

The background of the slide is a green pixelated pattern. It consists of a grid of squares in various shades of green, ranging from dark forest green to bright lime green. On the right side, there is a cluster of white squares of varying sizes, some of which are slightly blurred, giving a sense of depth or a 'glitch' effect.

Goodies

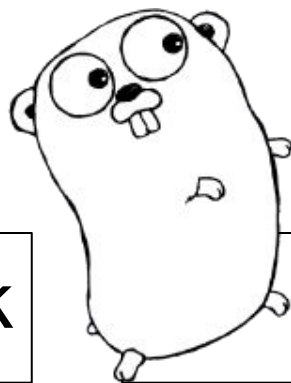
For Go programmers



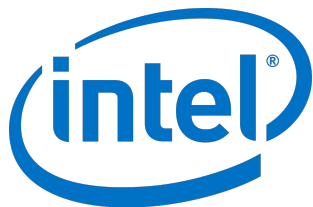
gocritic



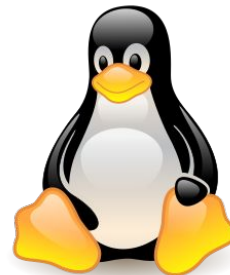
lintpack



golang



open source




@quasilyte




go list

**Query imports/deps
(built-in)**




```
# List strings package test files.  
$ go list -f '{{ .XTestGoFiles }}' strings  
[builder_test.go compare_test.go  
example_test.go reader_test.go  
replace_test.go search_test.go  
strings_test.go]
```

go list example 1



```
# List strings package direct imports.  
$ go list -f '{{ .Imports }}' strings  
[errors internal/bytealg io sync unicode  
unicode/utf8 unsafe]
```

go list example 2



```
# List all strings package dependencies.  
$ go list -f '{{ .Deps }}' strings  
[errors internal/bytealg internal/cpu  
internal/race io runtime  
runtime/internal/atomic runtime/internal/math  
runtime/internal/sys sync sync/atomic unicode  
unicode/utf8 unsafe]
```

go list example 3

Usage hints

- Use from programs working on Go packages to query info about package structure.
- Integrate into CI and check a number of project dependencies or write a rule to forbid some particular import.



go list: learn more


<https://dave.cheney.net/2014/09/14/go-list-your-swiss-army-knife>



gogrep

Go syntax-aware grep

[mvdan/gogrep](https://github.com/mvdan/gogrep)



```
# Find non-idiomatic value swaps in stdlib.  
$ q='$tmp := $x; $x = $y; $y = $tmp'  
$ gogrep -x $q $(go list std)
```

```
runtime/pprof/internal/profile/profile.go:214:3:  
    tmp := p.Mapping[i]  
    p.Mapping[i] = p.Mapping[0]  
    p.Mapping[0] = tmp
```

gogrep example 1



```
# Using type constraints.
```

```
$ stdlib=$(go list std)
```

```
$ q='$a+$b+$c+$d'
```

```
$ gogrep -x $q $stdlib | wc -l
```

```
4383
```

```
$ gogrep -x $q -a 'type(int)' $stdlib | wc -l
```

```
37
```

gogrep example 2

Usage hints

Search for particular Go pattern over a big code base:

- Find more idiomatic form among alternatives.
- Prototype static analysis checks.
- Lookup for non-performant pieces.



go-consistent

Check code consistency

[Quasilyte/go-consistent](https://quasilyte.github.io/go-consistent/)



```
$ go-consistent bufio
```

```
$GOROOT/src/bufio/bufio.go:588:6:
```

```
range check: use align-center, like in `low < x && x < high`
```

```
$GOROOT/src/bufio/bufio_test.go:1165:7:
```

```
zero value ptr alloc: use new(T) for *T allocation
```

go-consistent example

Usage hints

- Run on a file to check file-local consistency.
- Run on a package to check intra-package consistency.
- Run on a project to check global consistency.



gocovmerge

Merge coverage profiles

[wadey/gocovmerge](https://github.com/wadey/gocovmerge)

Why?

