Programming in Go
Lesson 6: More Stuff

Matt Holiday

21 May 2019

Cardinal Peak

Lesson #6

What we'll cover today:

- Homework #5
- Enumerated types
- Variable argument lists
- Panic & recover
- Reflection
- Custom JSON decoding via reflection
- Building for distribution
- Containerizing Go
- Discussion

Homework #5

```
## who cares about the errors
## what matters is creating race conditions

$ go test -race .
got item=shoes&price=46 = 400 (<nil>)
got item=sandals&price=27 = 404 (<nil>)
got item=shoes = 400 (<nil>)
got item=socks&price=6 = 400 (<nil>)
got item=socks = 400 (<nil>)
got item=clogs&price=36 = 404 (<nil>)
got item=sandals = 400 (<nil>)
==================
WARNING: DATA RACE
Write at 0x00c000096d50 by goroutine 20:
got item=pants&price=30 = 404 (<nil>)
  runtime.mapassign_faststr()
      /usr/local/Cellar/go/1.11.5/libexec/src/runtime/map_faststr.go:190 +0x0
  hw5.database.add()
      /Users/mholiday/go/src/hw5/main.go:41 +0x295
  hw5.database.add-fm()
      /Users/mholiday/go/src/hw5/main.go:101 +0x65
  net/http.HandlerFunc.ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:1964 +0x51
  net/http.(*ServeMux).ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2361 +0x191
  net/http.serverHandler.ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2741 +0xc4
  net/http.(*conn).serve()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:1847 +0x80a
```

Homework #5

```
Previous read at 0x00c000096d50 by goroutine 91:
  runtime.mapaccess2_faststr()
      /usr/local/Cellar/go/1.11.5/libexec/src/runtime/map_faststr.go:101 +0x0
  hw5.database.update()
      /Users/mholiday/go/src/hw5/main.go:51 +0x107
  hw5.database.update-fm()
      /Users/mholiday/go/src/hw5/main.go:102 +0x65
  net/http.HandlerFunc.ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:1964 +0x51
  net/http.(*ServeMux).ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2361 +0x191
  net/http.serverHandler.ServeHTTP()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2741 +0xc4
  net/http.(*conn).serve()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:1847 +0x80a

Goroutine 20 (running) created at:
  net/http.(*Server).Serve()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2851 +0x4c5
  net/http.(*Server).ListenAndServe()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:2764 +0xe8
  net/http.ListenAndServe()
      /usr/local/Cellar/go/1.11.5/libexec/src/net/http/server.go:3004 +0xef
  hw5.runServer()
      /Users/mholiday/go/src/hw5/main.go:106 +0x376

. . .

==================
```

Homework #5

Test program:

```go
package main // main_test.go

import (
    "fmt"
    "net/http"
    "os"
    "testing"
    "time"
)

type sku struct {
    item  string
    price string
}
```

Homework #5

```
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:8000/" +
                          cmd + "?" + parms)
    if err == nil {
        defer resp.Body.Close()
    }
    fmt.Fprintf(os.Stderr, "got %s = %d (%v)\n", parms,
                resp.StatusCode, err)
}
```

Homework #5

```
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)
        }
    }
}
```

Homework #5

```
func runDrops() {
    for {
        for _, s := range items {
            doQuery("delete", "item="+s.item)
        }
    }
}

func TestServer(t *testing.T) {
    go runServer()

    go runAdds()
    go runDrops()
    go runUpdates()

    time.Sleep(30 * time.Second)
    t.Errorf("NO RACE?")
}
```

Homework #5

New main program:

```
package main // main.go

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

// NOTE: don't do this in real life
type dollars float32

func (d dollars) String() string {
    return fmt.Sprintf("$%.2f", d)
}
```

Homework #5

```
// We embed a sync.Mutex into the database but now it
// must be a struct and be passed by reference (ptr);
// it's not safe to copy a mutex

type database struct {
    sync.Mutex
    data map[string]dollars
}

func (db *database) list(w http.ResponseWriter,
                         req *http.Request) {
    db.Lock()
    defer db.Unlock()

    for item, price := range db.data {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

Homework #5

```
func (db *database) add(w http.ResponseWriter,
                        req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")

    db.Lock()
    defer db.Unlock()

    if _, ok := db.data[item]; ok {
        w.WriteHeader(http.StatusBadRequest) // 404

        fmt.Fprintf(w, "duplicate item: %q\n", item)
        return
    }
```

Homework #5

