

Если вы видите что-то необычное, просто сообщите мне.

Инструкция для линтинга Go программ

Линтинг это процесс обнаружения и оповещения различных шаблонов найденных в коде, с целью улучшения состояния, и отлавливания багов на ранних стадиях разработки. Это обычно полезно при работе в команде, так как помогает делать весь код одинаковым независимо от того кто именно пишет, убирать излишнюю сложность, и делать код легче в обслуживании. В этой статье, я расскажу со всех сторон о настройке линтинга для Go программ, и поговорим о лучших способах введения их в эксплуатацию.

Линтинг кода один из самых простых вещей, которые вы можете делать чтобы убедиться в содержании кодовых практик в проекте. Go уже заходит дальше других языков программирования используя `gofmt`, инструмент форматирования который проверяет, что весь код go выглядит одинаково, но правда работает только с тем, как код выглядит. Инструмент `vet` go языка так же может помочь с определением странных конструкций, которые могут быть пойманы компилятором, но он отлавливает только ограниченное количество потенциальных проблем.

Задача разработки более всесторонних линтинг инструментов была оставлена на широкую общественность, и общество уже принесло гору линтеров, каждый для своей цели.

Известные примеры включают в себя:

- `unused` - Проверка GO кода на неиспользуемые константы, переменные, функции и типы.
- `goconst` - Нахождение повторяющихся строк, которые могут быть заменены константой.
- `gosyclo` - Вычисления и проверки циклической сложности функций.
- `errcheck` - Обнаружение непроверяемых ошибок в программе на Go

Проблема в наличии такого большого набора отдельных инструментов, в том, что вам нужно качать каждую по отдельности и управлять их версиями. В добавок, запуск каждой из них по очереди может быть очень медленно. [golangci-lint](#) сборщик, который запускает линтеры в паралели, переиспользует Go кэш, и хранит результат анализа для того, чтобы улучшить производительность паралельного запуска, является одним из предпочитаемым способом для настройки линтера в Go проекте.

Проект `golangci-lint` был разработан для сбора и запуска нескольких отдельных линтеров в паралели, для удобства и производительности. Когда вы ставите программу, вы получаете порядка 48 линтеров включительно(во время написания), и вы можете продолжит выбирать какой из них важный для вашего проекта. Кроме того во время их локального запуска, есть возможность настроих их в качестве шага CI.

Установка golangci-lint

Команда ниже используется для устаовки `golangci-lint` локально на любую операционную систему.

```
$ go install github.com/golangci/golangci-lint/cmd/golangci-lint@latest
```

После установки, вы должны проверить версию пакета:

```
$ golangci-lint version
golangci-lint has version v1.40.1 built from (unknown, mod sum:
"h1:pBrCqt9BgI9LfGCTKRTSe1DfMjR6BkOPeRPaXJYXA6Q=") on (unknown)
```

