metrics", promhttp.Handler())
    . . .
```

Metrics page

We get our metrics as well as a many system metrics



```
# HELP all_queries How many queries we've received.
# TYPE all_queries counter
all_queries 5.0
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0.0"} 4.1343e-05
go_gc_duration_seconds{quantile="0.25"} 4.1343e-05
go_gc_duration_seconds{quantile="0.5"} 5.1555e-05
go_gc_duration_seconds{quantile="0.75"} 5.1555e-05
go_gc_duration_seconds{quantile="1.0"} 5.1555e-05
go_gc_duration_seconds_sum 9.2898e-05
go_gc_duration_seconds_count 2.0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 12.0
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="gol.13.12"} 1.0
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 2.23184e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
```

Prometheus metrics

Certain metrics will be a dead giveaway that we're leaking sockets

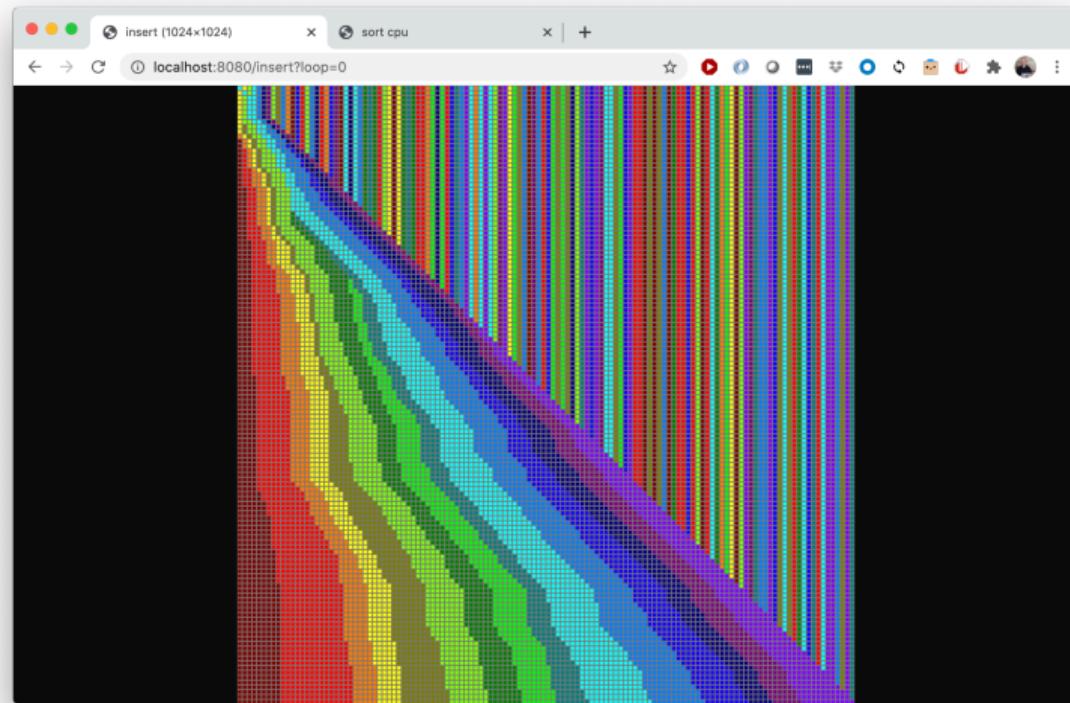
Not only will the goroutine count grow, but also the count of open file descriptors (FDs) *[Linux only – not on macOS]*

```
# HELP go_goroutines Number of goroutines that currently exist.  
# TYPE go_goroutines gauge  
go_goroutines 19.0
```

```
# HELP go_threads Number of OS threads created.  
# TYPE go_threads gauge  
go_threads 17.0
```

```
# HELP process_open_fds Number of open file descriptors.  
# TYPE process_open_fds gauge  
process_open_fds 19.0
```

Sort animation example



Sort animation example

The program animates various sort algorithms

- insertion sort: `insert`
- versions of quicksort: `qsort`, `qsortm`, `qsort3`, `qsorti`, `qsortf`

It creates an animated GIF with one frame for each row that changes
(each step in the sort until it's done)

It must draw a square of an entry's color outlined in gray

- 1024 rows and columns
- each entry is a square 8 pixels on a side

How to profile

We must build the program as a binary: `go build .`

We're going to use `pprof` again and hit the endpoint
`http://localhost:8080/debug/pprof/profile`

After 30 seconds it will download a profile file

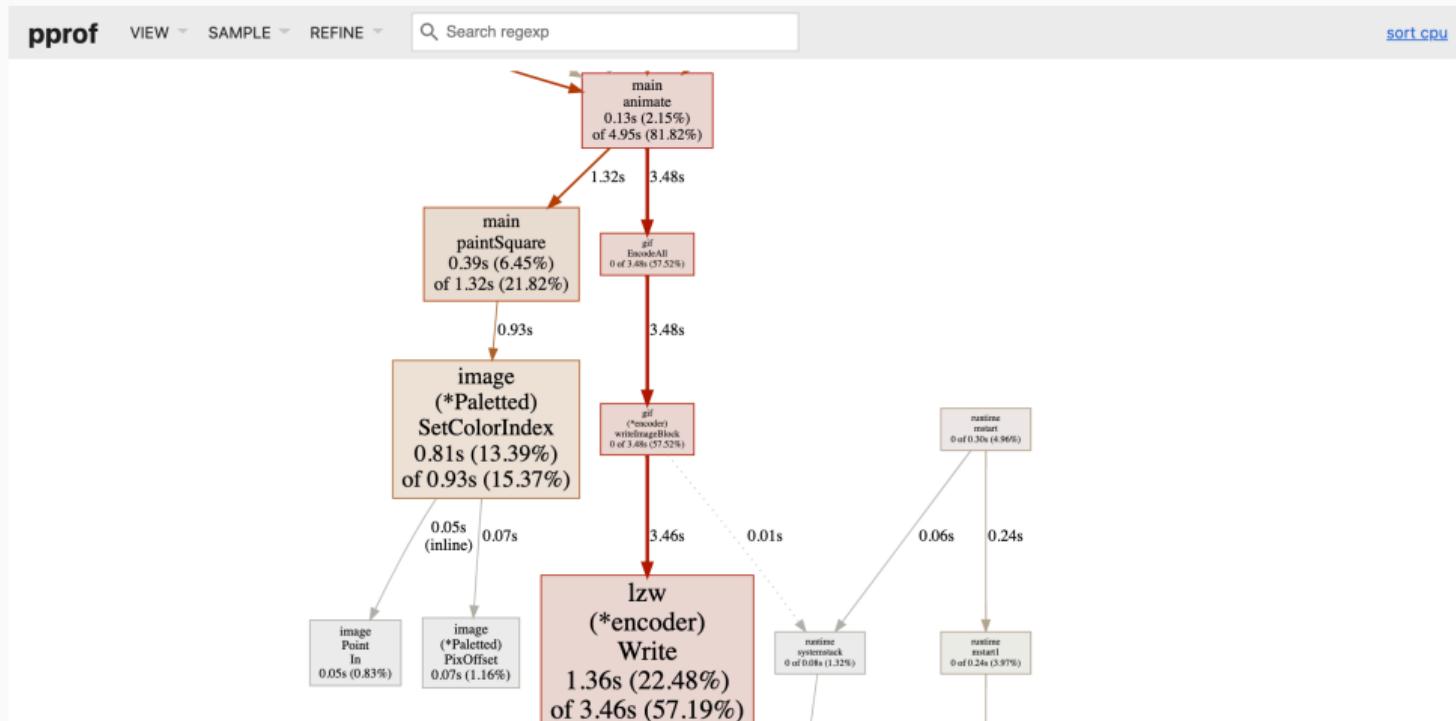
Then we can run the tool: `go tool pprof <binary> <profile-file>`
in one of three ways

- interactive, with a prompt
- just get the top entries with `-top`
- open a browser with `-http=:6060`

CPU flame graph



CPU profile



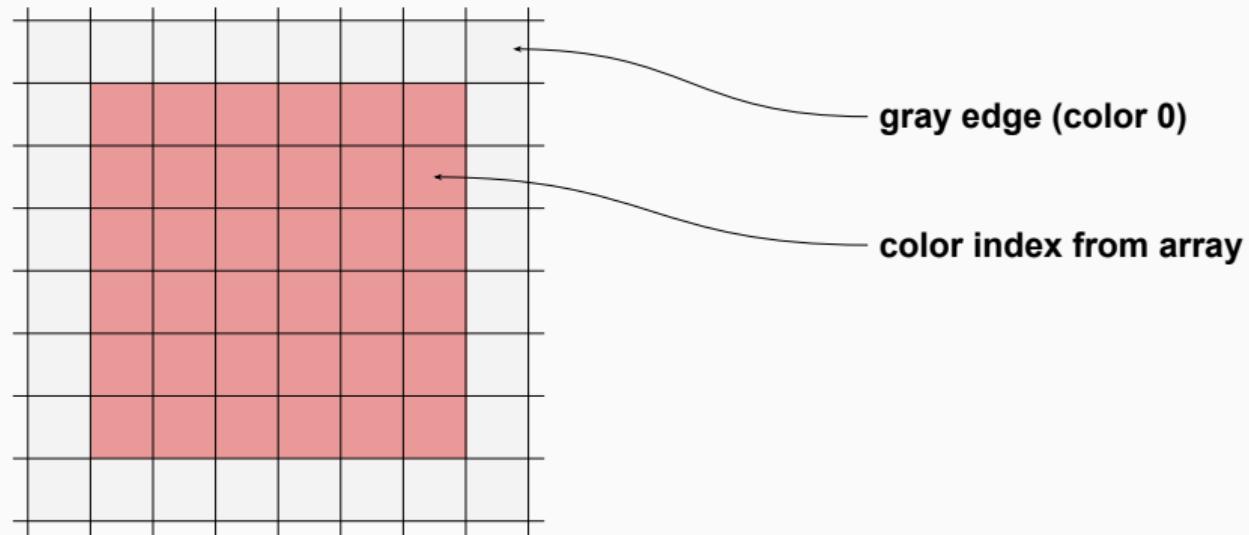
CPU profile