```
        if f64, err := strconv.ParseFloat(price, 32); err != nil {
            w.WriteHeader(http.StatusBadRequest) // 400

            fmt.Fprintf(w, "invalid price: %q\n", price)
        } else {
            db.data[item] = dollars(f64)

            fmt.Fprintf(w, "added %s with price %s\n", item,
                        dollars(f64))
        }
    }
```

Homework #5

```
func (db *database) update(w http.ResponseWriter,
                           req *http.Request) {
    item := req.URL.Query().Get("item")
    price := req.URL.Query().Get("price")

    db.Lock()
    defer db.Unlock()

    if _, ok := db.data[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
```

Homework #5

```
        if f64, err := strconv.ParseFloat(price, 32); err != nil {
            w.WriteHeader(http.StatusBadRequest) // 400

            fmt.Fprintf(w, "invalid price: %q\n", price)
        } else {
            db.data[item] = dollars(f64)

            fmt.Fprintf(w, "new price %s for %s\n", dollars(f64),
                        item)
        }
    }
```

Homework #5

```
func (db *database) fetch(w http.ResponseWriter,
                          req *http.Request) {
    item := req.URL.Query().Get("item")

    db.Lock()
    defer db.Unlock()

    if _, ok := db.data[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }

    fmt.Fprintf(w, "item %s has price %s\n", item,
                db.data[item])
}
```

Homework #5

```
func (db *database) drop(w http.ResponseWriter,
                         req *http.Request) {
    item := req.URL.Query().Get("item")

    db.Lock()
    defer db.Unlock()

    if _, ok := db.data[item]; !ok {
        w.WriteHeader(http.StatusNotFound) // 404

        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }

    delete(db.data, item)
    fmt.Fprintf(w, "dropped %s\n", item)
}
```

Homework #5

```
var db = database{data: map[string]dollars{"shoes": 50,
                                           "socks": 5}}

func runServer() {
    http.HandleFunc("/list", db.list)
    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))
}

func main() {
    runServer()
}
```

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` group and increments once on each successive 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)
    MiB
    GiB
    TiB
    PiB
    EiB
)
```

So `EiB` is set to $2^{60} = 1152921504606846976 \approx 10^{19}$

Iteration

The iter package doesn't allocate because `struct{}` requires no space (originally released as an "educational joke")

```
package main

import (
    "fmt"
    "github.com/bradfitz/iter"
)

func main() {
    for i := range iter.N(5) {
        fmt.Println(i)
    }
}
```

The playground now has 3rd-party package and multi-file support!

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 {
    total := 0

    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

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**

The two ways of handling errors

There are two main categories of errors:

- errors resulting from user input or environmental conditions;
  for example, a “file not found” error
- errors resulting from invalid program logic;
  for example, a nil pointer

Go handles the first case by returning the `error` type

For program logic errors, Go does a `panic`

Generally this will cause the program to crash with a traceback

When your program has a logic bug

Fail hard, fail fast

When should we panic?

Only when the error was caused by our own programming defect

- if we encoded something, then it should decode again
- if we built a data structure, we should be able to walk it
- we should recognize any message we send to ourselves

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!**

```
if node != root && !leaf(node) && len(node.children) < 2 {
    panic("internal node has too few children")
}
```

Exception handling

Exception handling was popularized to allow “graceful degradation” of safety-critical systems (e.g., Ada and flight control software)

Exception handling introduces many additional paths of execution that are effectively invisible in the code

Code with exceptions is harder to analyze

Ironically, most safety-critical systems are built without using exceptions!

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 can you do?
            fmt.Println("recover:", p)
        }
    }()