Вы можете так же посмотреть все доступные линтер следующей командой:

```
$ golangci-lint help linters
```

```
ayoy at Kreig in ~/d/r/spotcli (master|.)
> golangci-lint help linters
Enabled by default linters:
deadcode: Finds unused code [fast: false, auto-fix: false]
errcheck: Errcheck is a program for checking for unchecked errors in go programs. These unchecked errors can be critical bugs in some cases [fast: false, auto-fix: false]
gosimple (megacheck): Linter for Go source code that specializes in simplifying a code [fast: false, auto-fix: false]
govet (vet, vetshadow): Vet examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string [fast: false, auto-fix: false]
ineffassign: Detects when assignments to existing variables are not used [fast: true, auto-fix: false]
staticcheck (megacheck): Staticcheck is a go vet on steroids, applying a ton of static analysis checks [fast: false, auto-fix: false]
structcheck: Finds unused struct fields [fast: false, auto-fix: false]
typecheck: Like the front-end of a Go compiler, parses and type-checks Go code [fast: false, auto-fix: false]
unused (megacheck): Checks Go code for unused constants, variables, functions and types [fast: false, auto-fix: false]
varcheck: Finds unused global variables and constants [fast: false, auto-fix: false]

Disabled by default linters:
asciicheck: Simple linter to check that your code does not contain non-ASCII identifiers [fast: true, auto-fix: false]
bodyclose: checks whether HTTP response body is closed successfully [fast: false, auto-fix: false]
cyclop: checks function and package cyclomatic complexity [fast: false, auto-fix: false]
depguard: Go linter that checks if package imports are in a list of acceptable packages [fast: false, auto-fix: false]
dogsled: Checks assignments with too many blank identifiers (e.g. x, _, _, _, := f()) [fast: true, auto-fix: false]
dupl: Tool for code clone detection [fast: true, auto-fix: false]
durationcheck: check for two durations multiplied together [fast: false, auto-fix: false]
errorlint: errorlint is a linter for that can be used to find code that will cause problems with the error wrapping scheme introduced in Go 1.13. [fast: false, auto-fix: false]
exhaustive: check exhaustiveness of enum switch statements [fast: false, auto-fix: false]
exhaustivestruct: Checks if all struct's fields are initialized [fast: false, auto-fix: false]
exportloopref: checks for pointers to enclosing loop variables [fast: false, auto-fix: false]
forbidigo: Forbids identifiers [fast: true, auto-fix: false]
forcetypeassert: finds forced type assertions [fast: true, auto-fix: false]
funlen: Tool for detection of long functions [fast: true, auto-fix: false]
gci: Gci control golang package import order and make it always deterministic. [fast: true, auto-fix: true]
gochecknoglobals: check that no global variables exist [fast: true, auto-fix: false]
gochecknoimports: Checks that no init functions are present in Go code [fast: true, auto-fix: false]
gocognit: Computes and checks the cognitive complexity of functions [fast: true, auto-fix: false]
goconst: Finds repeated strings that could be replaced by a constant [fast: true, auto-fix: false]
gocritic: Provides many diagnostics that check for bugs, performance and style issues. [fast: false, auto-fix: false]
gocyclo: Computes and checks the cyclomatic complexity of functions [fast: true, auto-fix: false]
godot: Check if comments end in a period [fast: true, auto-fix: true]
godox: Tool for detection of FIXME, TODO and other comment keywords [fast: true, auto-fix: false]
goerr113: Golang linter to check the errors handling expressions [fast: false, auto-fix: false]
gofmt: Gofmt checks whether code was gofmt-ed. By default this tool runs with -s option to check for code simplification [fast: true, auto-fix: true]
```

Если вы запустили включенные линтеры в корне проекта, вы можете увидеть ошибки.

Каждая проблема содержит в себе описание того, что именно нужно исправить, и содержит краткое описание проблемы, а так же файл и строку с проблемой.

```
$ golangci-lint run # equivalent of golangci-lint run ./...
```

```
ayoy at Kreig in ~/d/r/spotcli (master|.)
> golangci-lint run
src/user.go:84:11: Error return value of `c.AddFunc` is not checked (errcheck)
    c.AddFunc("@every 30m", func() {
      ^
src/history.go:138:39: loopclosure: loop variable v captured by func literal (govet)
    ab, err := spotifyClient.GetAlbums(v ... )
                                   ^
src/history.go:202:40: loopclosure: loop variable v captured by func literal (govet)
    at, err := spotifyClient.GetArtists(v ... )
                                   ^
ayoy at Kreig in ~/d/r/spotcli (master|.)
> █
```

Вы можете выбрать, какую папку и файлы необходимо проанализировать указав в команде:

```
$ golangci-lint run dir1 dir2 dir3/main.go
```

Настройка golangci-lint

GolangCI-Lint разработан быть гибким насколько это возможно, для различного набора случаев. Настройка `golangci-lint` может управляться через параметры командной строки или файл настройки, так же имеет преимущества старого над новым если один параметр задается несколько раз. Вот пример который использует параметры командной строки для отключения всех линтеров и указания определенных линтеров, которые должны запускаться.

```
$ golangci-lint run --disable-all -E revive -E errcheck -E nilerr -E gosec
```

