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 isses, 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

```go
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

```go
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`

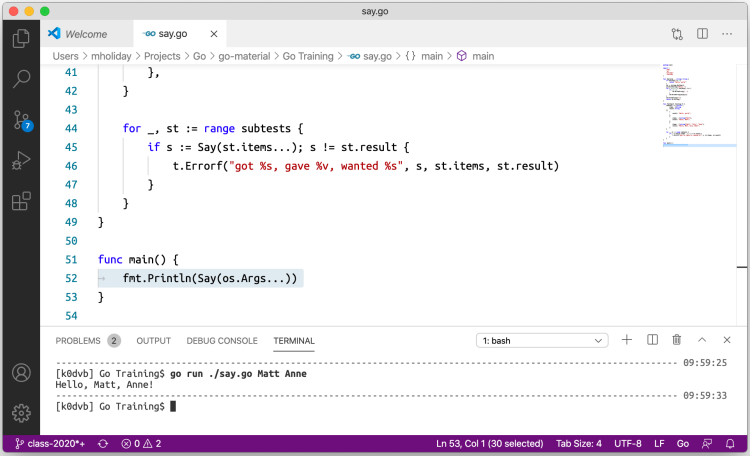
It can be configured with `.golangci.yml`

We use this in our CI/CD pipeline

Issues must be fixed for the build to pass

False positivescan be marked with `//nolint`

Visual Studio Code



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

offers-inventory-management – cache.go

offers-inventory-management › cache.go

make build/lint/test

Project

- offers-inventory-management
  - .github
  - cmd
  - docs
  - offer
  - schema
  - testdata
  - ui
  - .drone-v1.yml
  - .gcloudignore
  - .gitignore
  - .golangci.yml
  - .graphqlconfig
  - api.go
  - api_integration_test.go
  - api_test.go
  - auth.go
  - auth_test.go
  - cache.go
  - cache_test.go
  - coverprofile.out
  - db.go
  - db_test.go
  - editors.go
  - filter.go
  - filter_test.go
  - GetToken.sh
  - go.mod
  - graphql.config.json
  - home.go

auth.go | cache.go

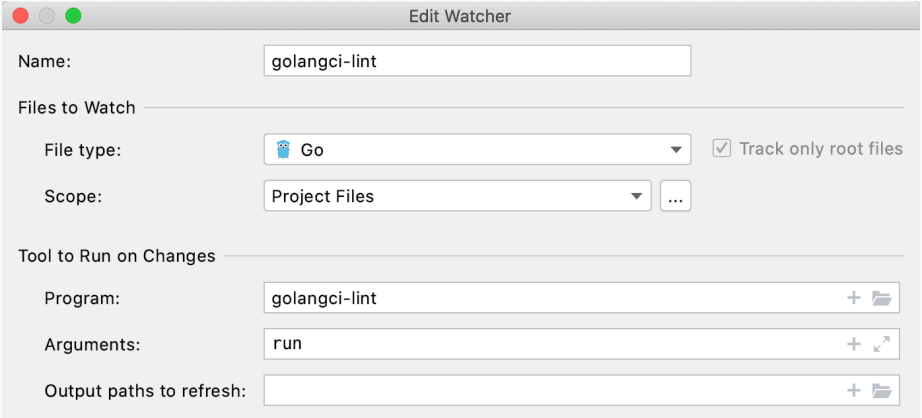
```go
// cachedPlatoClient is a plato client with responses cached.
type cachedPlatoClient struct {
    plato
    logger *leveledLog
}

var (
    defaultCountries = []string{"US", "CA", "GB", "FR", "AU", "IN"}

    // these are variables so we can reduce them in unit tests

    baseLifetime = 5 * time.Hour    // how long a cache entry lives (randomized later)
    reloadTime   = 20 * time.Minute // time considered "soon to expire", i.e., reload it now
    checkTime    = reloadTime / 2   // time between periodic DB/cache checks

    // these are used to create a single connection pool for Redis

    pool *redis.Pool
    once sync.Once
)

func newCachedPlatoClient(logger *leveledLog, client *platoClient) (*cachedPlatoClient, error) {
    // we will start the pool and test our connection
    // before we say we've got a working caching client

    if err := redisTest(getConn()); err != nil {
        return nil, fmt.Errorf( format: "redis failed: %w", err)
```

Git | TODO | Terminal | Event Log

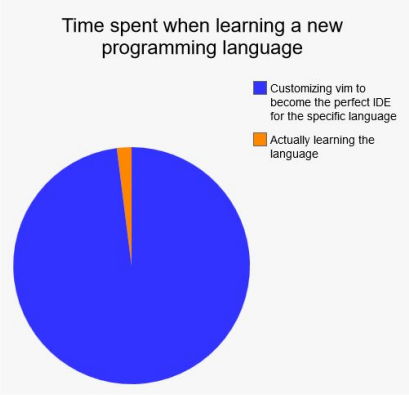
27:1 LF UTF-8 Tab offers-370-error

Sample GoLand settings

The formatting and linting tools are configured as file watchers in GoLand



Vim



Vim setup example