```
$ go tool pprof -top sort profile-slow
File: sort
Type: cpu
Time: Jan 31, 2021 at 8:29am (MST)
Duration: 30s, Total samples = 6.05s (20.17%)
Showing nodes accounting for 5.99s, 99.01% of 6.05s total
Dropped 31 nodes (cum <= 0.03s)

      flat  flat%  sum%          cum   cum%
  1.68s 27.77% 27.77%    1.68s 27.77%  syscall.syscall
  1.36s 22.48% 50.25%    3.46s 57.19%  compress/lzw.(*encoder).Write
  0.81s 13.39% 63.64%    0.93s 15.37%  image.(*Paletted).SetColorIndex
  0.39s  6.45% 70.08%    1.32s 21.82%  main.paintSquare
  0.25s  4.13% 74.21%    0.25s  4.13%  runtime.nanotime1
  0.22s  3.64% 77.85%    0.22s  3.64%  runtime.kevent
  0.19s  3.14% 80.99%    2.01s 33.22%  compress/lzw.(*encoder).writeLSB
  0.19s  3.14% 84.13%    0.41s  6.78%  runtime.netpoll
  0.14s  2.31% 86.45%    1.82s 30.08%  image/gif.blockWriter.WriteByte
  0.14s  2.31% 88.76%    0.14s  2.31%  runtime.pthread_cond_wait
  0.13s  2.15% 90.91%    4.95s 81.82%  main.animate
```

Painted squares

We represent each “item” in the array as a square



Painting each square

We're using a standard library function `SetColorIndex`

```
func paintSquare(i, k int, src []int, img *image.Paletted) {
    // Lay down a square with an outline using the default
    // color (gray; we deliberately excluded it from the data)

    for x := 0; x < scale; x++ {
        for y := 0; y < scale; y++ {
            idx := uint8(src[i])

            if x == 0 || y == 0 || x == scale-1 || y == scale-1 {
                idx = 0
            }
            img.SetColorIndex(i*scale+x, k*scale+y, idx)
        }
    }
}
```

Setting the color index

In `image/image.go`, we see it does extra work

- checking the point's location
- recalculating the pixel's offset

```
func (p *Paletted) PixOffset(x, y int) int {
    return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*1
}

func (p *Paletted) SetColorIndex(x, y int, index uint8) {
    if !(Point{x, y}.In(p.Rect)) {
        return
    }
    i := p.PixOffset(x, y)
    p.Pix[i] = index
}
```

Optimize the function

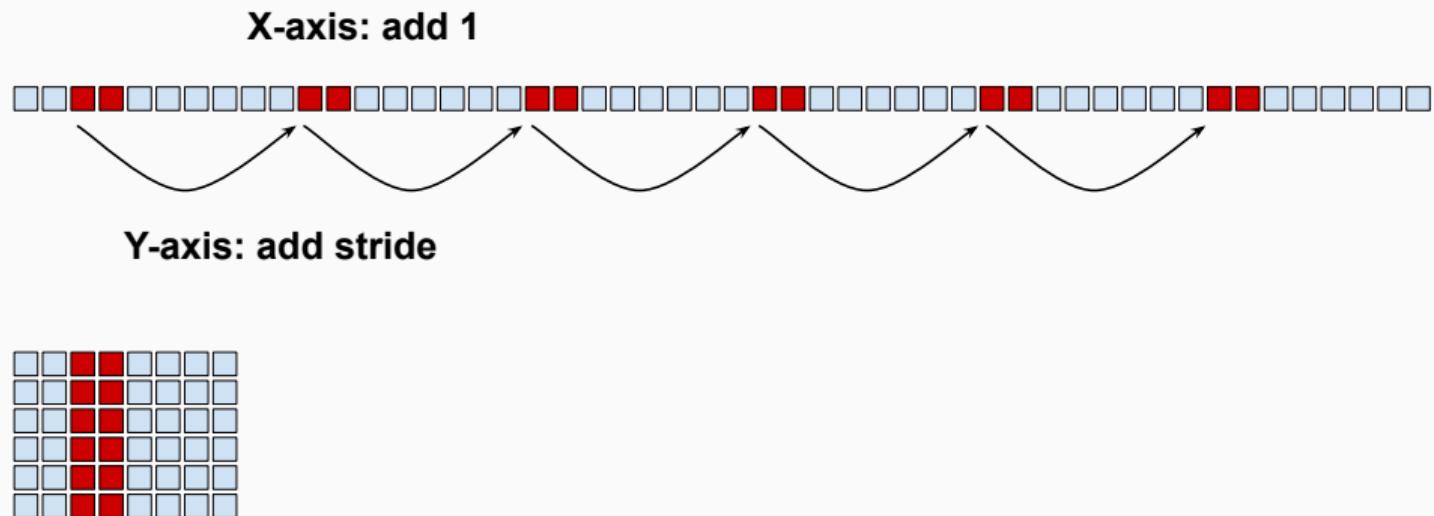
There are four things we can do:

- eliminate the unnecessary check
- move some multiplication out of the loop
- *strength reduction*: replace multiplication with addition *in the loop*
- provided that we reorder the loops (y then x)

We take advantage of the layout of pixels in the image
as well as the knowledge that we're filling in a square

Strength reduction

Pixels are kept in a slice, one row after another



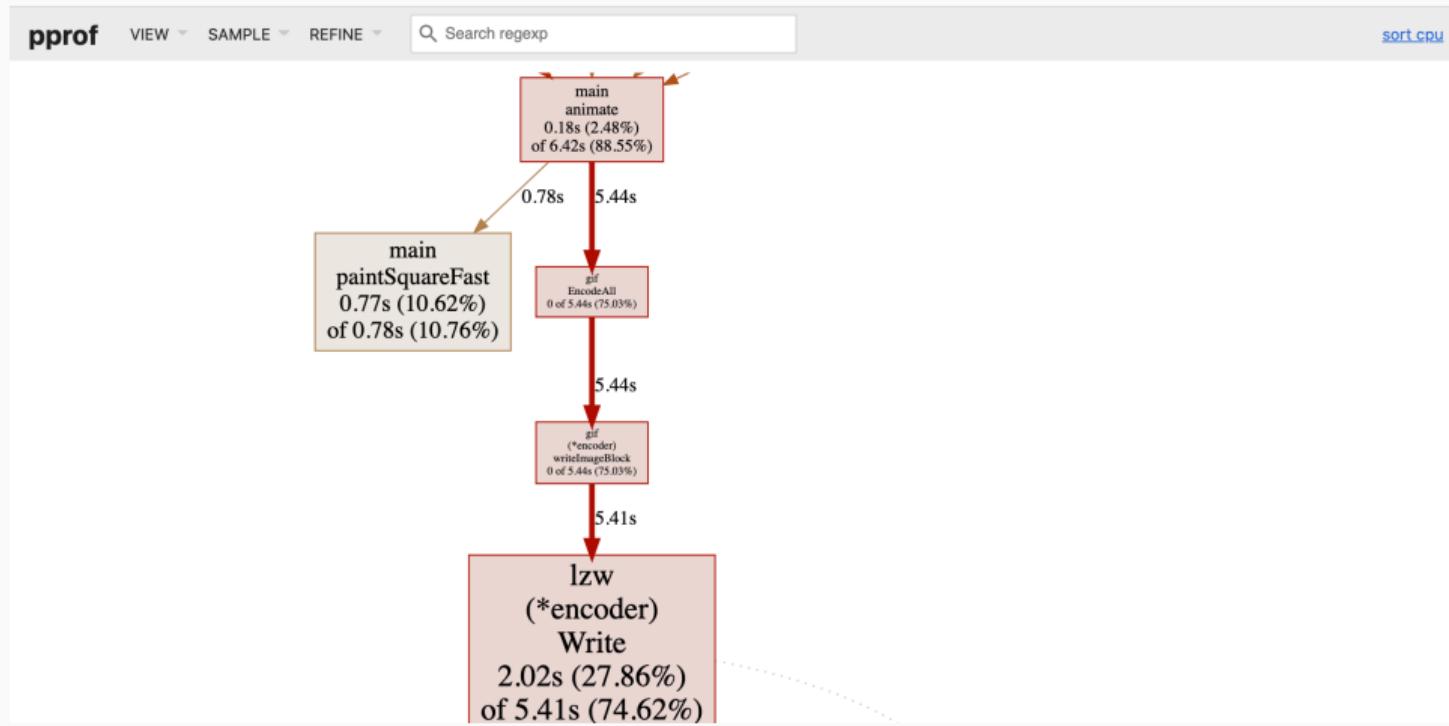
Painting each square #2

```
func paintSquareFast(i, k int, src []int, img *image.Paletted) {
    ci := uint8(src[i])
    is, ks := i * scale, k * scale
    px := (ks-img.Rect.Min.Y)*img.Stride + (is-img.Rect.Min.X)*1

    for y := 0; y < scale; y++ {
        for x := 0; x < scale; x++ {
            idx := ci

            if x == 0 || y == 0 || x == scale-1 || y == scale-1 {
                idx = 0
            }
            img.Pix[px+x] = idx
        }
        px += img.Stride
    }
}
```

CPU profile #2



CPU profile #2

```
$ go tool pprof -top sort profile-fast
File: sort
Type: cpu
Time: Jan 31, 2021 at 8:31am (MST)
Duration: 30s, Total samples = 7.25s (24.17%)
Showing nodes accounting for 7.11s, 98.07% of 7.25s total
Dropped 40 nodes (cum <= 0.04s)

      flat  flat%  sum%      cum  cum%
  2.93s 40.41% 40.41%  2.93s 40.41%  syscall.syscall
  2.02s 27.86% 68.28%  5.41s 74.62% compress/lzw.(*encoder).Write
  0.77s 10.62% 78.90%  0.78s 10.76% main.paintSquareFast
  0.35s  4.83% 83.72%  0.35s  4.83% runtime.nanotime1
  0.30s  4.14% 87.86%  3.37s 46.48% compress/lzw.(*encoder).writeLSB
  0.18s  2.48% 90.34%  6.42s 88.55% main.animate
```

CPU profile #2

pprof [VIEW](#) [SAMPLE](#) [REFINE](#) Search regexp [sort cpu](#)

main.paintSquareFast
/Users/mholiday/sort/sort.go

Total:	1.55s	1.58s (flat, cum)	11.69%
64	.	.	
65	.	.	
66	.	.	
67	.	.	
68	.	.	
69	10ms	10ms	
70	.	.	
71	.	.	
72	.	.	
73	.	.	
74	.	.	
75	10ms	20ms	
76	.	.	
77	.	10ms	
78	660ms	660ms	
79	.	.	
80	.	.	
81	10ms	20ms	
82	.	.	
83	.	.	
84	.	.	
85	830ms	830ms	
86	.	.	
87	.	.	
88	30ms	30ms	

