Daniel Martí
Maintainer at cuelang.org

From zero to func main: initializing packages

main packages

```
func main() {
    println("hello")
}
```

```
$ go run main.go
hello
```

main packages

```
var v = "hello"

func main() {
    println(v)
}
```

```
$ go run main.go
hello
```

main packages

```
var v = func() string {
    return "hello"
}()

func main() {
    println(v)
}
```

```
$ go run main.go
hello
```

main packages

```
var v = func() string {
    println("init")
    return "hello"
}()

func main() {
    println(v)
}
```

```
$ go run main.go
init
hello
```

main packages

```
var v string
func init() {
    println("init")
    v = "hello"
}

func main() {
    println(v)
}
```

```
$ go run main.go
init
hello
```

main packages

```
var v1 = v2 + 100
var v2 = f()
var counter = 3

func f() int {
    counter++
    return counter
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
```

main packages

```
var v1 = v2 + 100
var v2 = f()
var counter = 3

func f() int {
    counter++
    return counter
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
```

main packages

```
var v1 = v2 + 100
var v2 = f()
var counter = 3

func f() int {
    counter++
    return counter
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
```

main packages

```
var v1 = v2 + 100
var v2 = f()
var counter = 3

func f() int {
    counter++
    return counter
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
```

main packages

```
var v1 = v2 + 100
var v2 = f()
var counter = 3

func f() int {
    counter++
    return counter
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
104 4
```

main packages

```
var v1 = func() int {
    return v2
}
var v2 = func() int {
    return v1
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
```

main packages

```
var v1 = func() int {
    return v2
}
var v2 = func() int {
    return v1
}

func main() {
    println(v1, v2)
}
```

```
$ go run main.go
main.go:3:5: initialization
cycle for v1
    main.go:3:5: v1 refers to
    main.go:4:5: v2 refers to
    main.go:3:5: v1
```

main packages

```
import "flag"

var v = flag.Int("v", 0, "")

func main() {
    flag.PrintDefaults()
}
```

```
$ go run main.go
 -v int
```

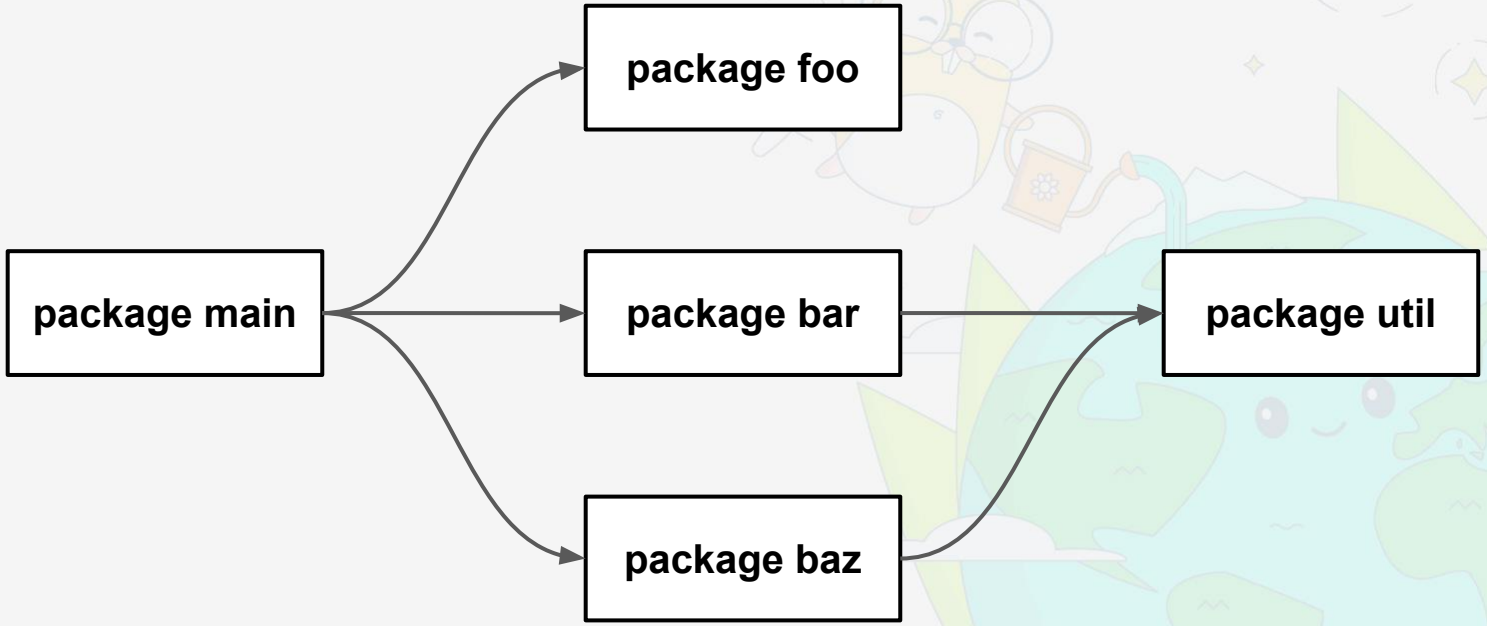
main packages

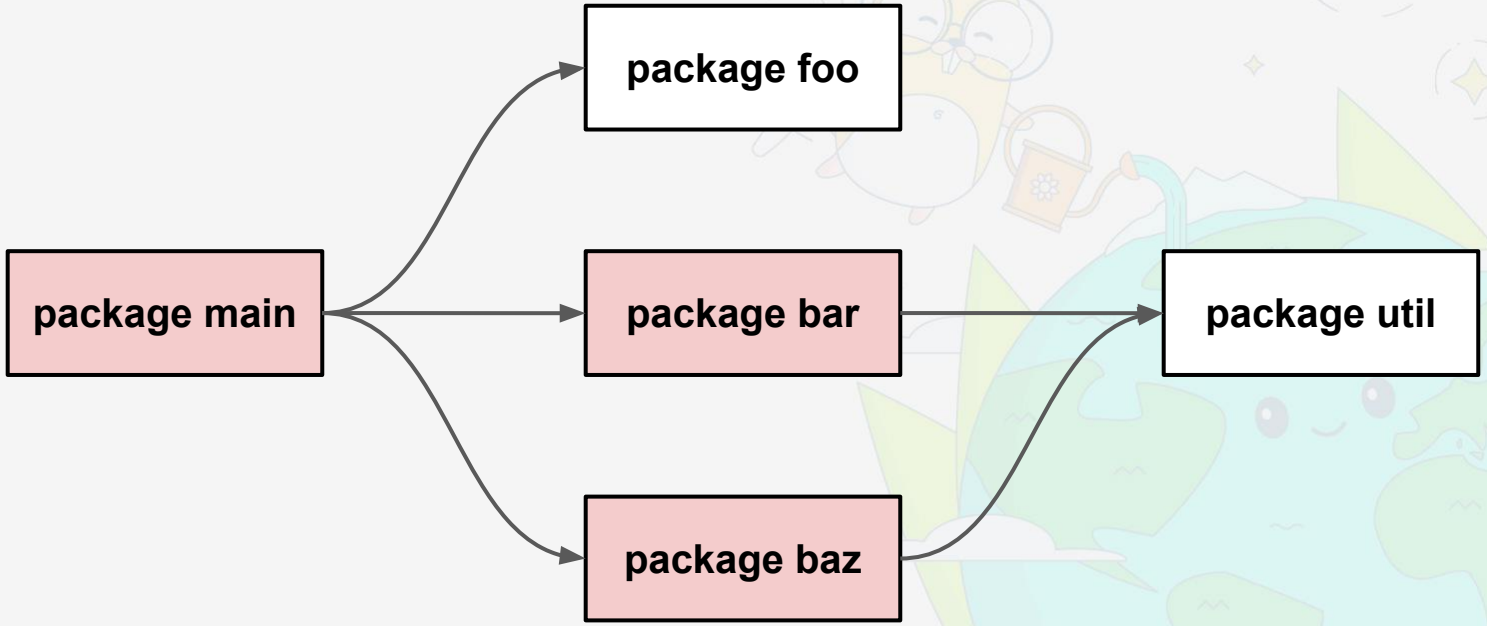
```
package flag

var CommandLine = NewFlagSet(os.Args[0], ExitOnError)

func Int(name string, value int, usage string) *int {
    return CommandLine.Int(name, value, usage)
}
```