To combine coverage profiles collected by a different methods. For example, unit tests plus specially-cooked “main” tests.

See [Measure End to End Tests Coverage](#).



benchstat


Compare bench results

[x/perf/cmd/benchstat](#)

Usage scheme

1. Collect “old” benchmark results.
2. Collect “new” benchmark results.
3. Use benchstat to compare them.





```
# Step 0: apply optimizations.  
$ go test -bench=. -count=10 . | tee new.txt  
# Now roll them back.  
$ git stash save  
$ go test -bench=. -count=10 . | tee old.txt  
$ benchstat old.txt new.txt
```

Using benchstat

Outputs results in a readable table format.

```
quasilyte strconv $ benchstat old.txt new.txt
name          old time/op  new time/op  delta
Atof64Decimal-8  48.2ns ± 0%  49.6ns ± 0%  +2.88%
Atof64Float-8    55.8ns ± 0%  57.6ns ± 0%  +3.23%
Atof64FloatExp-8 131ns ± 6%   124ns ± 0%  -5.27%
```

Using benchstat

go-benchrun

Simplifies benchstat usage

[Quasilyte/go-benchrun](https://github.com/Quasilyte/go-benchrun)

Usage scheme (vs benchstat)

- ~~1. Collect “old” benchmark results.~~
 - ~~2. Collect “new” benchmark results.~~
 - ~~3. Use benchstat to compare them.~~
- Just run “go-benchrun”.



When it's useful?

If you would like to keep both “old” and “new” versions of code in a same branch/worktree.


See “workflow” section in the go-benchrun documentation for more details.



upx


Compress (any) binaries

[upx/upx](#)



```
// hello.go
package main
import "fmt"
func main() {
    fmt.Println("hello, world")
}
```

Hello world program



```
$ go build -o old hello.go
$ upx new old
$ bincmp -no-symtab old new
```

binary	delta	old	new	
new	-982951	1988411	1005460	-49.43%

Building and compressing

The cost

That compression is not entirely free.

During the application start, there will be some overhead due the binary decompression.



bincmp


Compare (Go) binaries

[tzneal/bincmp](https://github.com/tzneal/bincmp)

Not just binary size comparison

- Shows Go functions size (old/new).
- Displays binary segments sizes.
- Uses output format close to the benchstat.






```
$ go build -o old hello.go
$ go build -o new -ldflags="-w" hello.go
$ bincmp old new
```

(Results are on a next slide.)

Stripping debug info



name	delta	old	new
.debug_abbrev	-467	467	
.debug_frame	-74428	74428	
.debug_info	-463938	463938	
.debug_line	-85159	85159	
.debug_loc	-388201	388201	
-47.57%			

Binary sections diff (shortened)

golangci-lint

Easy-to-use linters bundle

[golangci/golangci-lint](https://github.com/golangci/golangci-lint)

Benefits

- Simpler to use and install.
- Has various integrations.
- Runs best Go static analyzers.
- Unifies different linters output.
- Makes linters re-use AST/type data.





Includes github.com/go-critic/go-critic

gometalinter

As an alternative, there is also [gometalinter](#), but:

- It's slower.
- Includes less linters.
- Less features (lacks “diff” checking, etc).



go-unexport

Minimize the public API

[Quasilyte/go-unexport](https://github.com/Quasilyte/go-unexport)

How unexport works

Runs “gorename” for every exported package symbol. So, after a run over a package, everything that can be safely unexported, will be unexported. All you have to do is to commit changes.

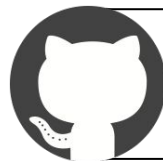


When it's useful?

Inside big projects with a lot of Go code and internal packages. It's OK to reduce internal package API if it's not used anywhere, since the only clients is your own code. In other words, it works well for CLI apps and mono-repos.



The end
Any questions?



@quasilyte