```
func paintSquareFast(i, k int, src []int, img *image.Paletted) {
    // lay down a square with an outline using the default
    // color (gray; we deliberately excluded it from the data)

    ci := uint8(src[i])
    is, ks := i*scale, k*scale
    px := (ks-img.Rect.Min.Y)*img.Stride + (is-img.Rect.Min.X)*1

    for y := 0; y < scale; y++ {
        for x := 0; x < scale; x++ {
            idx := ci

            if x == 0 || y == 0 || x == scale-1 || y == scale-1 {
                idx = 0
            }

            img.Pix[px+x] = idx
        }
    }

    px += img.Stride
}
```

Optimize the function, part deux

There are two more things we can do:

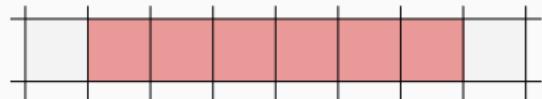
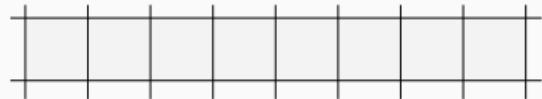
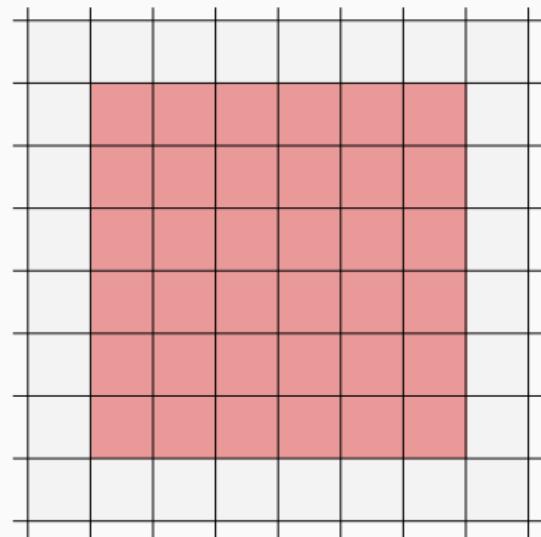
- eliminate the if-then logic in the loop
- reduce slice indexing (& bounds checks) from $O(n^2)$ to $O(n)$

We can do this by

- splitting the square into top, middle, bottom sections
- copying a slice of pixels of the correct color into the image

Painted squares

We only have two kinds of rows to paint



Painting each square #3

```
func paintSquareFastest(i, k int, src []int, img *image.Paletted) {
    ci := uint8(src[i])
    is, ks := i*scale, k*scale
    px := (ks-img.Rect.Min.Y)*img.Stride + (is-img.Rect.Min.X)*1
    py := px + (scale-1)*img.Stride

    // create a row for the top and bottom

    rw := make([]uint8, scale)

    // paint the top & bottom rows of the square

    copy(img.Pix[px:px+scale], rw)
    copy(img.Pix[py:py+scale], rw)

    . . .
```

Painting each square #3

```
    . . .

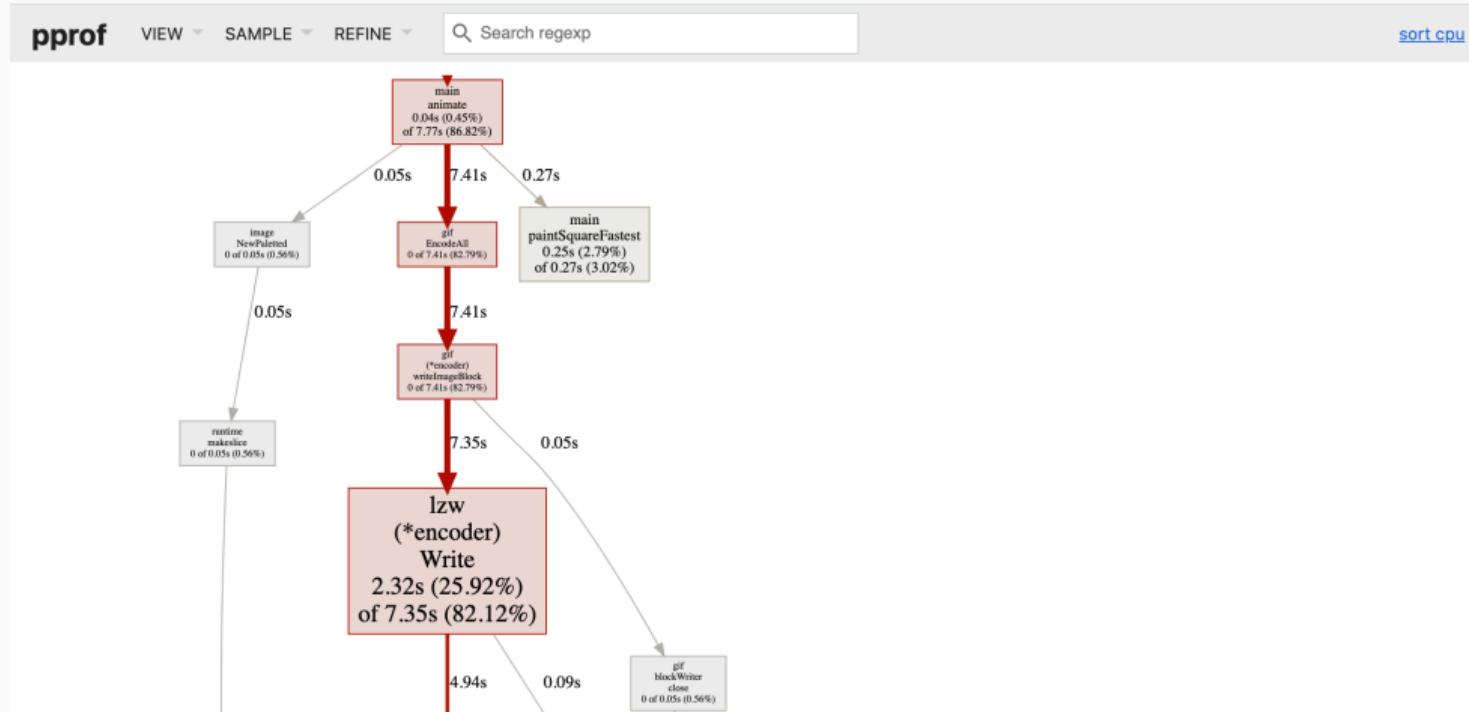
// fill in the middle part of the row

for x := 1; x < scale-1; x++ {
    rw[x] = ci
}

// paint the middle part of the square

for y := 1; y < scale-1; y++ {
    px += img.Stride
    copy(img.Pix[px:px+scale], rw)
}
}
```

CPU profile #3



CPU profile #3

```
$ go tool pprof -top sort /Users/mholiday/Downloads/profile-fastest-3
File: sort
Type: cpu
Time: Jan 31, 2021 at 12:40pm (MST)
Duration: 30.11s, Total samples = 8.95s (29.72%)
Showing nodes accounting for 8.80s, 98.32% of 8.95s total
Dropped 44 nodes (cum <= 0.04s)

      flat  flat%  sum%          cum  cum%           file
  4.52s 50.50% 50.50%      4.53s 50.61%  syscall.syscall
  2.32s 25.92% 76.42%      7.35s 82.12% compress/lzw.(*encoder).Write
  0.35s  3.91% 80.34%      0.35s  3.91% runtime.kevent
  0.32s  3.58% 83.91%      0.32s  3.58% runtime.nanotime1
  0.25s  2.79% 86.70%      0.27s  3.02% main.paintSquareFastest
  0.24s  2.68% 89.39%      4.94s 55.20% compress/lzw.(*encoder).writeLSB
  0.15s  1.68% 91.06%      4.70s 52.51% image/gif.blockWriter.WriteByte

  ...
  0.04s  0.45% 97.32%      7.77s 86.82% main.animate
```

CPU profile #3

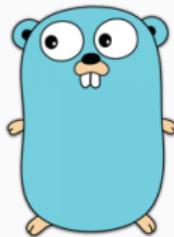
pprof [VIEW](#) [SAMPLE](#) [REFINE](#) [Search regexp](#) [sort cpu](#)

main.paintSquareFastest
/Users/mholiday/sort/sort.go

Total:	250ms	270ms (flat, cum)	3.02%
87	.	.	
88	.	.	px += img.Stride
89	.	.	}
90	.	.	}
91	.	.	
92	.	.	func paintSquareFastest(i, k int, src []int, img *image.Paletted) {
93	.	.	ci := uint8(src[i])
94	.	10ms	is, ks := i*scale, k*scale
95	.	.	px := (ks-img.Rect.Min.Y)*img.Stride + (is-img.Rect.Min.X)*1
96	.	.	PY := px + (scale-1)*img.Stride
97	.	.	rw := make([]uint8, scale)
98	.	.	
99	60ms	60ms	copy(img.Pix[px:px+scale], rw)
100	30ms	30ms	copy(img.Pix[py:py+scale], rw)
101	.	.	
102	20ms	20ms	for x := 1; x < scale-1; x++ {
103	.	.	rw[x] = ci
104	.	.	}
105	.	.	
106	.	10ms	for y := 1; y < scale-1; y++ {
107	.	.	px += img.Stride
108	140ms	140ms	copy(img.Pix[px:px+scale], rw)
109	.	.	}
110	.	.	}
111	.	.	

Programming in Go

Matt Holiday
Christmas 2020



Static Analysis

Reading culture

“So much has been said, about the importance of readability, not just in Go, but all programming languages. People like me . . . use words like simplicity, readability, clarity, productivity, but ultimately they are all synonyms for one word — *maintainability*.”

“Go is not a language that optimises for clever one liners. Go is not a language which optimises for the least number of lines in a program. We’re not optimising for the size of the source code on disk. . . . Rather, **we want to optimise our code to be clear to the reader**. Because it’s the reader who’s going to have to maintain this code.” — Dave Cheney

Static analysis

“Static” means the program isn’t running (“compile time”)

Static analysis transfers effort from people to tools

- mental effort while coding
- code review effort

Static analysis improves code hygiene

- correctness
- efficiency
- readability
- maintainability

Start clean, stay clean

Static analysis allows us to find many issues beyond compiler bugs as well as meet community guidelines for “clean code”

If our code compiles & passes static analysis, we can have a lot of confidence in it
even before running unit tests

I run these tools in my IDE every time I save a file:

- format the code
- fix the imports
- look for issues

Gofmt and Goimports

`gofmt` will put your code in standard form (spacing, indentation)

`goimports` will do that and also update import lists

Having a canonical code format is an important part of good software engineering

**The standard practice is to run one or the other on every save in your IDE/editor
(as a [save file hook](#))**

They can also be run as a [pre-commit hook](#) in your local repo

Golint

`golint` will check for non-format style issues, for example:

- exported names should have comments for godoc
- names shouldn't have `under_scores` or be in ALLCAPS
- `panic` shouldn't be used for normal error handling
- the error flow should be indented, the happy path not
- variable declarations shouldn't have redundant type info

The “rules” are based on [Effective Go](#) and Google’s [Go Code Review Comments](#)

Go vet

`go vet` will find some issues the compiler won't

- suspicious “printf” format strings
- accidentally copying a mutex type
- possibly invalid integer shifts
- possibly invalid atomic assignments
- possibly invalid struct tags
- unreachable code

No static analysis tool can find all possible errors

Other tools

`goconst` finds literals that should be declared with `const`

`gosec` looks for possible security issues

`ineffasign` finds assignments that are “ineffective” (shadowed?)

`gocyclo` reports high **cyclomatic complexity** in functions

`deadcode`, `unused`, and `varcheck` find unused/dead code

`unconvert` finds redundant type conversions

I treat some of these as **warnings** because there are false positives

Example: ineffassign

The first assignment is ineffective because it's overwritten without being read (which means we missed handling the error)