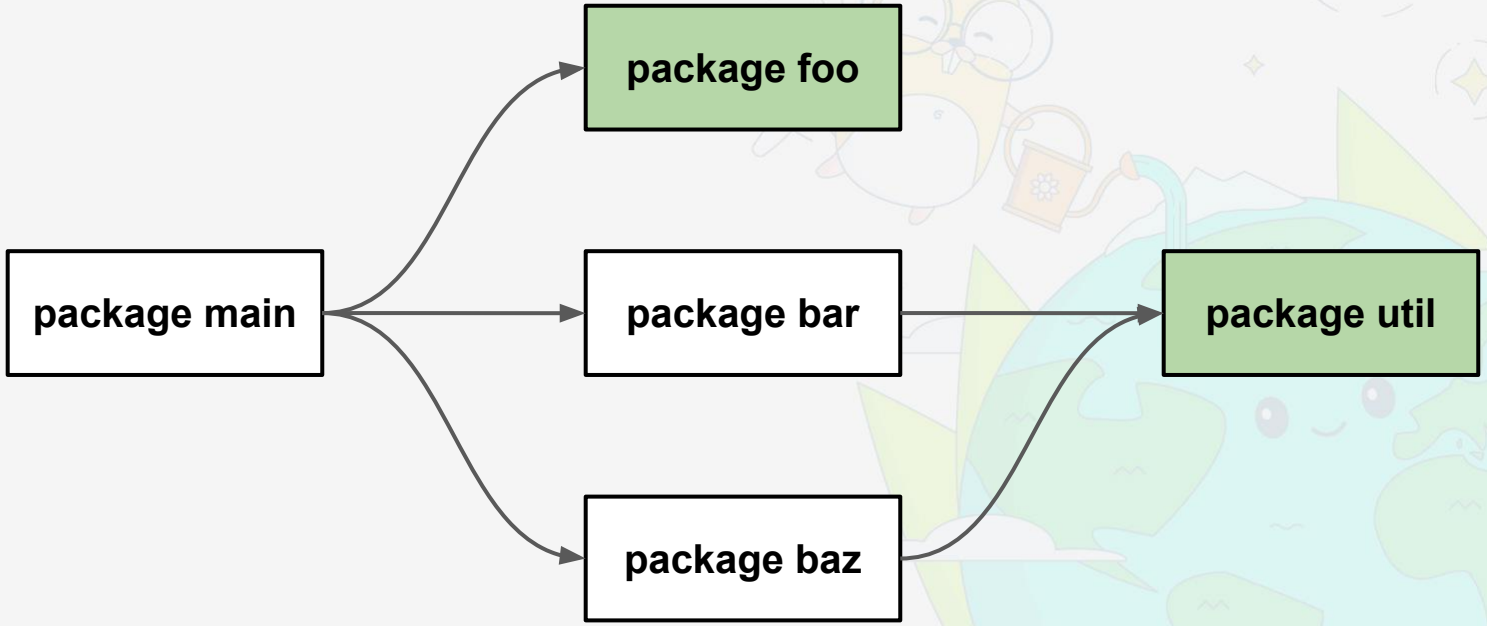
init basics

go.dev/ref/spec

- Imported packages are initialized first
- Vars initialized in declaration order
- Dependency analysis to prevent using variables before initialization
- Sequential; no concurrency


