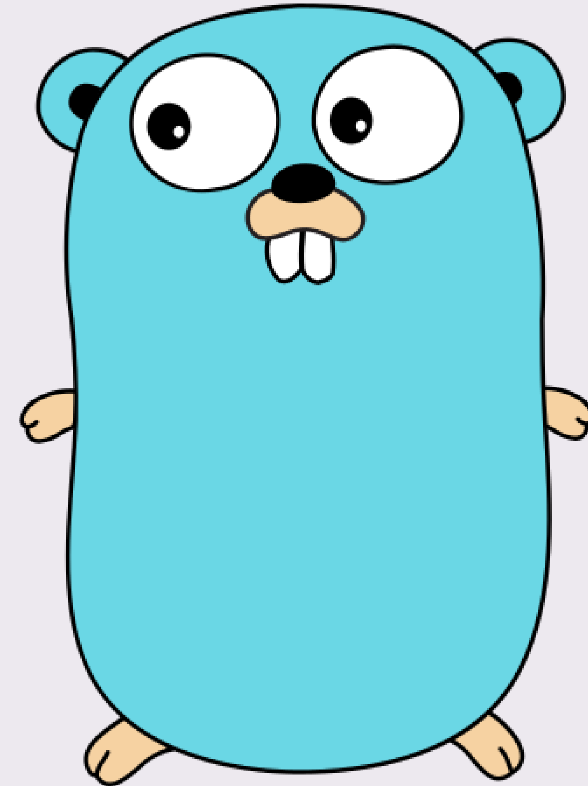
Welcome!



Why your next project should be written in Go

Confessions of a recovering C developer

Zygmunt Krynicki

- zygmunt.krynicki@huawei.com | me@zygoon.pl

About me

Zygmunt grew up with Atari and 386 in the 80s and 90s in central european capital city. He's a self-taught programmer proficient in C, Python and Go. Most of his carrier he's been involved in open and free source software, writing services, utilities and tools.

Principal Technologist at Huawei Open Source Technology Center, Ex-Canonical

In spare time enjoys biking, retro programming and collecting books to read one day.

Agenda

- World
- Tooling
- Language
- Interfaces
- Highlights
- Q&A

World

Premise

Embedded world has changed

- Move from micro-controllers to small computers
- Increasing complexity of applications
- Security & maintenance requirements

Tooling

Go tools are a joy to use

Go is easy to build

`go build` is all you need

- Forget bitbake, dev-tool and Yocto
- Forget configure, autotools, meson, cmake, ...
- Forget pkg-config, foo-config scripts
- Forget dependency hell

Plain `make` is common for larger projects though.

Go builds are fast

- Clever language design
- Clever compiler optimizations
- Iterations counted in seconds (builds&test, build&run)

Go is easy to share

```
go install example.org/stuff/pkg/potato@latest
```

- Use locally created or 3rd party software
- Highly cross platform (Linux/Windows/MacOS)
- This builds a static binary, no compiler dependency

Go is easy to deploy

- Static linking
  - no runtime / interpreter / VM as dependency
  - usually just one file
- Not using libc unless Cgo is needed
  - Forget glibc vs musl vs uclibc woes
- Predictable runtime behavior
  - Great for containers and servers alike
- Binaries may embed extra file-systems
  - Ship web app *and* assets together

Go has great cross-compile story

```
GOOS=linux GOARCH=arm64 go build ./...
```

- Compile to and from Linux, Windows, MacOS, ..., x86, arm, risc, ...
  - Also for: Android, iOS, wasm
- Cgo needs external C cross compiler
- Yes it's this simple, two env variables.

`scp` and run :)

Go has a great dependency story

Package is the smallest unit for compile-time include syntax

Module is the smallest unit for dependency management

- Each module is a set of packages
- Module paths are unique (URLs)
- No central infrastructure for registering module names required
- https://pkg.go.dev/ indexes public modules wherever they are

Go has more tooling goodies

- Standard way to test, benchmark and fuzz (since 1.18)
- Standard way to re-format all code
- Official language server (great IDE integration)

Go Language

Opinionated selection of interesting properties

Not nearly enough time to list all nice things

Go is easy to read and write

- The language is clear and simple
- You can pick it up and be productive in days
- Well suited for engineering tasks
  - not a research playground
  - stability, practicality, maturity

Stability

https://go.dev/doc/go1compat

> It is intended that programs written to the Go 1 specification will continue to compile and run correctly, unchanged, over the lifetime of that specification. At some indefinite point, a Go 2 specification may arise, but until that time, Go programs that work today should continue to work even as future "point" releases of Go 1 arise (Go 1.1, Go 1.2, etc.).
> Compatibility is at the source level. Binary compatibility for compiled packages is not guaranteed between releases. After a point release, Go source will need to be recompiled to link against the new release.
> The APIs may grow, acquiring new packages and features, but not in a way that breaks existing Go 1 code.