```
prices, err := r.prices(region, . . . )
regularPrices, err := r.regularPrices(region, . . . )    // probably added later

if err != nil {
    return nil,
    fmt.Errorf("price not available for region %s", region)
}

// ineffectual assignment to err
```

Example: govet

The format string is mismatched

```
func main() {
    fmt.Printf("%s\n", 20)
}

// Printf format %s has arg 20 of wrong type int
```

Example: golint

The formatting of an error message uses bad style

```
if !ok {
    return fmt.Errorf("id is not a string: %v\n", idRef)
}

// error strings should not be capitalized or end
// with punctuation or a newline
```

Example: gosimple

The code in question may be simplified

```
// should merge variable declaration with assignment on
// next line, i.e., var responseData = data.Data

var responseData []data
responseData = response.Data

// should omit comparison to bool constant

if reservation[i].NonExpiring == true {
    . . .
}
```

One tool to rule them all

We run all these tools using `golangci-lint`

It can be configured with `.golangci.yml`

We use this in our CI/CD pipeline

Issues must be fixed for the build to pass

False positives can be marked with `//nolint`

Visual Studio Code

A screenshot of the Visual Studio Code interface. The main area shows a Go file named `say.go` with the following code:

```
41     },
42 }
43
44 for _, st := range subtests {
45     if s := Say(st.items...); s != st.result {
46         t.Errorf("got %s, gave %v, wanted %s", s, st.items, st.result)
47     }
48 }
49
50
51 func main() {
52     fmt.Println(Say(os.Args...))
53 }
54
```

The code editor has syntax highlighting for Go. A tooltip or status bar at the bottom indicates "Ln 53, Col 1 (30 selected)".

Below the code editor is a terminal window showing the output of a Go command:

```
[k0dvb] Go Training$ go run ./say.go Matt Anne
Hello, Matt, Anne!
[09:59:25]
[09:59:33]
```

The terminal window shows two entries: the first entry is timestamped [09:59:25] and the second is [09:59:33].

Sample VSC settings

```
{  
    "go.vetOnSave": "package",  
    "go.formatTool": "goimports",  
    "go.formatFlags": [  
        "-local github.com/xxx,github.com/yyy"  
    ],  
    "go.lintTool": "golangci-lint",  
    "go.lintFlags": [  
        "--fast" ]  
    "go.lintOnSave": "package"  
}
```

Jetbrains GoLand IDE

The screenshot shows the Jetbrains GoLand IDE interface. The title bar reads "offers-inventory-management - cache.go". The main window displays the "cache.go" file with the following code:

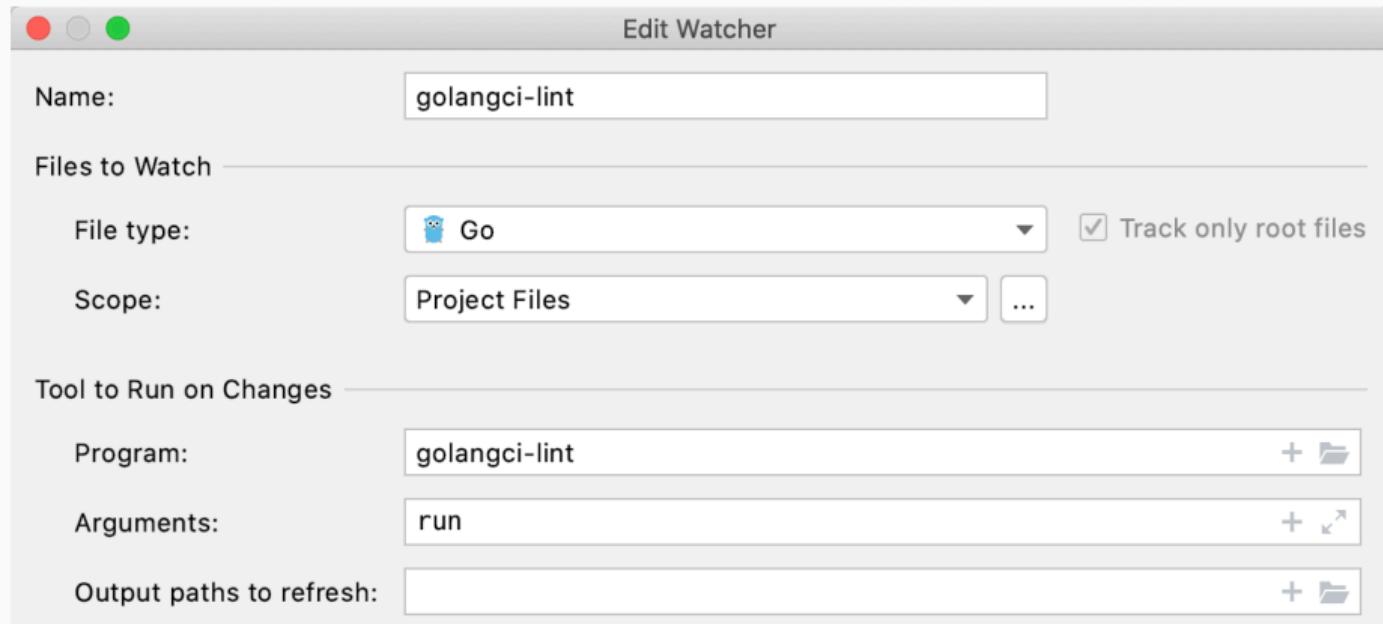
```
offers-inventory-management - cache.go
cache.go

21 // cachedPlatoClient is a plato client with responses cached.
22 type cachedPlatoClient struct {
23     plato
24     logger *leveledLog
25 }
26
27 var {
28     defaultCountries = []string{"US", "CA", "GB", "FR", "AU", "IN"}
29
30     // these are variables so we can reduce them in unit tests
31
32     baseLifetime = 5 * time.Hour    // how long a cache entry lives (randomized later)
33     reloadTime   = 20 * time.Minute // time considered "soon to expire", i.e., reload it now
34     checkTime    = reloadTime / 2  // time between periodic DB/cache checks
35
36     // these are used to create a single connection pool for Redis
37
38     pool *redis.Pool
39     once sync.Once
40 }
41
42 func newCachedPlatoClient(logger *leveledLog, client *platoClient) (*cachedPlatoClient, error) {
43     // we will start the pool and test our connection
44     // before we say we've got a working caching client
45
46     if err := redisTest(getConn()); err != nil {
47         return nil, fmt.Errorf("redis failed: %w", err)
48     }
49 }
```

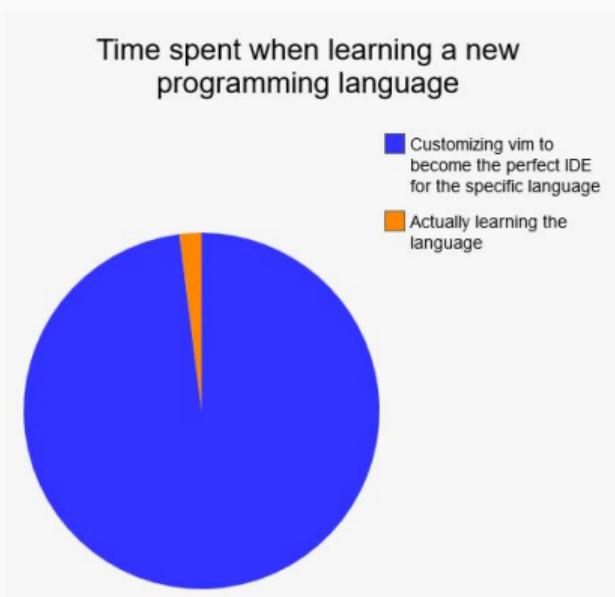
The left sidebar shows the project structure under "Project" mode, with several files listed in green, indicating they are part of the current workspace. The bottom navigation bar includes tabs for "File", "Edit", "View", "Tools", "File", "Git", "TODO", and "Terminal". The status bar at the bottom right shows "Event Log", "27-1 LF UTF-8 Tab", and "offers-370-error".

Sample GoLand settings

The formatting and linting tools are configured as file watchers in GoLand



Vim



Vim setup example

Programming in Go

Matt Holiday
Christmas 2020



Testing in Go

Go test features

Go has standard tools and conventions for testing

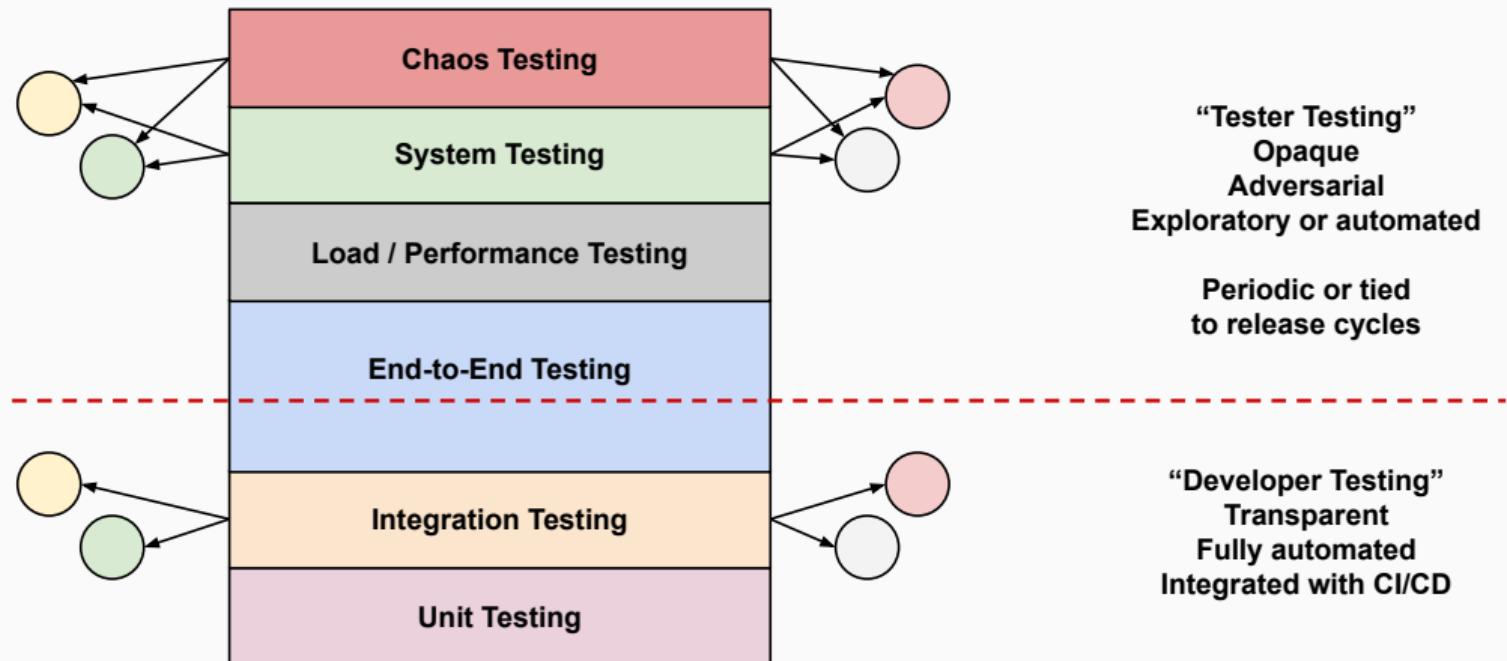
Test files end with `_test.go` and have `TestXXX` functions
(they can be in the package directory, or in a separate directory)

You run tests with `go test`