package main
package foo
package bar
package baz
package util

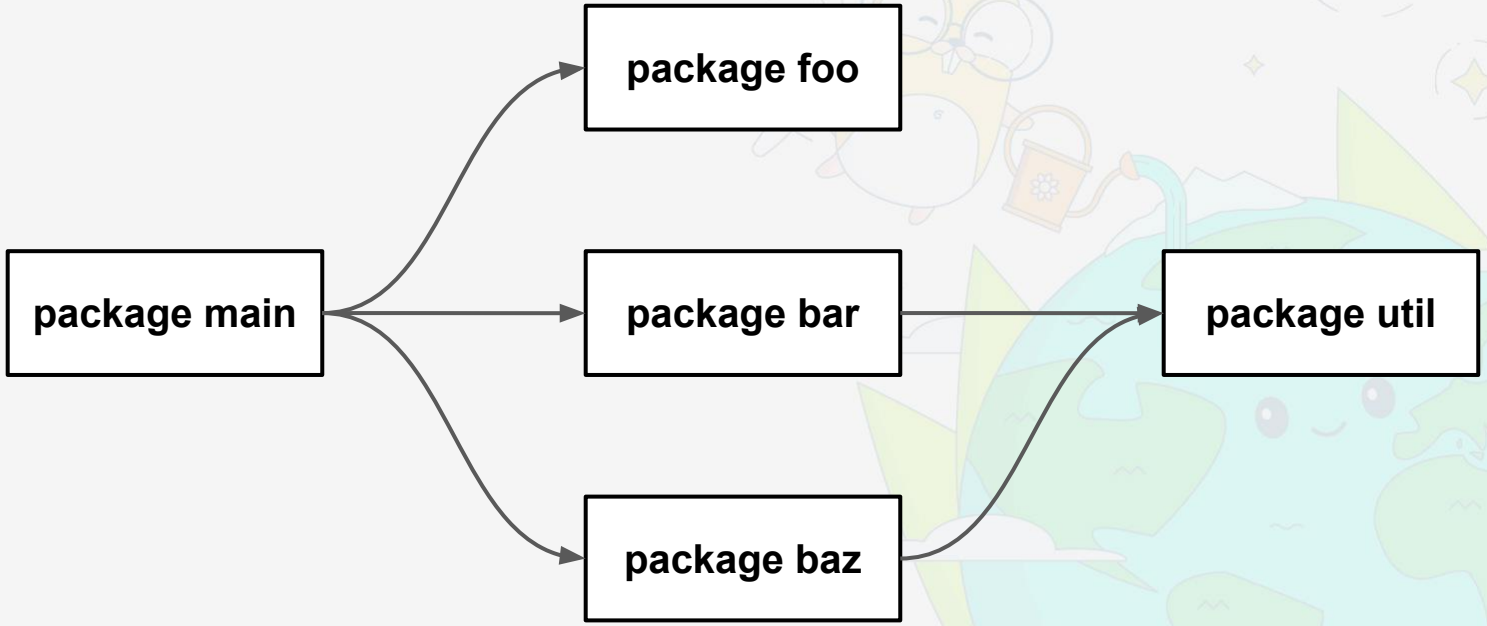

package foo
package main
package bar
package util
package baz

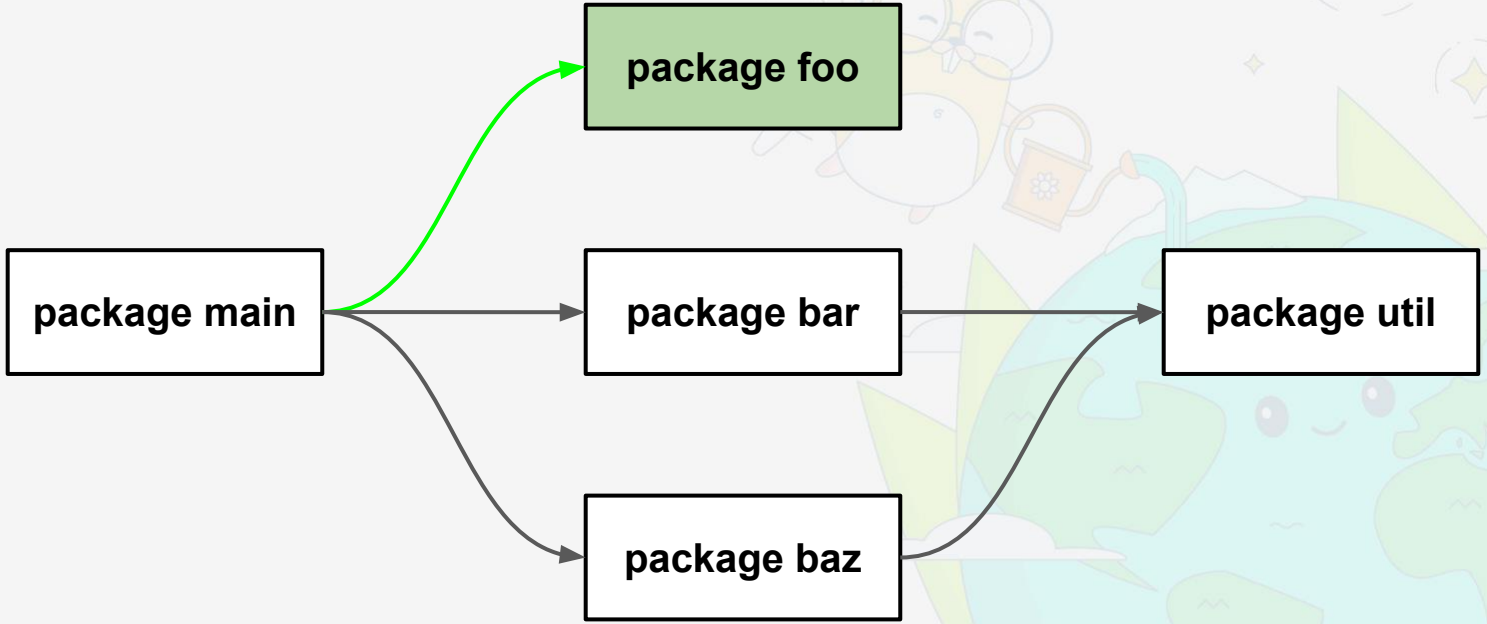

package foo
package main
package bar
package util
package baz