Old and familiar

- Looks and feels a lot like C
- Statically typed, compiled, imperative
- Values, structures, functions, arrays
- No function overloading
- No complex inheritance
- No exceptions and unwind stacks

C interop: Cgo

Using C and Go together

```
package main

// #include <stdlib.h>
// #include <stdio.h>
// /* any C code you want here */
import "C"
import "unsafe"

func main() {
	s := C.CString("C stands for CVE")
	C.puts(s)
	C.free(unsafe.Pointer(s))
}
```

- Same `go build` as before
- `C` is a special magic package with included C symbols
- Rules for interaction, somewhat involved but usable

Cgo consequences

- Uses platform compiler
  - Links to libc
  - Less version independent
- Easier to crash
  - It's C after all
- Useful to have when needed

New and improved

- **Memory safe**, GC
- Slices and maps
- No preprocessor
- Interface types
- No implicit conversions
- Public/private symbols
- Multiple return values
- Error values, defer, panic, recover
- Reflection, Structure tags
- Concurrency (channels, goroutines!)

Interfaces

Everything about interfaces in awesome

Interface types

- Interface is a *set of methods* (signatures)
- Any type with matching set can be used though this interface
- Interface values are small, fixed size and passed by value
- Type + Value pairs inside, value usually a pointer
  - Compile-time optimization of interesting cases

    Cost of call identical to C++ virtual method calls

Interface syntax

```go
type Greeter interface {
    Hello() string
    // Other methods here
}
```

- Any type implementing a `Hello() string` method is a `Greeter`
- No equivalent for Java's `implements`
- Interface names don't matter
  - `interface Hellower { Hello() string }` - same thing

Interfaces decouple interchangeable things

This is `io.Writer`, one of standard interfaces.

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

This is `fmt.Fprintf`, the `fprintf` equivalent.

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Print to anything with `Write([]byte) (int, error)` using `fmt.Fprintf`.

The empty interface

```
interface{}
```

- Every type implements this interface
  - The `void *` of Go
  - Fully type-safe
- Type assertions can recover concrete type
- Reflection provides access to all details

Highlights

Great features don't have to be large

Reflection

```
func foo(thing {}interface) {
  val := reflect.ValueOf(thing)
  val.Kind() // Bool, Int, Int8, Int16, ...
             // Array, Slice, Map, Chan, Struct, ...
             // Func, Ptr, Interface, ...
  // Lots of methods

  typ := reflect.TypeOf(thing)
  // Lots of methods
}
```

- Look at any value of any time
- Explore and manipulate
  - read, write, convert, call
- Access type meta-data

Struct tags

```
type Person struct {
        Nick string `json:"nick"`
        Away bool   `json:"away,omitempty"`
}
```

- Attach strings to structure fields
- Retrieve specific tag through reflection
- Extensible

```
data, err := json.Marshal(Person{Name: "zyga", Away: false})
```

- `data` is `{"name": "zyga"}` as `[]byte`

Zero values

- Each type has a corresponding zero value
  - `0`, `0.0`, `false`, `""`, `nil` and so on
- No uninitialized memory anywhere

Symbol visibility

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

func Hello() {
  greet()
}
```

- `Capitalized` symbols are exported
  - can be used from other packages
- `other` symbols are private to the package
- Draws attention to public API surface

Deferred calls

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

// Use f ...
```

- Resource cleanup logic right next to acquisition
- Easy to do the right thing
- Defer uses stack

Embedding

```
import "embed"

//go:embed image/* template/*
//go:embed html/index.html
var content embed.FS
```

- `//go:embed` embeds files into application binary
- variable types: `[]byte`, `string`, `embed.FS`
- `embed.FS` provides `Open`, `ReadDir`, `ReadFile`

Consider Go for your project

Go is engineered to be scalable, practical, easy and safe

Go has excellent standard library

Go has pretty good 3rd party ecosystem

Go has great tooling

Get started with Go

Unified web presence at https://go.dev/

Thank you!

Feedback

- irc: zyga (libera) | twitter: @zygoon
- https://gitlab.com/zygoon | https://github.com/zyga

Q&A

If time permits