```
go test ./...
ok    xyz/test        56.841s
ok    xyz/pkg/acedb (cached)
```

Tests aren't run if the source wasn't changed since the last test

Layers of testing



Goals

Things to test for

- extreme values
- input validation
- race conditions
- error conditions
- boundary conditions
- pre- and post-conditions
- randomized data (fuzzing)
- configuration & deployment
- interfaces to other software

Test functions

Test functions have the same signature using `testing.T`

```
func TestCrypto(t *testing.T) {
    uuid := "650b5cc5-5c0b-4c00-ad97-36b08553c91d"
    key1 := "75abbabc1f9f8d28d55200b43fd95962"
    key2 := "75abbabc1f9f8d28d66200b43fd95962"

    ct, err := secrets.MakeAppKey(key1, uuid)

    if err != nil {
        t.Errorf("make failed: %s", err)
    }
    .
}
```

Errors are reported through parameter `t` and fail the test

Table-driven tests

```
func TestValueFreeFloat(t *testing.T) {
    table := []struct {
        v float64
        s string
    }{
        {1, "1"},
        {1.1, "1.1"},
    }

    for _, tt := range table {
        v := Value{T: floater, V: tt.v, m: &Machine{}}

        if s := v.String(); s != tt.s {
            t.Errorf("%v: wanted %s, got %s", tt.v, tt.s, s)
        }
    }
}
```

Table-driven subtests

We can run *subtests* under the parent using `t.Run()`

```
func TestGraphqlResolver(t *testing.T) {
    table := []subTest{
        name string
        .
        .
        .
    }{
        name: "retrieve_offer",
        .
        .
    }

    for _, st := range table {
        t.Run(st.name, func(t *testing.T) {           // closure
            .
            .
        })
    }
}
```

A complex unit test example

```
func TestScanner(t *testing.T) {
    scanTests := []struct{
        name  string
        input string
        want  []token.Token
    }{
        {
            name:  "simple-add",
            input: "2 1 +",
            want: []token.Token{
                {Type: token.Number, Line: 1, Text: "2"},
                {Type: token.Number, Line: 1, Text: "1"},
                {Type: token.Operator, Line: 1, Text: "+"},
            },
        },
        ...
    }
}
```

A complex unit test example

```
for _, st := range scanTests {
    t.Run(st.name, func(t* testing.T) {
        b := bytes.NewBufferString(st.input)
        s := NewScanner(ScanConfig{}, st.name, b)

        var got []token.Token

        for tok := s.Next(); tok.Type != token.EOF; tok = s.Next() {
            got = append(got, tok)
        }

        if !reflect.DeepEqual(st.want, got) {
            t.Errorf("line %q, wanted %v, got %v", st.input, st.want, got)
        }
    })
}
```

A complex unit test refactored

```
type scanTest struct {
    name  string
    input string
    want  []token.Token
}
```

A complex unit test refactored

```
func (st scanTest) run(t *testing.T) {
    b := bytes.NewBufferString(st.input)
    c := ScanConfig{}
    s := NewScanner(c, st.name, b)

    var got []token.Token

    for tok := s.Next(); tok.Type != token.EOF; tok = s.Next() {
        got = append(got, tok)
    }

    if !reflect.DeepEqual(st.want, got) {
        t.Errorf("line %q, wanted %v, got %v", st.input, st.want, got)
    }
}
```

A complex unit test refactored

```
var scanTests = []scanTest{
{
    name: "simple-add-comma",
    input: "2 1 +, 3+",
    want: []token.Token{
        {Type: token.Number, Line: 1, Text: "2"},
        {Type: token.Number, Line: 1, Text: "1"},
        {Type: token.Operator, Line: 1, Text: "+"},
        {Type: token.Comma, Line: 1},
        {Type: token.Number, Line: 2, Text: "3"},
        {Type: token.Operator, Line: 2, Text: "+"},
    },
},
. . .
}
```

A complex unit test refactored

```
func TestScanner(t *testing.T) {
    for _, st := range scanTests {
        t.Run(st.name, st.run)          // method value = func(*testing.T)
    }
}
```

More refactoring

```
type checker interface {
    check(*testing.T, string, string) bool
}

type subTest struct {
    name      string
    shouldFail bool
    checker   checker // parameterize how we check results
    ...
}

// we can now define different checker types
type checkGolden() struct { . . . }

func (c checkGolden) check(t *testing.T, got, want string) bool {
    ...
}
```

Mocking or faking

```
type DB interface {
    GetThing(string) (thing, error)
    ...
}

type mockDB struct {
    shouldFail bool
}

var errShouldFail = errors.New("db should fail")

func (m mockDB) GetThing(key string) (thing, error) {
    if m.shouldFail {
        return thing{}, fmt.Errorf("%s: %w", key, errShouldFail)
    }
    ...
}
```

Main test functions

You can define a root function for all testing; it will then run all tests from this point

```
func TestMain(m *testing.M)
    stop, err := startEmulator()

    if err != nil {
        log.Println("*** FAILED TO START EMULATOR ***")
        os.Exit(-1)
    }

    result := m.Run() // run all UTs

    stop()
    os.Exit(result)
}
```

Special test-only packages

If you need to add test-only code as part of a package, you can place it in a package that ends in `_test`

That package, like `XXX_test.go` files, will not be included in a regular build

Unlike normal test files, it will only be allowed to access *exported* identifiers, so it's useful for “opaque” or “black-box” tests

```
// file myfunc_test.go

package myfunc_test
// this package is not part of package myfunc, so
// it has no internal access
```

See this [StackOverflow answer](#)

Philosophy of Testing

Testing culture

“Your tests are the contract about what your software does and does not do. Unit tests at the package level should lock in the behaviour of the package’s API. They describe, in code, what the package promises to do. If there is a unit test for each input permutation, you have defined the contract for what the code will do in code, not documentation.”

“This is a contract you can assert as simply as typing `go test`. At any stage, you can know with a high degree of confidence, that the behaviour people relied on before your change continues to function after your change.” — Dave Cheney

Testing culture

You should assume your code doesn't work *unless*

- you have tests (unit, integration, etc.)
- they work correctly
- you run them
- they pass

Your work isn't done until you've added or updated the tests

This is basic code hygiene: **start clean, stay clean**

Psychology of computer programming

“The hardest bugs are those where your mental model of the situation is just wrong, so you can't see the problem at all.” — Brian Kernighan

This issue applies to testing also

In general, **developers test to show that things are done & working according to their understanding of the problem & solution**

Most difficulties in software development are *failures of the imagination*

"I never thought they would press it"



Program correctness

There are eight levels of correctness “in order of increasing difficulty of achievement” (Gries & Conway)

1. it compiles [and passes static analysis]
2. it has no bugs that can be found just running the program
3. it works for some hand-picked test data
4. it works for typical, reasonable input
5. it works with test data chosen to be difficult
6. it works for all input that follows the specifications
7. it works for all valid inputs and likely error cases
8. **it works for all input**

“It works” means it produces the desired behavior or fails safely

Program correctness

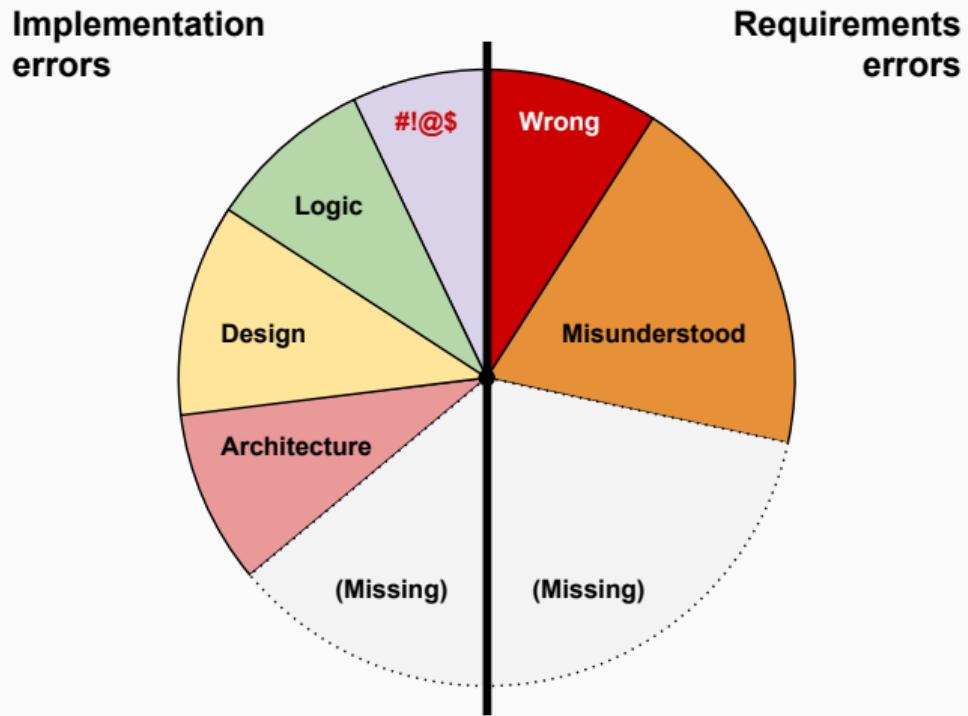
There are four distinct types of errors (Gries & Conway):

1. errors in understanding the problem requirements
2. errors in understanding the programming language
3. errors in understanding the underlying algorithm
4. errors where you knew better but simply slipped up
(even experienced developers make mistakes)

“Type 1 errors tend to increase as problems become larger, more varied, and less precisely stated.”

Even worse, some requirements may just be **missing**

Sources of errors



Developer testing is necessary

You should aim for 75-85% code coverage

- unit tests
- integration tests
- post-deployment sanity checks

Developers **must** be responsible for the quality of their code

They shouldn't just “throw crap over the wall”

Tests can be part of your documentation

Testing is not “quality assurance”

Confusing “test” and “QA” is a basic mistake

- QA is a different discipline in software development
- we’re not dealing with a manufacturing process
- you can’t “test in” or prove quality

Testing is not about running “acceptance” tests to show that things work

It’s about surfacing defects by causing the system to fail (breaking it)

The *wrong testing mindset* leads to inadequate testing

Developer testing isn't enough

You can have 100% code coverage and still be wrong

- the code may be bug-free, but not match the requirements
- the requirements may not match expectations
- you can't test code that's missing

Testers test to show that things *don't* work

But they can't test your system well if the requirements aren't documented
(this is a major limitation of the agile method *as practiced**)

Code & unit tests are simply not enough documentation

Reality check

Pick any two

- good
- fast
- cheap

You can't have all three in the real world

Effective and thorough testing is hard & expensive

Software is annoying because most orgs pick fast and cheap over good

Programming in Go

Matt Holiday
Christmas 2020



Code Coverage

Testing culture

You should assume your code doesn't work unless

- you have tests (unit, integration, etc.)
- they work correctly
- you run them
- they pass

Your change isn't done until you've added / updated the tests

The correct order is compile, format, lint, test and *then* commit

This is basic code hygiene: **start clean, stay clean**

Testing JSON