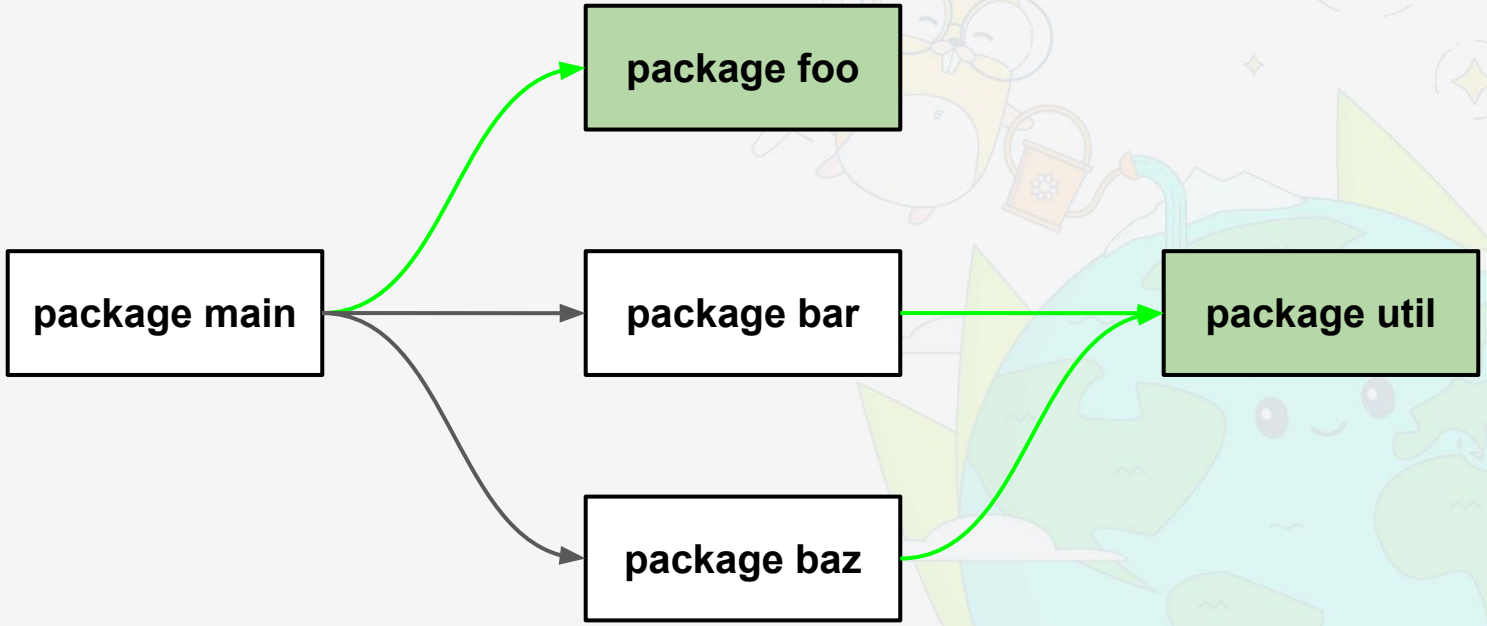
Go 1.21 change

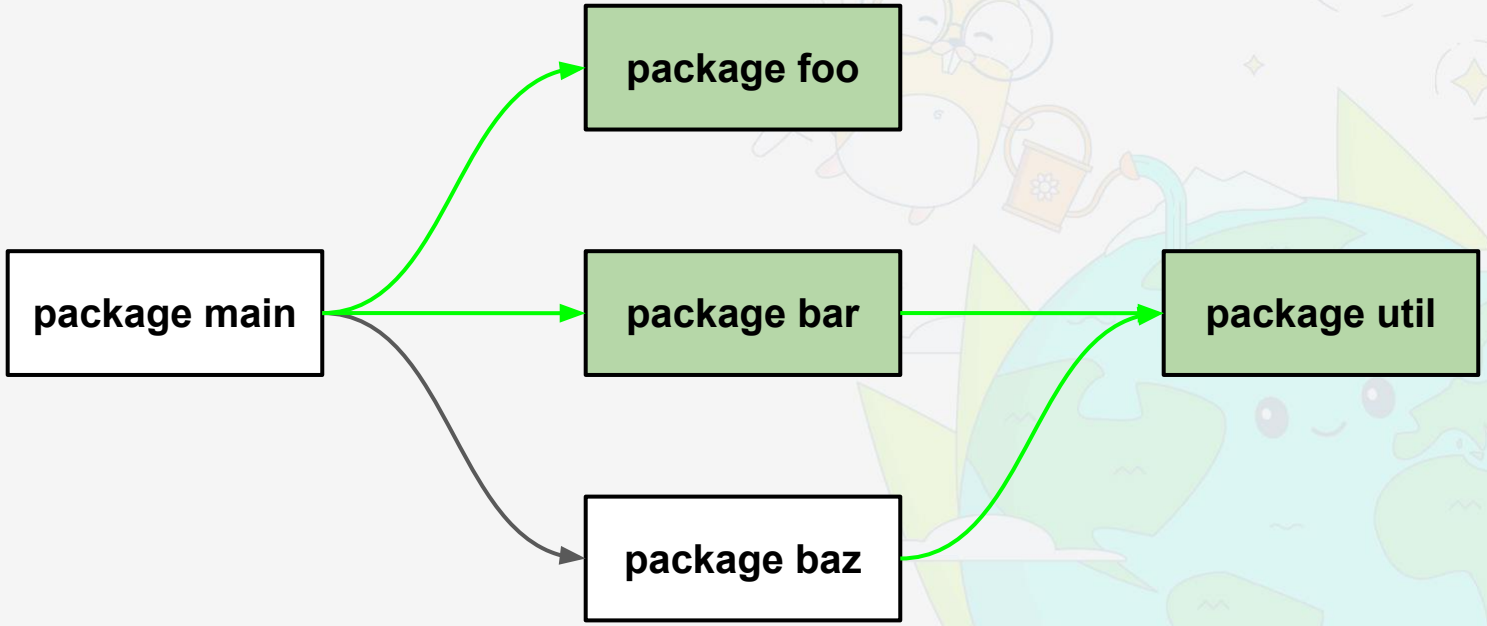
go.dev/doc/go1.21

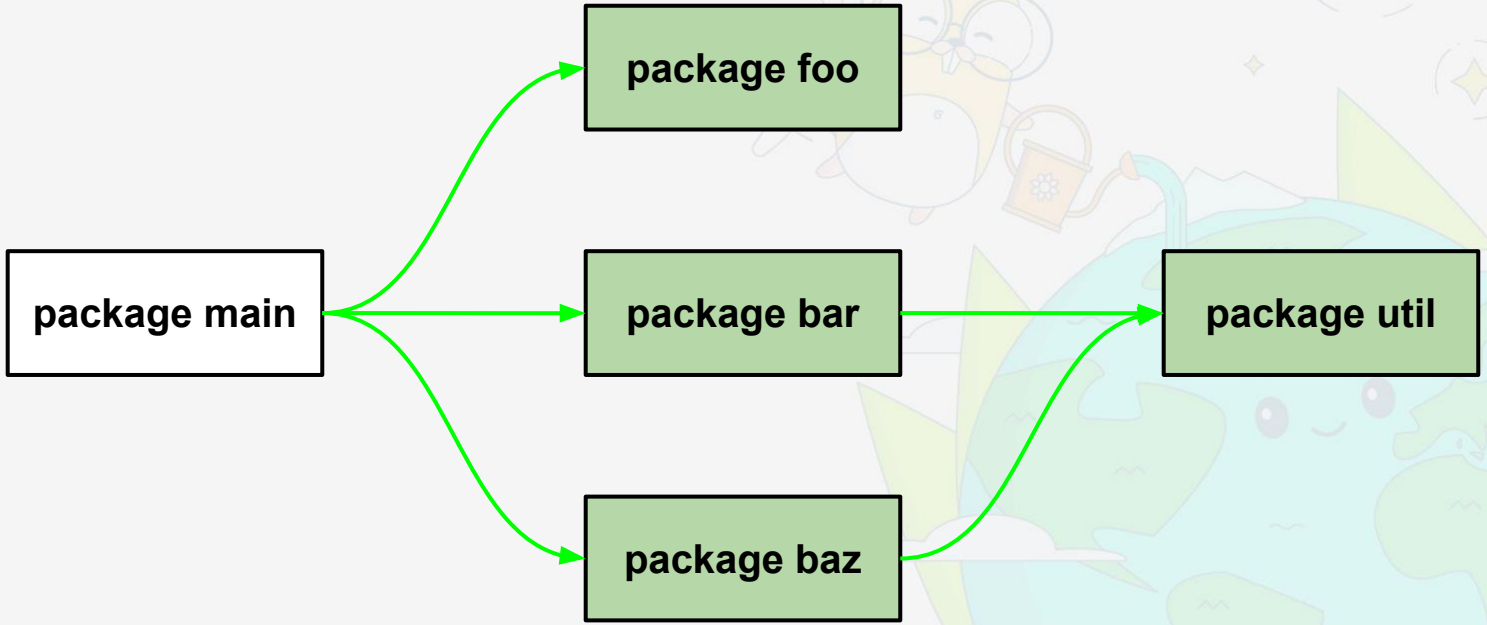
1. Sort all packages by import path
2. Find first not yet initialized whose imports are all initialized
3. Initialize it and return to 2 until done