Вы можете так же запустить с настройками по-умолчанию из `golangci-lint`. Вот как найти информацию о доступных настройках.

```
$ golangci-lint help linters | sed -n '/Linters presets:/,$p'
```

Linters presets:

bugs: asciicheck, bodyclose, durationcheck, errcheck, errorlint, exhaustive, exportloopref, gosec, govet, makezero, nilerr, noctx, rowserrcheck, scopelint, sqlclosecheck, staticcheck, typecheck

comment: godot, godox, misspell

complexity: cyclop, funlen, gocognit, gocyclo, nestif

error: errcheck, errorlint, goerr113, wrapcheck

format: gci, gofmt, gofumpt, goimports

import: depguard, gci, goimports, gomodguard

metalinter: gocritic, govet, revive, staticcheck

module: depguard, gomoddirectives, gomodguard

performance: bodyclose, maligned, noctx, prealloc

sql: rowserrcheck, sqlclosecheck

style: asciicheck, depguard, dogsled, dupl, exhaustivestruct, forbidigo, forcetypeassert, gochecknoglobals,

gochecknoinits, goconst, gocritic, godot, godox, goerr113, goheader, golint, gomnd, gomoddirectives,

gomodguard, goprintfuncname, gosimple, ifshort, importas, interfacer, lll, makezero, misspell, nakedret,

nlreturn, nolintlint, paralleltest, predeclared, promlinter, revive, stylecheck, tagliatelle, testpackage, thelper,

```
tparallel, unconvert, wastedassign, whitespace, wrapcheck, wsl
test: exhaustivestruct, parallelltest, testpackage, tparallel
unused: deadcode, ineffassign, structcheck, unparam, unused, varcheck
```

Теперь можно запустить настройки по-умолчанию передав их имена в ключе `--preset` или `-p`:

```
$ golangci-lint run -p bugs -p error
```

Настройку `golangci-lint` для проекта лучше производить через файл. Таким образом, вы сможете настроить отдельные настройки линтера, которые не доступны из командной строки. Вы можете указать `yaml`, `toml` или `json` формат файла настройки, но я рекомендую остановиться на `yaml` формате, так сказано в оф документации.

Говоря в общем, вы должны создать отдельную настройку для проекта в корне. Программа автоматически найдет его в папке и пойдет выше до корня проекта. Это значит вы можете достигнуть глобальной настройки для всех проектов поместив файл с настройками в домашней директории(не советую). Этот файл будет использован если локальные настройки не будут найдены.

Простой файл настройки доступен на веб сайте `golangci-lint` со всеми поддерживаемыми опциями, их описание, и стандартные значения. Вы можете использовать их как остчетную точку при создании своей настройки. Имейте в виду, что некоторые линтеры выполняют похожие функции, поэтому нужно включать линтеры обдуманно, чтобы избежать избыточных записей. Вот общая настройка которую можно использовать для проекта:

```
.golangci.yml
linters-settings:
  errcheck:
    check-type-assertions: true
  goconst:
    min-len: 2
    min-occurrences: 3
  gocritic:
    enabled-tags:
      - diagnostic
      - experimental
      - opinionated
      - performance
```

- style

govet:

check-shadowing: true

nolintlint:

require-explanation: true

require-specific: true

linters:

disable-all: true

enable:

- bodyclose
- deadcode
- depguard
- dogsled
- dupl
- errcheck
- exportloopref
- exhaustive
- goconst
- gocritic
- gofmt
- goimports
- gomnd
- gocyclo
- gosec
- gosimple
- govet
- ineffassign
- misspell
- nolintlint
- nakedret
- prealloc
- predeclared
- revive
- staticcheck
- structcheck
- stylecheck
- thelper
- tparallel
- typecheck

- unconvert
- unparam
- varcheck
- whitespace
- wsl

run:

issues-exit-code: 1

Решение ошибок линтера

Иногда необходимо выключить определенные проблемы линтера, во время работы с кодом. Это можно получить двумя способами: через команду `nolint` и через правила исключения в файле настройки. Давайте посмотрим каждый подход по очереди.