```
func contains(known, unknown map[string]interface{}) error {
    for k, v := range known {
        switch x := v.(type) {
        case float64:
            if !matchNum(k, x, unknown) {
                return fmt.Errorf("%s unmatched (%d)",
                    k, int(x))
            }
        case string:
            if !matchString(k, x, unknown) {
                return fmt.Errorf("%s unmatched (%s)", k, x)
            }
        ...
    }
}
```

Testing JSON

```
    . . .
case map[string]interface{}:
    if val, ok := unknown[k]; !ok {
        return fmt.Errorf("%s missing in resp", k)
    } else if unk, ok := val.(map[string]interface{}); ok {
        if err := contains(x, unk); err != nil {
            return fmt.Errorf("%s unmatched (%+v): %s",
                               k, x, err)
        }
    } else {
        return fmt.Errorf("%s wrong in resp (%#v)",
                           k, val)
    }
}
}

return nil
}
```

Testing JSON

```
func CheckData(known string, unknown []byte) error {
    var k, u map[string]interface{}

    if err := json.Unmarshal([]byte(known), &k); err != nil {
        return err
    }

    if err := json.Unmarshal(unknown, &u); err != nil {
        return err
    }

    return contains(k, u)
}
```

Testing JSON

Run the tests and analyze the code coverage

```
// go test -v
// go test ./... -cover
// go test ./... -coverprofile=c.out -covermode=count
// go tool cover -html=c.out

var unknown = `{
    "id": 1,
    "name": "bob",
    "addr": {
        "street": "Lazy Lane",
        "city": "Exit",
        "zip": "99999"
    },
    "extra": 21.1
}`
```

Testing JSON

```
func TestContains(t *testing.T) {
    var known = []string{
        `{"id": 1}`,
        `{"extra": 21.1}`,
        `{"name": "bob"} `,
        `{"addr": {"street": "Lazy Lane", "city": "Exit"}}`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err != nil {
            t.Errorf("invalid: %s (%s)\n", k, err)
        }
    }
}
```

Testing JSON

```
func TestNotContains(t *testing.T) {
    var known = []string{
        `{"id": 2}`,
        `{"pid": 2}`,
        `{"name": "bobby"} `,
        `{"first": "bob"} `,
        `{"addr": {"street": "Lazy Lane", "city": "Alpha"} }`,
    }

    for _, k := range known {
        if err := CheckData(k, []byte(unknown)); err == nil {
            t.Errorf("false positive: %s\n", k)
        } else {
            t.Log(err)
        }
    }
}
```

Code coverage

Running `go test -cover` finds what part of the code is exercised by the unit tests

```
$ go test -cover  
PASS  
coverage: 85.2% of statements
```

Using the `-coverprofile` flag generates a file with coverage counts

This can be passed to another tool to display coverage visually

```
$ go tool cover -html=coverage.out
```

Using the `-covermode=count` flag turns it into a heat map

Code coverage

```
/Users/mholiday/tmp/json/main.go (85.2%) not tracked no coverage low coverage * * * * * * * * * * high coverage
    case float64:
        if !matchNum(k, x, data) {
            return fmt.Errorf("%s unmatched (%d)", k, int(x))
        }

    case string:
        if !matchString(k, x, data) {
            return fmt.Errorf("%s unmatched (%s)", k, x)
        }

    case map[string]interface{}:
        if val, ok := data[k]; !ok {
            return fmt.Errorf("%s missing in data", k)
        } else if unk, ok := val.(map[string]interface{}); ok {
            if err := contains(x, unk); err != nil {
                return fmt.Errorf("%s unmatched (%+v): %s", k, x, err)
            }
        } else {
            return fmt.Errorf("%s wrong in data (%#v)", k, val)
        }
    }

    return nil
}

func CheckData(want, got []byte) error {
    var w, g map[string]interface{}

    if err := json.Unmarshal(want, &w); err != nil {
        return err
    }

    if err := json.Unmarshal(got, &g); err != nil {
        return err
    }

    if !reflect.DeepEqual(w, g) {
        return fmt.Errorf("want: %v\ngot: %v", want, got)
    }
}
```

Code coverage

The heat map shows two cases we haven't covered:

- The case where the key is missing
- The case where it has the wrong type (not an object)

We need to add some more subtests to cover this code

```
$ go test ./... -coverprofile=c.out -covermode=count
ok    _/Users/mholiday/tmp/json 0.173s coverage: 92.6% of statements

$ go tool cover -html=c.out
```

Testing JSON

```
func TestNotContains(t *testing.T) {
    var known = []string{
        `{"id": 2}`,
        `{"pid": 2}`,
        `{"name": "bobby"}  
`,
        `{"first": "bob"}  
`,
        `{"addr": {"street": "Lazy Lane", "city": "Alpha"}  
`},
        `{"city": {"avenue": "Lazy Ave"}  
`}, // missing
        `{"name": {"avenue": "Lazy Ave"}  
`}, // wrong
    }
    . . .
}
```

Code coverage

```
/Users/mholiday/tmp/json/main.go (92.6%) not tracked no coverage low coverage * * * * * * * * * * * high coverage
case TFloat64:
    if !matchNum(k, x, data) {
        return fmt.Errorf("%s unmatched (%d)", k, int(x))
    }

case string:
    if !matchString(k, x, data) {
        return fmt.Errorf("%s unmatched (%s)", k, x)
    }

case map[string]interface{}:
    if val, ok := data[k]; !ok {
        return fmt.Errorf("%s missing in data", k)
    } else if unk, ok := val.(map[string]interface{}); ok {
        if err := contains(x, unk); err != nil {
            return fmt.Errorf("%s unmatched (%#v): %s", k, x, err)
        }
    } else {
        return fmt.Errorf("%s wrong in data (%#v)", k, val)
    }
}

return nil
}

func CheckData(want, got []byte) error {
    var w, g map[string]interface{}

    if err := json.Unmarshal(want, &w); err != nil {
        return err
    }

    if err := json.Unmarshal(got, &g); err != nil {
        return err
    }

    if w != g {
        return fmt.Errorf("want: %v\ngot: %v", w, g)
    }
}
```

Programming in Go

Matt Holiday
Christmas 2020



Go Modules

Why modules?

Go module support is intended to solve several problems:

- avoid the need for \$GOPATH
- group packages versioned/released together
- support semantic versioning & backwards compatibility
- provide in-project dependency management
- **offer strong dependency security & availability**
- continue support of vendoring
- **work transparently across the Go ecosystem**

Go modules with proxying offers the value of vendoring without requiring your project to vendor all the 3rd-party code in your repo

Why modules?

Go's dependency management protects against some risks:

- flaky repos
- packages that disappear
- conflicting dependency versions
- surreptitious changes to public packages

But it cannot ensure the actual quality or security of the *original* code; see

- [Reflections on Trusting Trust](#) by Ken Thompson
- [Our Software Dependency Problem](#) by Russ Cox

“A little copying is better than a little dependency” — Go Proverb

Import compatibility rule

“If an old package and a new package have the same import path,
the new package must be backwards compatible with the old package”

An *incompatible* updated package should use a new URL (version)

```
package hello

import (
    "github.com/x"
    x2 "github.com/x/v2"
)
```

Note that you can import both versions if necessary

Some control files

The `go.mod` file has your module name along with direct dependency requirements (and from Go 1.13, the version of Go)

```
module hello
require github.com/x v1.1
go 1.13
```

The `go.sum` file has checksums for all *transitive* dependencies

```
github.com/x v1.1 h1:KqKTd5BnrG8aKH3J...
github.com/y v0.2 h1:Qz0iS0pjZuFQy/z7...
github.com/z v1.5 h1:r8zfno3MHue2Ht5s...
```

Always check them in to your repo

Some environment variables

We typically use the defaults for these

```
GOPROXY=https://proxy.golang.org,direct  
GOSUMDB=sum.golang.org
```

and set this for private repos