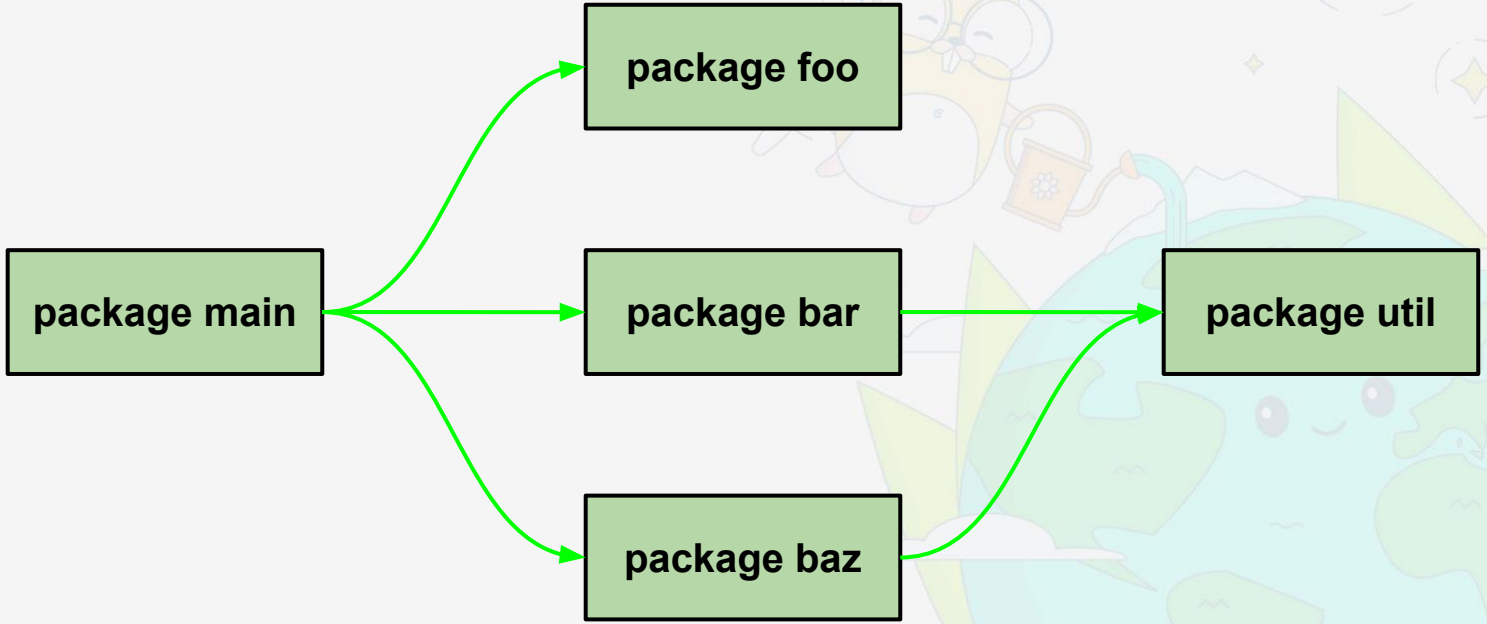

package foo
package main
package bar
package util
package baz


package foo
package main
package bar
package util
package baz


package foo
package main
package bar
package util
package baz


package main
package foo
package bar
package baz
package util


package main
package foo
package bar
package baz
package util

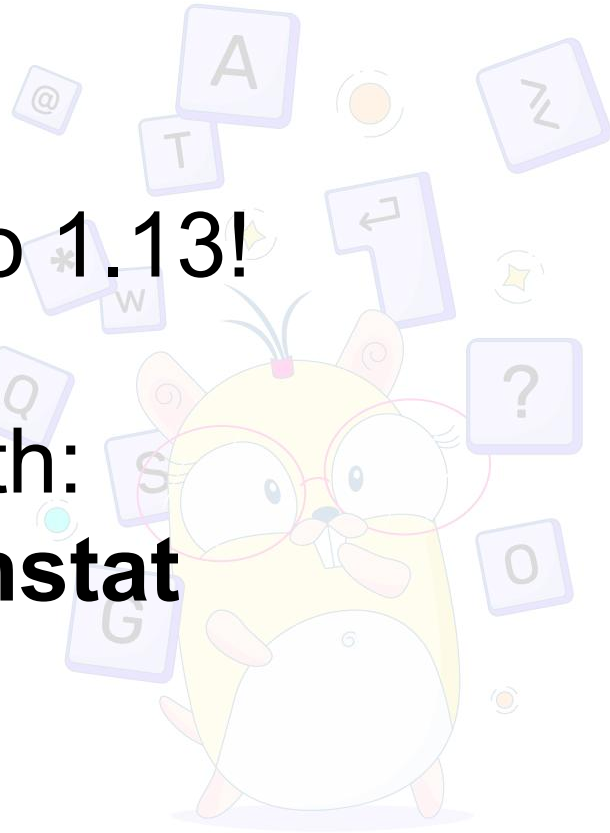

package main
package foo
package bar
package baz
package util

OK but…
Why care about init?

cmd/go too slow for tools

go.dev/issue/29382

- Tools like gopls exec cmd/go
- "go env GOMOD" to locate the current Go module if any
- "go list -json" to find and load Go packages like "staticcheck ./…"
- Every exec cost ~10ms at init, noticeable for interactive tools



This improved with Go 1.13!

Can be shown with:
benchcmd + benchstat

benchcmd + benchstat

```
$ go test -bench=. -count=3
BenchmarkIsNumber-16        304130          4003 ns/op
BenchmarkIsNumber-16        298827          3993 ns/op
BenchmarkIsNumber-16        296257          4062 ns/op
```

benchcmd + benchstat

```
$ go install github.com/aclements/go-misc/benchcmd@master

$ benchcmd -n 3 GoEnv go env GOMOD
BenchmarkGoEnv    1    2302480 ns/op    2347000 user-ns/op
BenchmarkGoEnv    1    2229281 ns/op    2315000 user-ns/op
BenchmarkGoEnv    1    1676279 ns/op    1533000 user-ns/op
```

benchcmd + benchstat

```
$ go install github.com/aclements/go-misc/benchcmd@master

$ benchcmd -n 3 GoEnv go env GOMOD
BenchmarkGoEnv    1    2302480 ns/op    2347000 user-ns/op
BenchmarkGoEnv    1    2229281 ns/op    2315000 user-ns/op
BenchmarkGoEnv    1    1676279 ns/op    1533000 user-ns/op

$ go test -bench=. -count=3 -benchtime=1x
BenchmarkIsNumber-16          1        50776 ns/op
BenchmarkIsNumber-16          1        31169 ns/op
BenchmarkIsNumber-16          1        26210 ns/op
```