    abc()
}

// prints recover: omg
```

Define errors out of existence

Error cases are one of the primary sources of complexity

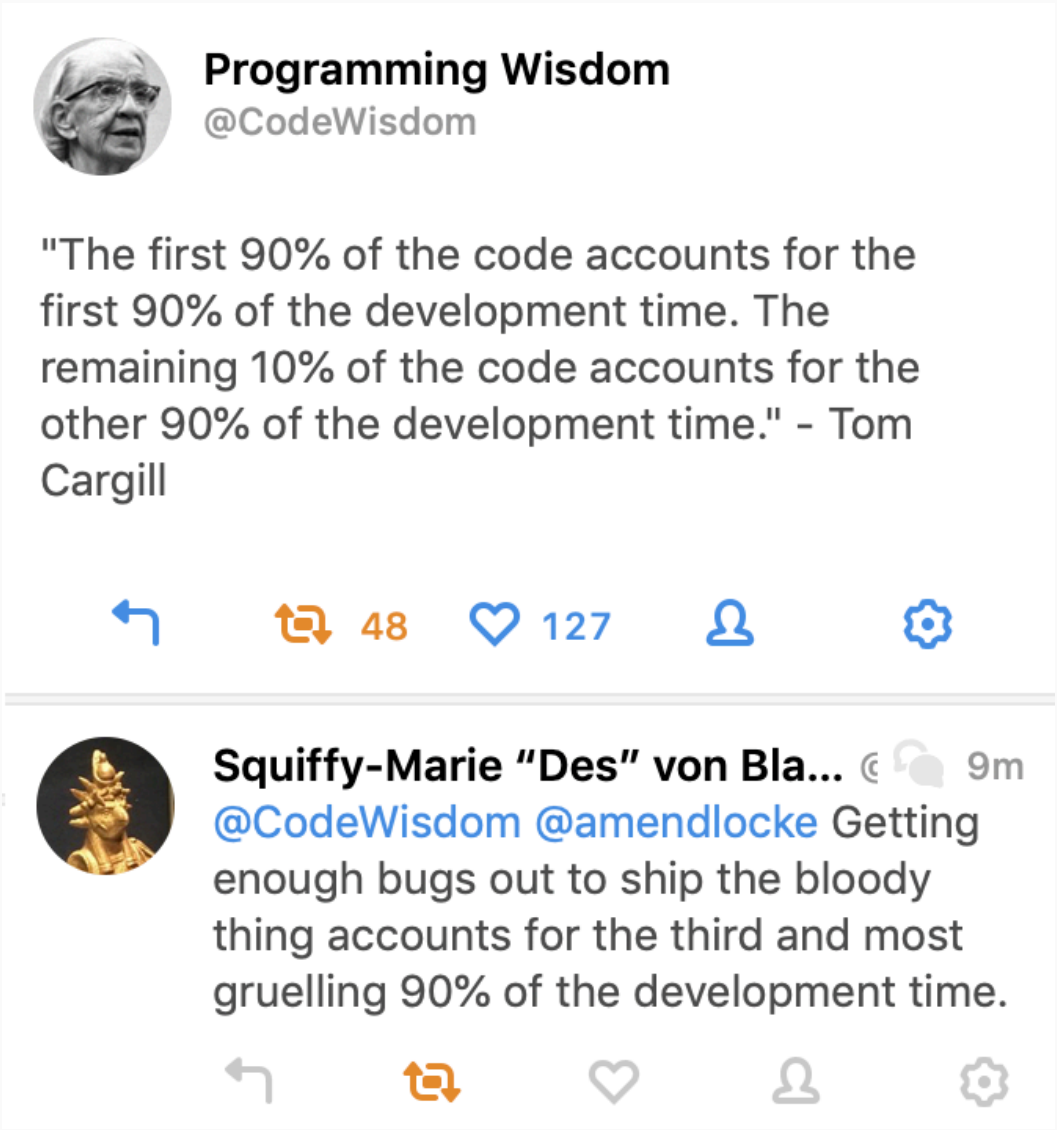
The best way to deal with many errors is to make them impossible

Design abstractions so that all inputs are meaningful:

- deleting a non-existant item from a map
- taking the length of a nil slice (returns 0)
- reading from a nil map (returns a default value)

All these things reduce “special-case” logic that’s hard to test and debug (or even think about!)

Getting the bugs out



Using Reflection

Switching on type

`interface{}` can be anything, since it has no methods

Which means it's a "generic" thing; we need its concrete type

```
func Println(args ...interface{}) {
    buf := make([]byte, 0, 80)

    for arg := range args {
        switch a := arg.(type) {
            case string:
                buf = append(buf, a...)
            case fmt.Stringer:
                buf = append(buf, a.String()...)
            . . .
        }
    }
}
```

Type extraction

We can also extract a concrete type with a conversion

If we use the two-result version, we can avoid `panic`

```
data map[string]interface{}

// this code panics if the conversion can't be made

score := data["score"].(float64)

// this code handles failure gracefully ...

if intent, ok := data["intent"].(string); ok {
    w.Intent = intent
} else {
    return fmt.Errorf("missing intent")
}
```

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"}
}
```

You can't describe this easily using a fixed `struct` definition

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 the `respWrapper` because it must have a separate unmarshal function from the `response` type (see below)