команда nolint

Предположим у нас есть следующий код, который выводит псевдслучайное целое число в стандартный вывод.

```
main.go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())
    fmt.Println(rand.Int())
}
```

Выполнив `golang-lint run` для этого файла мы получим следующий набор ошибок, так как включен `gosec` линтер:

```
$ golangci-lint run -E gosec
main.go:11:14: G404: Use of weak random number generator (math/rand instead of crypto/rand) (gosec)
fmt.Println(rand.Int())
^
```

Линтер поощряет использование метода `Int` из `crypto/rand` взамен, из-за криптографической безопасности. но он имеет бескомпромисно менее дружелюбный API и медленную производительность. Если вас это не беспокоит, вы можете просто проигнорировать ошибку добавив команду `nolint` в нужный файл:

```
func main() {
    rand.Seed(time.Now().UnixNano())
    fmt.Println(rand.Int()) //nolint
}
```

Согласно договоренности, комментарии для машины не должны содержать пробела. поэтому нужно использовать `//nolint` вместо `// nolint`.

Использование `nolint` приведет к тому, что эта найденная проблема будет проигнорированна. Вы можете выключить проблемы определенного линтера указав её имя в команде. Это позволит пропустить строку конкретному линтеру, но остальные пройдутся по ней.

```
main.go
func main() {
    rand.Seed(time.Now().UnixNano())
    fmt.Println(rand.Int()) //nolint:gosec
}
```

Когда вы используете `nolint` команду в начале файла, то он выключает линтер для всего файла:

```
main.go
//nolint:govet,errcheck
package main
```

Вы можете так же отключить линтер для блока кода(например функции), используя `nolint` команду в начале блока кода.

```
main.go
//nolint
func aFunc() {

}
```

После добавления `nolint` команды, рекомендуем, добавить комментарий который объясняет зачем это нужно. Этот комментарий должен быть помещен на строку с флагом:

```
main.go
func main() {
    rand.Seed(time.Now().UnixNano())
    fmt.Println(rand.Int()) //nolint:gosec // for faster performance
}
```

Вы можете указать правила для вашей команды относительно `nolint` комментирования включив `nolintlint` линтер. Он может оповещать проблемы относительно проблем `nolint` без указания определенных отключенных линтеров, или без требования комментария.

```
$ golangci-lint run
main.go:11:26: directive `//nolint` should mention specific linter such as `//nolint:my-linter` (nolintlint)
    fmt.Println(rand.Int()) //nolint
    ^
```

Правила исключения

Правила исключения могут быть указаны в файле настроек для более детального контроля, какие файлы должны быть залинтованы, и о какой проблеме нужно сообщать. Для примера, вы можете выключить определенный линтер из запуска файлов тестов(`_test.go`) или вы можете выключить линтер для всего проекта

```
.golangci.yml
issues:
```

```
exclude-rules:
- path: _test\go # disable some linters on test files

linters:
- gocyclo
- gosec
- dupl

# Exclude some gosec messages project-wide
- linters:
  - gosec
  text: "weak cryptographic primitive"
```