Use GOTOOLCHAIN from Go 1.21

go.dev/doc/toolchain

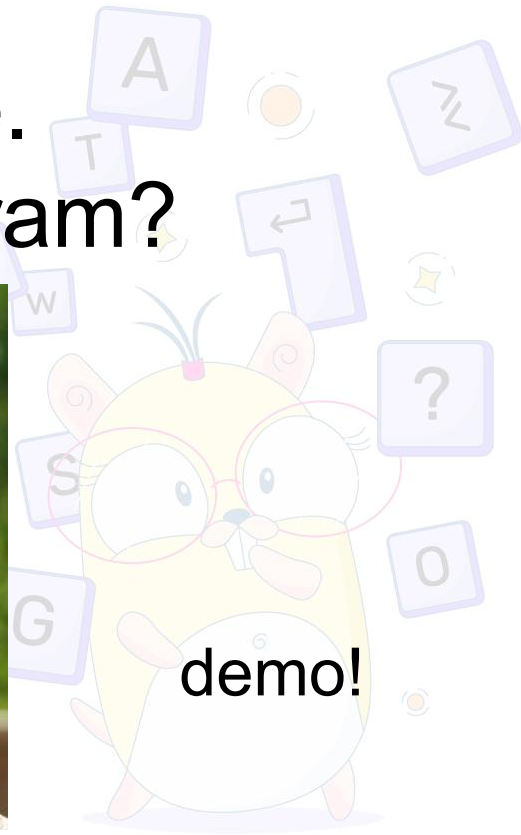
benchcmd + benchstat

```
$ go install github.com/aclements/go-misc/benchcmd@master
$ go install golang.org/x/perf/cmd/benchstat@master

$ GOTOOLCHAIN=go1.12 benchcmd -n 20 GoEnv go env GOMOD >old
$ GOTOOLCHAIN=go1.13 benchcmd -n 20 GoEnv go env GOMOD >new
$ benchstat old new
      |     old      |                  new                  |
      |   sec/op     |   sec/op     vs base                  |
GoEnv   6.668m ± 4%   4.867m ± 7%  -27.01% (p=0.000 n=20)
```

GODEBUG=inittrace

```
$ GODEBUG=inittrace=1 go env GOMOD
init internal/bytealg @0 ms, 0 ms clock, 0 bytes, 0 allocs
init runtime @0.015 ms, 0.023 ms clock, 0 bytes, 0 allocs
init internal/platform @0.11 ms, 0.045 ms clock, 3056 bytes, 3 allocs
init math @0.17 ms, 0.002 ms clock, 0 bytes, 0 allocs
init errors @0.18 ms, 0 ms clock, 0 bytes, 0 allocs

[100+ lines omitted...]

init cmd/go/internal/modcmd @1.5 ms, 0.010 ms clock, 7920 bytes, 93 allocs
init cmd/go/internal/modget @1.5 ms, 0.005 ms clock, 5952 bytes, 49 allocs
init cmd/go/internal/vet @1.5 ms, 0.004 ms clock, 6048 bytes, 41 allocs
init cmd/go/internal/workcmd @1.5 ms, 0.003 ms clock, 3648 bytes, 47 allocs
init main @1.6 ms, 0.001 ms clock, 288 bytes, 1 allocs
```

cmd/go is quite large.
How about a small program?



demo!

GODEBUG=inittrace

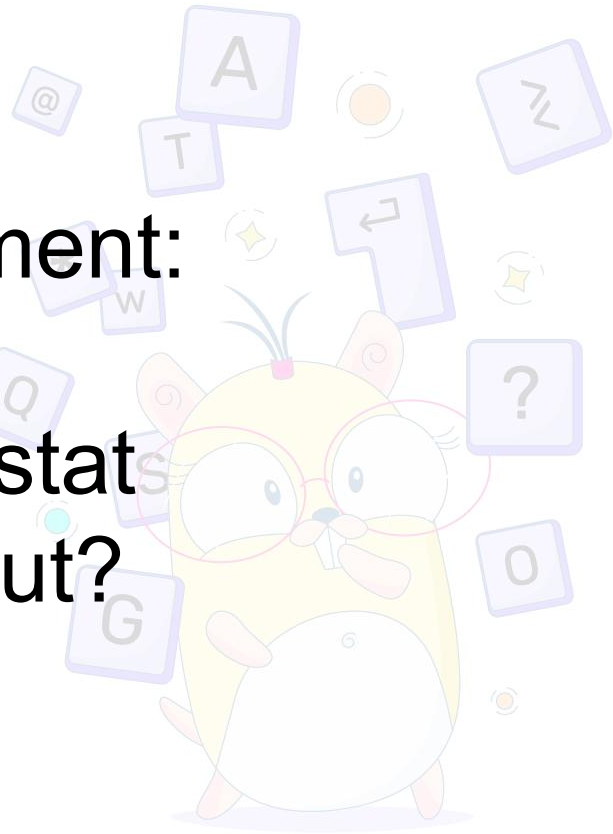
```
$ GODEBUG=inittrace=1 ./main -h
init internal/bytealg @0 ms, 0 ms clock, 0 bytes, 0 allocs
init runtime @0.014 ms, 0.027 ms clock, 0 bytes, 0 allocs
init math @0.096 ms, 0 ms clock, 0 bytes, 0 allocs
init errors @0.10 ms, 0 ms clock, 0 bytes, 0 allocs
[...]
init flag @0.22 ms, 0.003 ms clock, 128 bytes, 2 allocs
init regexp/syntax @0.23 ms, 0.002 ms clock, 3960 bytes, 4 allocs
init regexp @0.25 ms, 0.001 ms clock, 0 bytes, 0 allocs
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs
```

GODEBUG=inittrace

```
$ GODEBUG=inittrace=1 ./main -h
init internal/bytealg @0 ms, 0 ms clock, 0 bytes, 0 allocs
init runtime @0.014 ms, 0.027 ms clock, 0 bytes, 0 allocs
init math @0.096 ms, 0 ms clock, 0 bytes, 0 allocs
init errors @0.10 ms, 0 ms clock, 0 bytes, 0 allocs
[...]
init flag @0.22 ms, 0.003 ms clock, 128 bytes, 2 allocs
init regexp/syntax @0.23 ms, 0.002 ms clock, 3960 bytes, 4 allocs
init regexp @0.25 ms, 0.001 ms clock, 0 bytes, 0 allocs
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs
```

GODEBUG=inittrace

```
$ GODEBUG=inittrace=1 ./main -h
init internal/bytealg @0 ms, 0 ms clock, 0 bytes, 0 allocs
init runtime @0.014 ms, 0.027 ms clock, 0 bytes, 0 allocs
init math @0.096 ms, 0 ms clock, 0 bytes, 0 allocs
init errors @0.10 ms, 0 ms clock, 0 bytes, 0 allocs
[...]
init flag @0.22 ms, 0.003 ms clock, 128 bytes, 2 allocs
init regexp/syntax @0.23 ms, 0.002 ms clock, 3960 bytes, 4 allocs
init regexp @0.25 ms, 0.001 ms clock, 0 bytes, 0 allocs
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs
```



Measuring improvement:

Can we use benchstat
on inittrace=1 output?