```
GOPRIVATE=github.com/xxx,github.com/yyy  
GONOSUMDB=github.com/xxx,github.com/yyy
```

Remember also you must be set up for access to private Github repos in order to download private modules

Module Proxy

The `go.mod` file records direct dependencies, while `go.sum` records checksums for all (transitive) dependencies

The proxy caches modules and keeps a secure history tree



Some details

The `go.mod` file may record **pseudo-versions** (for non-release/trunk versions) as well as “replacements”

```
require (
    cloud.google.com/go v0.35.1
    github.com/gen2brain/malgo v0.0.0-20181117112449-af6b9a0d538d
    github.com/gorilla/context v1.1.1 // indirect
    github.com/gorilla/mux v1.6.2
    github.com/gorilla/websocket v1.4.0
    github.com/satori/go.uuid v1.2.0
    golang.org/x/net v0.0.0-20190119204137-ed066c81e75e
    google.golang.org/api v0.1.0
    google.golang.org/genproto v0.0.0-20190123001331-8819c946db44
    google.golang.org/grpc v1.18.0
    gopkg.in/yaml.v2 v2.2.2
)

replace github.com/satori/go.uuid v1.2.0 =>
    github.com/satori/go.uuid v1.2.1-0.20181028125025-b2ce2384e17b
```

Maintaining dependencies

Start a project with

```
$ go mod init <module-name>    ## create the go.mod file  
$ go build                      ## building updates go.mod
```

Once a version is set, Go will not update it automatically; you can update every dependency with

```
$ go get -u ./...                ## update transitively  
$ go mod tidy                     ## remove unneeded modules
```

You **must commit** the `go.mod` and `go.sum` files in your repo

Maintaining dependencies

You can list available versions of a dependency

```
$ go list -m -versions rsc.io/sampler
```

There are several ways to update a single dependency

```
$ go get github.com/gorilla/mux@latest
$ go get github.com/gorilla/mux@v1.6.2
$ go get github.com/gorilla/mux@e3702bed2
$ go get github.com/gorilla/mux@c856192      ## non-release
$ go get github.com/gorilla/mux@master        ## non-release
```

You **must commit** the `go.mod` and `go.sum` files in your repo

Vendoring and the local cache

Use `go mod vendor` to create the vendor directory; it must be in the module's root directory (along with `go.mod`)

In Go 1.13, you must use `go build -mod=vendor` to use it
(not required 1.14+)

Go keeps a local cache in `$GOPATH/pkg`

- each package (using a directory tree)
- the hash of the root checksum DB tree

Use `go clean -modcache` to remove it all (i.e., in `make clean`)

More info

The screenshot shows a web browser window with the following details:

- Title Bar:** Modules - golang/go Wiki
- Address Bar:** github.com/golang/go/wiki/Modules
- Toolbar:** Apps, NYT Access, Workday, Offers, GAE Docs, FunRetro, Jira, Drone, Datadog, Docs
- Page Content:**
 - ## Modules

Todd Gao edited this page 28 days ago · 351 revisions
 - ## Go Modules

Go has included support for versioned modules as proposed [here](#) since 1.11. The initial prototype `vgo` was [announced](#) in February 2018. In July 2018, versioned modules [landed](#) in the main Go repository.

In [Go 1.14](#), module support is considered ready for production use, and all users are encouraged to migrate to modules from other dependency management systems. If you are unable to migrate due to a problem in the Go toolchain, please ensure that the problem has an [open issue](#) filed. (If the issue is not on the Go1.15 milestone, please comment on why it prevents you from migrating so it can be prioritized appropriately). You can also provide an [experience report](#) for more detailed feedback.
 - ### Recent Changes
- Sidebar:**
 - Pages (166)**
 - WhyGo**
 - Home
 - WhyGo
 - Install**
 - Windows
 - Ubuntu
 - Source
- Bottom:** Clone this wiki locally, URL: <https://github.com/golang/go/wiki/Modules>

Programming in Go

Matt Holiday
Christmas 2020



Building for Distribution

Go build tools

We've been using `go run` or maybe `go test` to run programs

Now it's time to distribute

- `go build` makes a binary
- `go install` makes one and copies it to `$GOPATH/bin`

Pure Go

We can build “pure” Go programs (with some cautions):

```
$ CGO_ENABLED=0 go build -a -tags netgo,osusergo \
    -ldflags "-extldflags '-static' -s -w" \
    -o lister .
```

Here we must tell Go we’re going to use pure Go networking

A “pure” Go program can put it into a “from-scratch” container

```
$ ldd lister
    not a dynamic executable
```

Go build platforms

Go can cross-compile, too

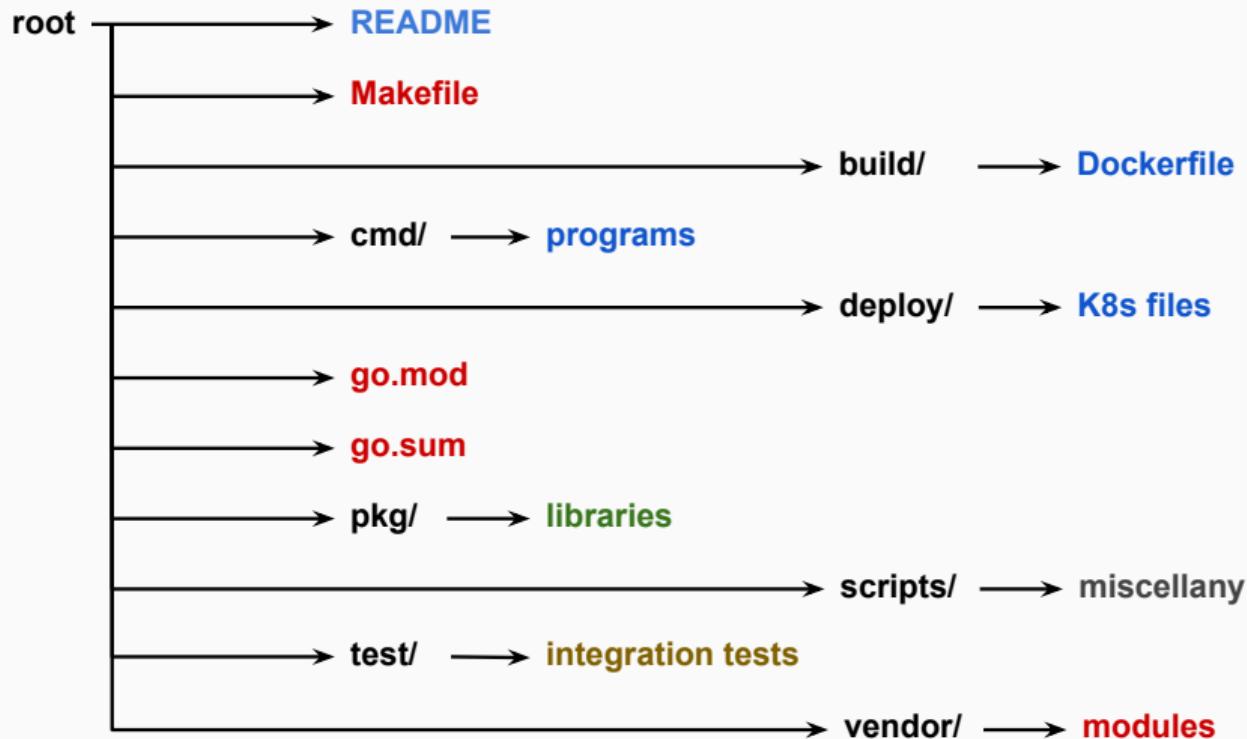
- \$GOARCH defines the architecture (e.g., amd64 or arm64)
- \$GOOS defines the operating system (e.g., linux or darwin)
- \$GOARM for the ARM chip version (v7, etc.)

We can build for the Raspberry Pi

```
$ GOOS=linux GOARCH=arm GOARM=7 CGO_ENABLED=0 \
    go build -a -tags netgo,osusergo \
    -ldflags "-extldflags '-static' -w" \
    -o mainPi ./main.go

$ file mainPi
mainPi: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
      statically linked, not stripped
```

Project layout



Documentation

Your README.md file should talk about (among other things)

- overview — who and what is it for?
- developer setup
- project & directory structure
- dependency management
- how to build and/or install it (make targets, etc.)
- how to test it (UTs, integration, end-to-end, load, etc.)
- how to run it (locally, in Docker, etc.)
- database & schema
- credentials & security
- debugging monitoring (metrics, logs)
- CLI tools and their usage

Makefiles

Reasons we might want a Makefile

- we need to calculate parameters
- we have other steps and/or dependencies
- because the options are waaaay to long to type
- and we may have non-Go commands (Docker, cloud provider, etc.)

Versioning the executable

In the main program code:

```
// MUST BE SET by go build -ldflags "-X main.version=999"  
// like 0.6.14-0-g26fe727 or 0.6.14-2-g9118702-dirty  
  
var version string // do not remove or modify
```

See [Setting compile-time variables for versioning](#)

From the makefile:

```
VERSION=$(shell git describe --tags --long --dirty 2>/dev/null)  
BRANCH=$(shell git rev-parse --abbrev-ref HEAD)  
  
xyz: $(SOURCES)  
    go build -mod=vendor -ldflags "-X main.version=$(VERSION)" -o $@ ./cmd/xyz
```

Building in Docker

We can use Docker to build as well as run

- multi-stage builds
- use a `golang` image to build it
- copy the results to another image

The result is a small Docker container built for Linux

And you can build it without even having Go installed!

This is great for CI/CD environments

Dockerfile extracts (1)

```
FROM golang:1.15-alpine AS builder

RUN /sbin/apk update && /sbin/apk --no-cache add ca-certificates \
git tzdata && /usr/sbin/update-ca-certificates

RUN adduser -D -g '' sort
WORKDIR /home/sort

COPY go.mod /home/sort
COPY go.sum /home/sort
COPY cmd    /home/sort/cmd
COPY *.go   /home/sort

ARG VERSION

RUN CGO_ENABLED=0 go build -a -tags netgo,osusergo \
-lldflags "-extldflags '-static' -s -w" \
-lldflags "-X main.version=$VERSION" -o sort ./cmd/sort
```

Dockerfile extracts (2)

```
FROM busybox:musl

COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=builder /usr/share/zoneinfo /usr/share/zoneinfo
COPY --from=builder /etc/passwd /etc/passwd
COPY --from=builder /home/sort/sort /home/sort

USER sort
WORKDIR /home
EXPOSE 8081

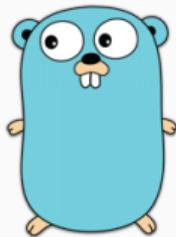
ENTRYPOINT ["/home/sort"]
```

And build it:

```
docker build -t sort-anim:latest . -f build/Dockerfile --build-arg VERSION=$(VERSION)
docker tag ${NAME}:latest ${HOST}/sort-anim:${VERSION}
docker push ${HOST}/sort-anim:${VERSION}
```

Programming in Go

Matt Holiday
Christmas 2020



Parametric Polymorphism



Generics in Go

“Generics” is shorthand for *parametric polymorphism*

That means we have a **type parameter** on a type or function

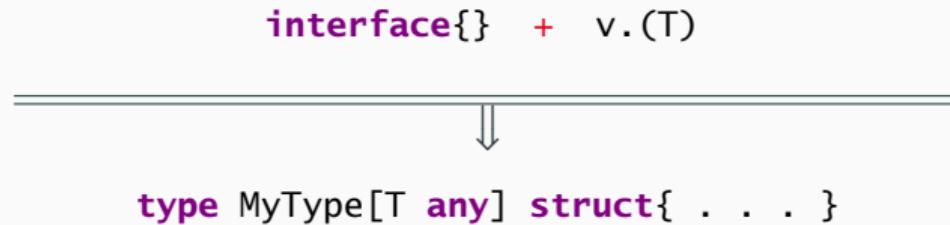
```
type MyType[T any] struct {
    v T // can be any valid Go type
    n int
}
```

Generics are a powerful feature for abstraction

And a possible source of *unnecessary* abstraction and complexity

Generics in Go

Use type parameters to **replace dynamic typing** with static typing



If it runs faster, consider that a bonus

Continue to use (non-empty) interfaces wherever possible

Performance should not be your principal reason for generics (in most cases)

Generic type & function

```
type Vector[T any] []T

func (v *Vector[T]) Push(x T) {
    *v = append(*v, x)           // may reallocate
}

// note: F and T are both used in the parameter list

func Map[F, T any](s []F, f func(F) T) []T {
    r := make([]T, len(s))

    for i, v := range s {
        r[i] = f(v)
    }

    return r
}
```

Generic type & function

```
func main() {
    v := Vector[int]{}

    v.Push(1)
    v.Push(2)

    s1 := Map(v, strconv.Itoa)
    s2 := Map([]int{1, 2, 3}, strconv.Itoa)

    fmt.Println(v)
    fmt.Printf("%#v\n", s1)
    fmt.Printf("%#v\n", s2)
}
```

Note: Map is a textbook example and not necessarily a good idea

Go 2 playground

The [go2go](#) Playground [Run](#) [Format](#) [Share](#) Hello, playground ▾

About

```
8 type Vector[T any] []T
9
10 func (v *Vector[T]) Push(x T) {
11     *v = append(*v, x)
12 }
13
14 func Map[F, T any](s []F, f func(F) T) []T {
15     r := make([]T, len(s))
16     for i, v := range s {
17         r[i] = f(v)
18     }
19     return r
20 }
21
22 func main() {
23     v := Vector[int]{}
24     v.Push(1)
25     v.Push(2)
26     fmt.Println(v)
27
28     s1 := Map(v, strconv.Itoa)
29     fmt.Printf("%#v\n", s1)
30
31     s2 := Map([]int{1, 2, 3}, strconv.Itoa)
```

```
[1 2]
[[]string{"1", "2"}]
[[]string{"1", "2", "3"}]
```

Program exited.

Generic type & method

```
type num int

func (n num) String() string {
    return strconv.Itoa(int(n))
}

// type constraint: T must have String() method

type StringableVector[T fmt.Stringer] []T
```

Generic type & method

```
func (s StringableVector[T]) String() string {
    var sb strings.Builder
    sb.WriteString("<<")
    for i, v := range s {
        if i > 0 {
            sb.WriteString(", ")
        }
        sb.WriteString(v.String())
    }
    sb.WriteString(">>")
    return sb.String()
}

func main() {
    var s StringableVector[num] = []num{1, 2, 3} // [num] required on type
    fmt.Println(s)
}
```

Go 2 playground

The [go2go](#) Playground [Run](#) [Format](#) [Share](#) Hello, playground ▾

About

```
 9 type num int
10
11 func (n num) String() string {
12     return strconv.Itoa(int(n))
13 }
14
15 type StringableVector[T fmt.Stringer] []T
16
17 func (s StringableVector[T]) String() string {
18     var sb strings.Builder
19     sb.WriteString("<<")
20     for i, v := range s {
21         if i > 0 {
22             sb.WriteString(", ")
23         }
24         sb.WriteString(v.String())
25     }
26     sb.WriteString(">>")
27     return sb.String()
28 }
29
30 func main() {
31     var s StringableVector[num] = []num{1, 2, 3}
32     fmt.Println(s)
```

<<1, 2, 3>>

Program exited.

Go 2 instantiation error

The [go2go](#) Playground [Run](#) [Format](#) [Share](#) Hello, playground ▾

About

```
9 type num int
10
11 func (n num) String() string {
12     return strconv.Itoa(int(n))
13 }
14
15 type StringableVector[T fmt.Stringer] []T
16
17 func (s StringableVector[T]) String() string {
18     var sb strings.Builder
19     sb.WriteString("<<")
20     for i, v := range s {
21         if i > 0 {
22             sb.WriteString(", ")
23         }
24         sb.WriteString(v.String())
25     }
26     sb.WriteString(">>")
27     return sb.String()
28 }
29
30 func main() {
31     var s StringableVector = []num{1, 2, 3}
32     fmt.Println(s)
```

type checking failed for main
prog.go2:31:8: cannot use generic type StringableVector[T fmt.Stringer] without instantiation

Go build failed.

Programming in Go

Matt Holiday
Christmas 2020



Parting Thoughts

Selected Go proverbs

Words to live by 😊

- **Clear is better than clever**
- A little copying is better than a little dependency
- Concurrency is not parallelism
- Channels orchestrate; mutexes serialize
- Don't communicate by sharing memory, share memory by communicating
- Make the zero value useful
- The bigger the interface, the weaker the abstraction
- Errors are values
- Don't just check errors, handle them gracefully

A parting word from Tony Hoare

“There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.”

— Tony Hoare
The Emperor's Old Clothes
(Turing Award Lecture, 1980)

Elegance



Simplicity



Complexity

Some non-Go things

Here is a short list of presentations I think everyone should watch:

- *Simple made easy* (Rich Hickey, QCon 2012)
- *Small is beautiful* (Kevlin Henney, GOTO 2016)
- *Software that fits in your head* (Dan North, GOTO 2016)
- *Worse is better, for better or for worse* (Kevlin Henney, GOTO 2013)
- *Solid snakes or How to take 5 weeks of vacation* (Hynek Schlawack, PyCon 2017)

None of these are about Go, and all of them are about how to build software in Go

(no, I don't necessarily agree with every point made by these or other speakers)

Learning resources (books, blogs, videos)

The screenshot shows a GitHub repository page for `matt4biz/go-resources`. The repository has 2 stars, 2 forks, and 2 commits. The `Code` tab is selected. The `README.md` file contains a section titled **Go resources** which lists books, articles, and other resources about learning Go. It also mentions `Awesome Go` and includes an apple emoji next to some items.

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master 1 branch 0 tags Go to file Add file Code

matt4biz Fix headers in the video section 7934081 on May 18, 2020 22 commits

docs Fix headers in the video section 9 months ago

README.md The great re-organization 9 months ago

About
Resources for Go (golang) developers
[Readme](#)

Releases
No releases published
[Create a new release](#)

Packages
No packages published
[Publish your first package](#)

Contributors 2

matt4biz Matt Holiday
mholiday-nyt Matthew Holiday

README.md

Go resources

This is a list of books, articles, videos, and other resources about learning Go along with some related topics.

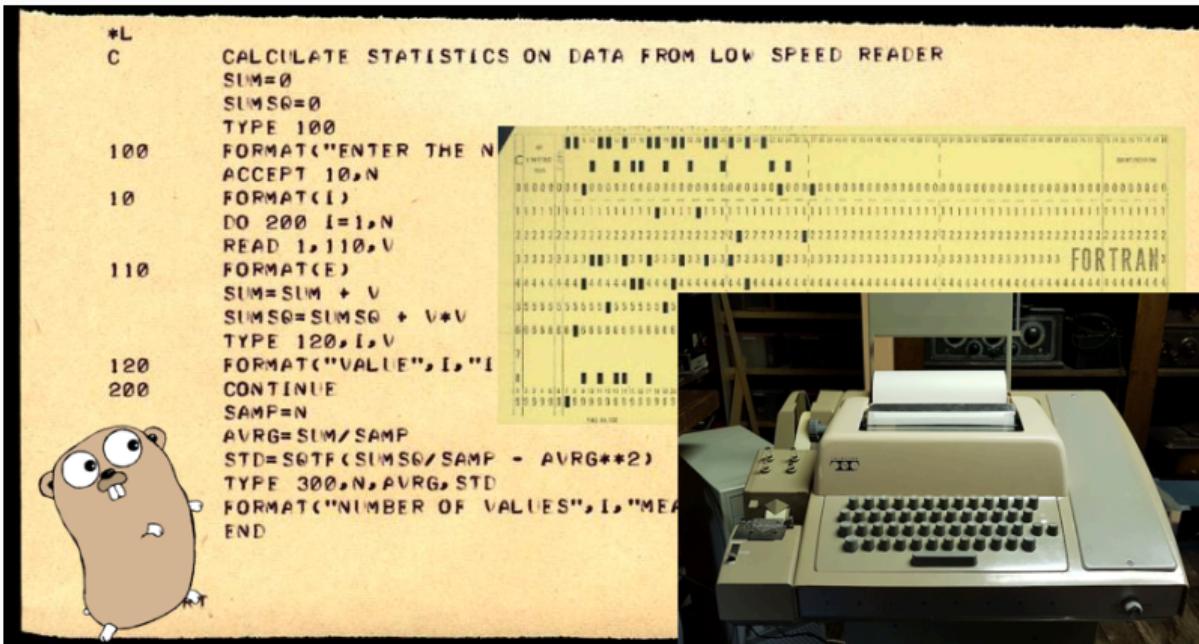
Although a few tools or frameworks are mentioned, this site is particularly oriented towards tutorials or articles to improve one's knowledge of Go or software development, particularly development for the cloud.

If you're looking for links to software, [Awesome Go](#) is a site primarily oriented toward Go projects with some information on conventions and tutorials also.

The apple 🍎 marks items of particular note.

Goals

“Never stop learning”



One more thing . . .

I will follow this class with another on building web apps in Go

- basic REST APIs
- databases
- GraphQL
- testing