Custom JSON decoding

```
func (r *respWrapper) UnmarshalJSON(b []byte) (err error) {
    var raw map[string]interface{}

    // ignore error handling
    err = json.Unmarshal(b, &r.response)
    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 contains(unknown, known map[string]interface{}) error {
    for k, v := range known {
        switch x := v.(type) {
        case string:
            if !matchString(k, x, unknown) {
                return fmt.Errorf("%s unmatched (%s)", k, x)
            }

        . . .
```

Testing JSON

```
        . . .

        case map[string]interface{}:
            if v, ok := unknown[k]; !ok {
                return fmt.Errorf("%s missing", k)
            } else if u, ok := v.(map[string]interface{}); ok {
                if err := contains(u, x); err != nil {
                    return fmt.Errorf("%s != %+v: %s",
                                      k, x, err)
                }
            } else {
                return fmt.Errorf("%s not obj (%#v)", k, v)
            }
        }
    }

    return nil
}
```

Testing JSON

```
func matchString(key, exp string,
                 resp map[string]interface{}) bool {

    // if the key is present, extract the "any" value
    // and convert it to a string and test for equality

    if v, ok := resp[key]; ok {
        if val, ok := v.(string); ok && val == exp) {
            return true
        }
    }

    return false
}
```

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`

We can build "pure" Go programs (with some cautions):

```
$ CGO_ENABLED=0 go build -a -tags netgo \
                -ldflags '-w' \
                -o <name> <src path>
```

Here we must tell Go we're going to use pure Go networking, which might be a little slower

Go build platforms

Go can cross-compile, too

- $GOARCH defines the architecture (e.g., amd64)
- $GOOS defines the operating system (e.g., darwin)
- For ARM, we have $GOARM to choose the chip version

We can build for the Raspberry Pi

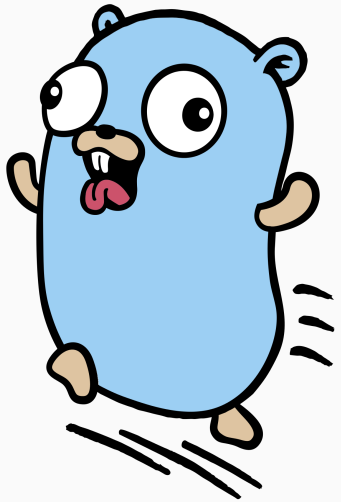
```
$ GOOS=linux GOARCH=arm GOARM=7 CGO_ENABLED=0 \
                go build -a -tags netgo \
                -ldflags '-w' \
                -o mainPi ./main.go

$ file mainPi
mainPi: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
        statically linked, not stripped
```

Gokrazy!

Gokrazy builds a container that boots on the Raspberry Pi (3B, 3B+)

- It's pure Go
- It requires no operating system
- All it does is run your program