GODEBUG=inittrace

```
$ GODEBUG=inittrace=1 ./main -h
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs


$ go test -bench=.
BenchmarkIsNumber-16  313450  3673 ns/op
```

GODEBUG=inittrace

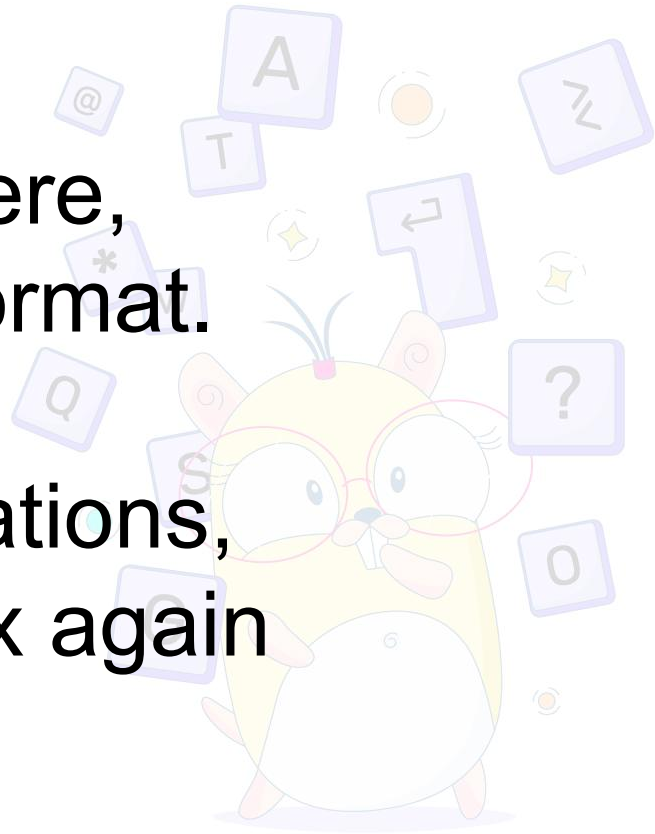
```
$ GODEBUG=inittrace=1 ./main -h
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs


$ go test -bench=.
BenchmarkIsNumber-16  313450  3673 ns/op
```

GODEBUG=inittrace

```
$ GODEBUG=inittrace=1 ./main -h
init main @0.25 ms, 0.062 ms clock, 42912 bytes, 264 allocs


$ go test -bench=.
BenchmarkIsNumber-16   313450   3673 ns/op

$ go test -bench=. -benchmem
BenchmarkIsNumber-16   318336   3659 ns/op   5435 B/op   73 allocs/op
```

All the data is there,
just in a different format.

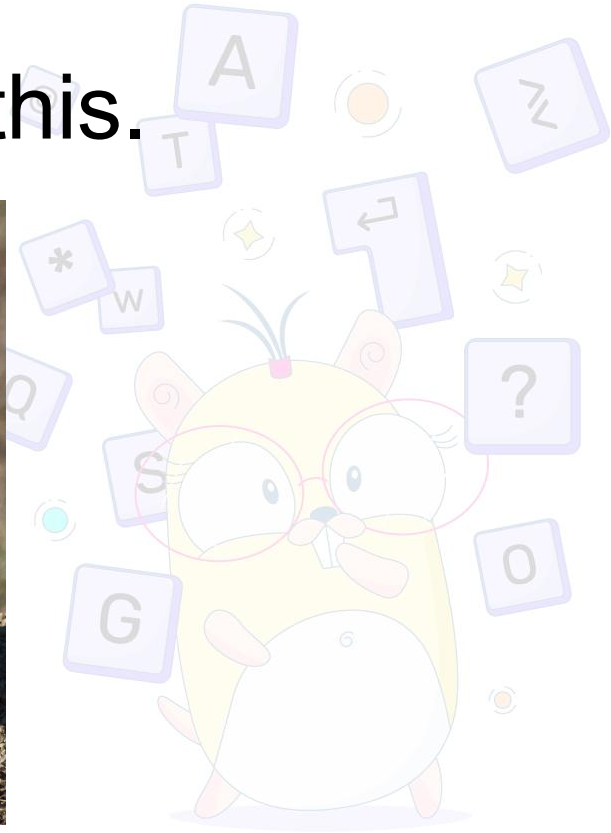
We need more iterations,
or it's -benchtime=1x again



GODEBUG=inittrace

```
$ for n in {1..10}; do GODEBUG=inittrace=1 ./main -h |& grep
'^init main'; done
init main @0.21 ms, 0.069 ms clock, 42992 bytes, 264 allocs
init main @0.15 ms, 0.081 ms clock, 42992 bytes, 264 allocs
init main @0.11 ms, 0.050 ms clock, 43008 bytes, 264 allocs
init main @0.11 ms, 0.057 ms clock, 42992 bytes, 264 allocs
init main @0.12 ms, 0.052 ms clock, 43008 bytes, 264 allocs
init main @0.10 ms, 0.046 ms clock, 42992 bytes, 264 allocs
init main @0.13 ms, 0.046 ms clock, 43008 bytes, 264 allocs
init main @0.11 ms, 0.050 ms clock, 42992 bytes, 264 allocs
init main @0.12 ms, 0.045 ms clock, 42992 bytes, 264 allocs
init main @0.15 ms, 0.090 ms clock, 43008 bytes, 264 allocs
```

I'm lazy; automate this.



benchinit

mvdan.cc/benchinit

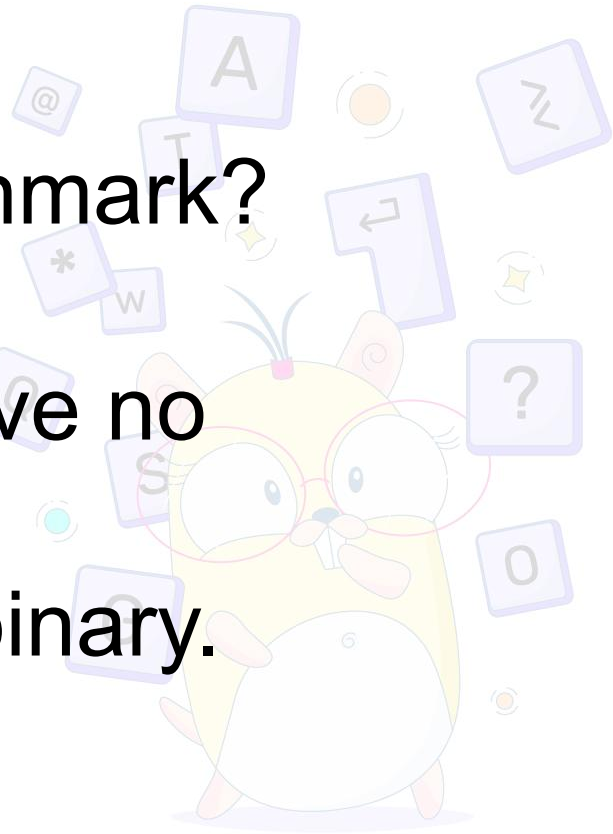
- Take packages as arguments
- Generate a Go benchmark to run
- Benchmark loop executes test binary with GODEBUG=inittrace=1
- Parse each inittrace output and generate a new benchmark result

GODEBUG=inittrace

```
init main @0.21 ms, 0.069 ms clock, 42992 bytes, 264 allocs
init main @0.15 ms, 0.081 ms clock, 42992 bytes, 264 allocs
init main @0.11 ms, 0.050 ms clock, 43008 bytes, 264 allocs
init main @0.11 ms, 0.057 ms clock, 42992 bytes, 264 allocs
[...]


$ benchinit .
BenchmarkTestDemo1  1576  43756 ns/op 43008 B/op 264 allocs/op
```

Why generate a benchmark?

Library packages have no
"main" binaries.
Plus, reuse the test binary.



We can use benchinit with benchstat now!

demo!

A real life large program: cmd/cue

demo!

How do we get a useful cpu or memory profile from this?

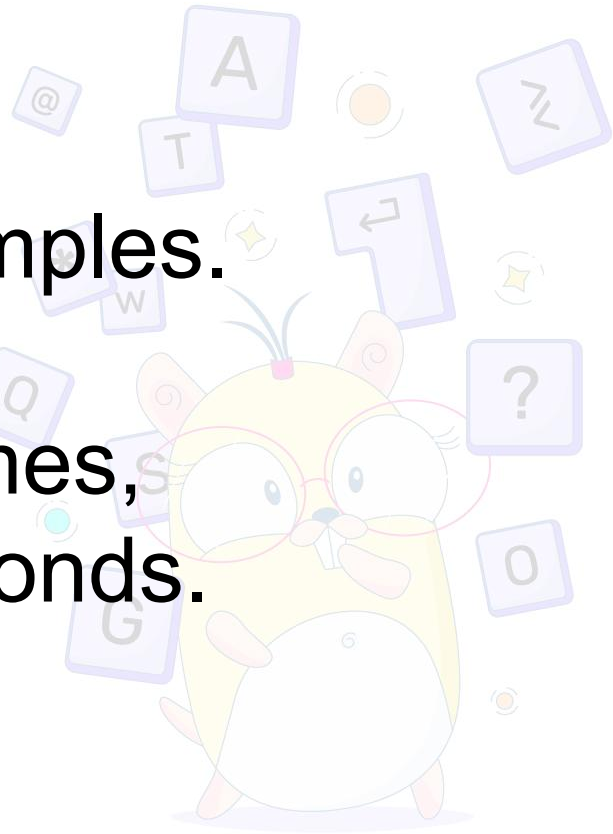
We can with Go benchmarks:
-cpuprofile -memprofile

Can we use pprof with init?

Not really; pprof is implemented in "go test".

pprof is based on samples.

Not great for init times,
measured in milliseconds.



One alternative on Linux:

perf

demo!

earlier attempt

thanks to
Josh Bleecher Snyder

commaok.xyz/post/
benchmark-init

Jan 2019, before GODEBUG=inittrace=1

Regular Go benchmark with a loop:

- Force the runtime to mark a package as not initialized
- Force the runtime to initialize said package

Go 1.12 runtime hackery

```
var initdone· uint8
func init() {
    if initdone· > 1 { return }
    if initdone· == 1 { throw() }
    initdone· = 1
    // for all imported packages {
          pkg.init()
    // }
    init.ializers()
    init.<n>() // call user init functions, if any
    initdone· = 2
    return
}
func init.ializers() { /* init global variables for this package */ }
```

Go 1.12 runtime hackery

```
var initdone· uint8
func init() {
    if initdone· > 1 { return }
    if initdone· == 1 { throw() }
    initdone· = 1
    // for all imported packages {
          pkg.init()
    // }
    init.ializers()
    init.<n>() // call user init functions, if any
    initdone· = 2
    return
}
func init.ializers() { /* init global variables for this package */ }
```

Go 1.12 runtime hackery

```
var initdone· uint8
func init() {
    if initdone· > 1 { return }
    if initdone· == 1 { throw() }
    initdone· = 1
    // for all imported packages {
          pkg.init()
    // }
    init.ializers()
    init.<n>() // call user init functions, if any
    initdone· = 2
    return
}
func init.ializers() { /* init global variables for this package */ }
```

Go 1.12 runtime hackery

```
import _ "net/http" // we need net/http.init in the executable
import _ "unsafe"   // must import unsafe to use go:linkname

//go:linkname _initdone net/http.initdone·
var _initdone uint8
//go:linkname _init net/http.init
func _init()

func BenchmarkNetHTTPInit(b *testing.B) {
    for i := 0; i < b.N; i++ {
      _initdone = 0
      _init()
    }
}
```

This kinda worked!

Except not: global state.

main packages

```
import "flag"

var v = flag.Int("v", 0, "")

func main() {
    flag.PrintDefaults()
}
```

```
$ go run main.go
 -v int
```

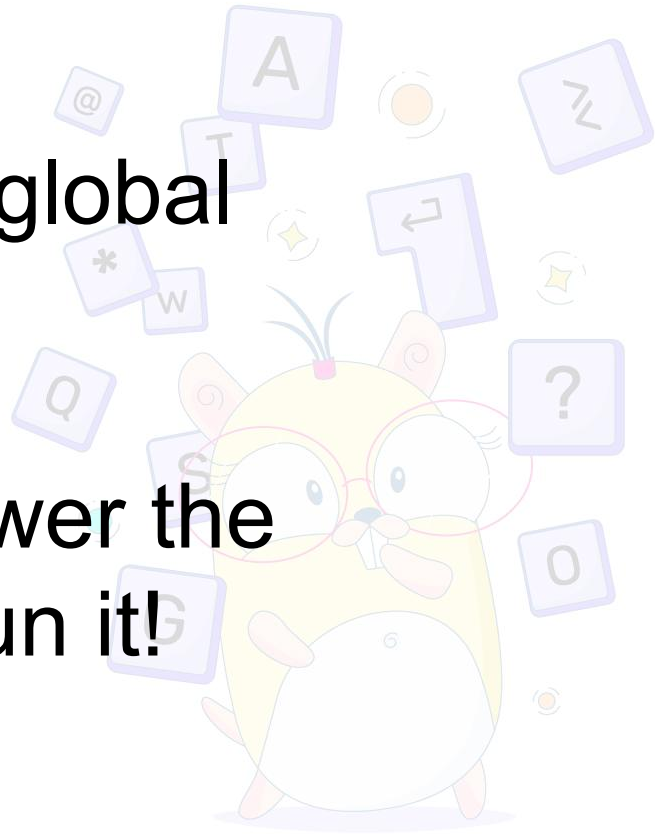
main packages

```
import "flag"

var v = flag.Int("v", 0, "")
var v2 = flag.Int("v", 0, "")

func main() {
    flag.PrintDefaults()
}
```

```
$ go run main.go
panic: flag redefined: v
```

If init appends to a global variable…

Benchmark gets slower the more times you run it!



Worth noting:

It was a regular benchmark,
so pprof actually worked!

Still, a flawed design.

pprof support

potential next step with inittrace=1?

- Start the profile (where? how?) after initializing runtime/pprof
- Stop and output the profile at main (how? instrumentation?)
- Collect profiles from many exec invocations and merge them

Questions, ideas, complaints?

Daniel Martí

github.com/mvdan