You use it to build a custom memory chip to boot your Pi

You get a simple Web API to start/stop your program

Vendoring code

We don't need `$GOPATH` as of Go 1.11

And we may want to control our 3rd-party dependencies

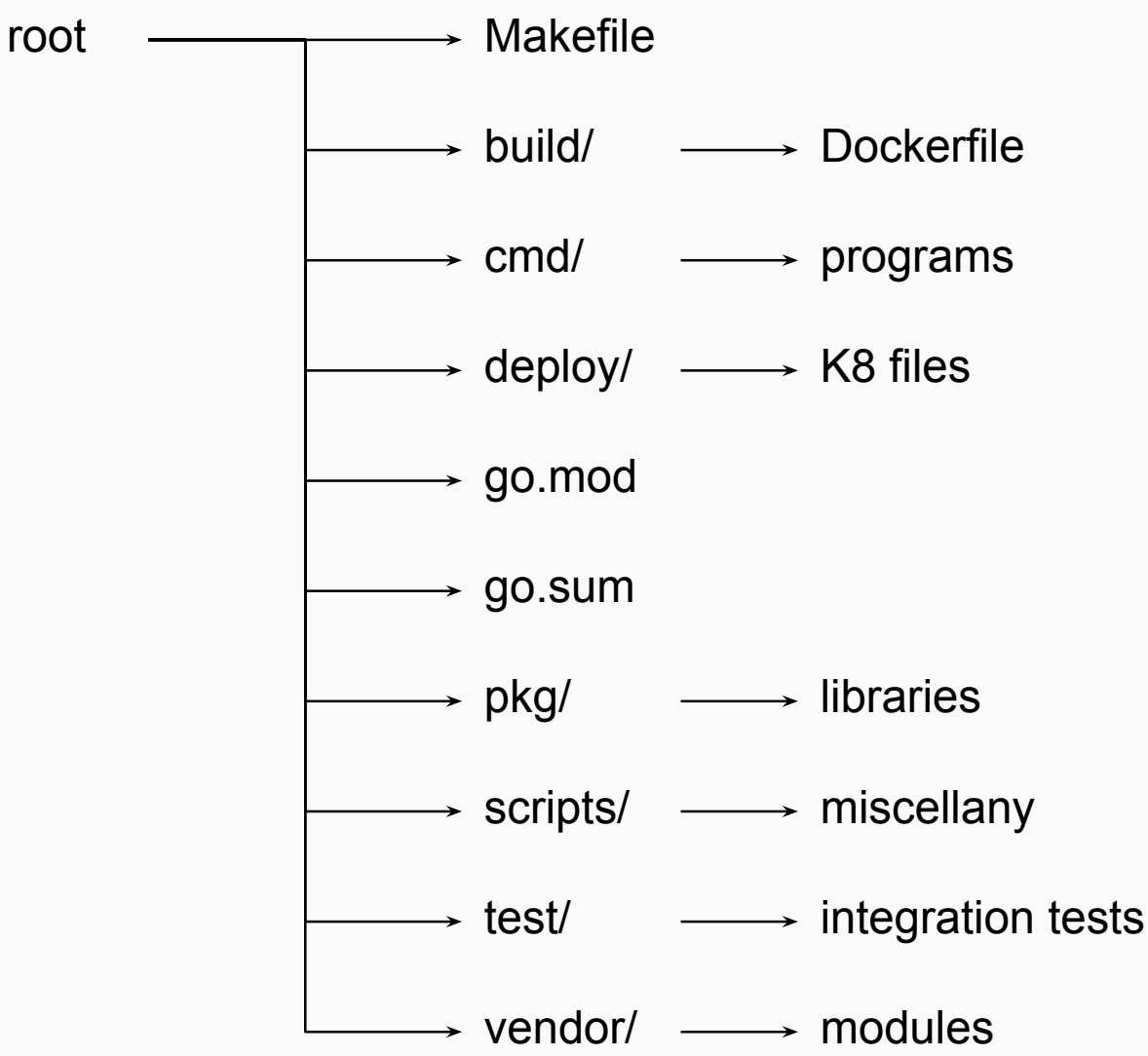
```
# outside $GOPATH

$ go mod init
$ go build mod=vendor ...
```

You end up with two files: `go.mod` and `go.sum` and a `vendor` directory — commit them all

See Converting to modules

Directory structure



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

// do not remove or modify
var version string
```

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)

bvi: $(SOURCES)
    go build -mod=vendor -ldflags "-X main.version=$(version)" \
        -o $@ ./cmd/bvi
```

Makefile extracts

```
SOURCES := $(wildcard */*.go */*/*.go)

version=$(shell git describe --tags --long --dirty)

bvi: $(SOURCES)
go build -mod=vendor -ldflags "-X main.version=$(version)" -o $@ ./cmd/bvi

bvi-linux: $(SOURCES)
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -mod=vendor -a -tags netgo \
    -ldflags "-w" -ldflags "-X main.version=$(version)" -o $@ ./cmd/bvi

.PHONY: committed
committed:
@git diff --exit-code >/dev/null || (echo "** NOT COMMITED **"; exit 1)

.PHONY: docker
docker: build/Dockerfile $(SOURCES)
sed -e "/FIXME/s/FIXME/${version}/" -i.bak build/Dockerfile
docker build -t sleigh-bvi:latest . -f build/Dockerfile
mv build/Dockerfile.bak build/Dockerfile
```

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 a from-scratch 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

```
FROM golang:1.12.5-alpine3.9 AS builder
RUN /sbin/apk update && /sbin/apk --no-cache add ca-certificates git \
    tzdata && /usr/sbin/update-ca-certificates
RUN adduser -D -g '' bvi
WORKDIR /home/bvi
COPY go.mod /home/bvi
COPY go.sum /home/bvi
COPY vendor /home/bvi/vendor
COPY cmd    /home/bvi/cmd
COPY pkg    /home/bvi/pkg
RUN CGO_ENABLED=0 go build -mod=vendor -a -tags netgo -ldflags "-w" \
    -ldflags "-X main.version=FIXME" -o bvi ./cmd/bvi

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/bvi/bvi /home/bvi
USER bvi
WORKDIR /home
EXPOSE 8444
ENTRYPOINT ["/home/bvi", "2>&1"]
```

Discussion

What questions do you have?

What else would you like me to talk about?

RIP Grumpy Cat 2012–2019