Интеграция в существующий проект

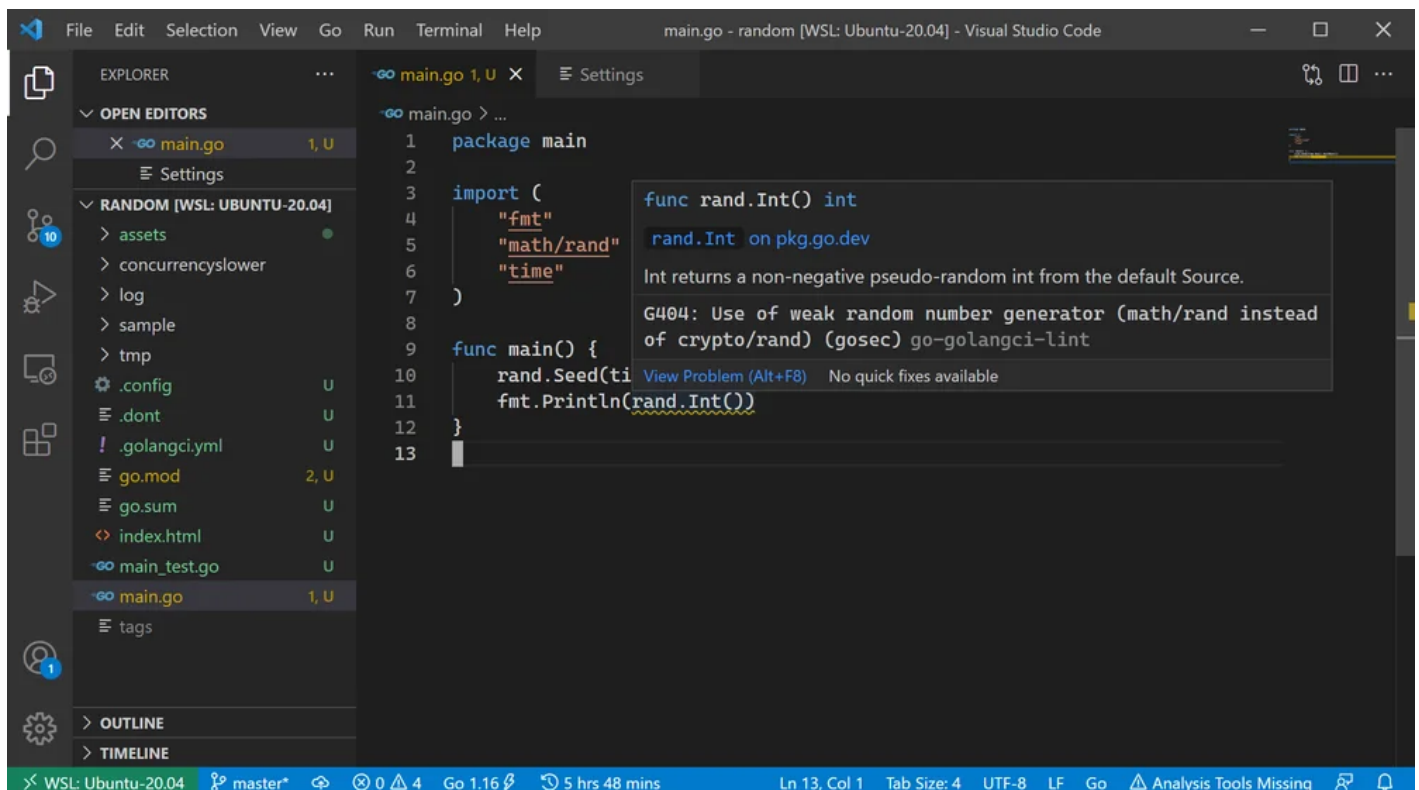
При добавлении `golangci-lint` в существующий проект, вы можете получить большое количество проблем и может быть трудно исправить все их одновременно. Однако, это не значит, что вы должны бросить идею линтинга для вашего проекта по этой причине. When adding `golangci-lint` to an existing project, you may get a lot of issues and it may be difficult to fix all of them at once. However, that doesn't mean that you should abandon the idea of linting your project for this reason. There is a `new-from-rev` setting that allows you to show only new issues created after a specific git revision which makes it easy to lint new code only until adequate time can be budgeted to fix older issues. Once you find the revision you want to start linting from (with git log), you can specify it in your configuration file as follows:

```
.golangci.yml
issues:
  # Show only new issues created after git revision: 02270a6
  new-from-rev: 02270a6
```

Integrating golangci-lint in your editor

GolangCI-Lint supports integrations with several editors in order to get quick feedback. In Visual Studio Code, all you need to do is install the Go extension, and add the following lines to your settings.json file:

```
settings.json
{
  "go.lintTool": "golangci-lint",
  "go.lintFlags": [
    "--fast"
  ]
}
```



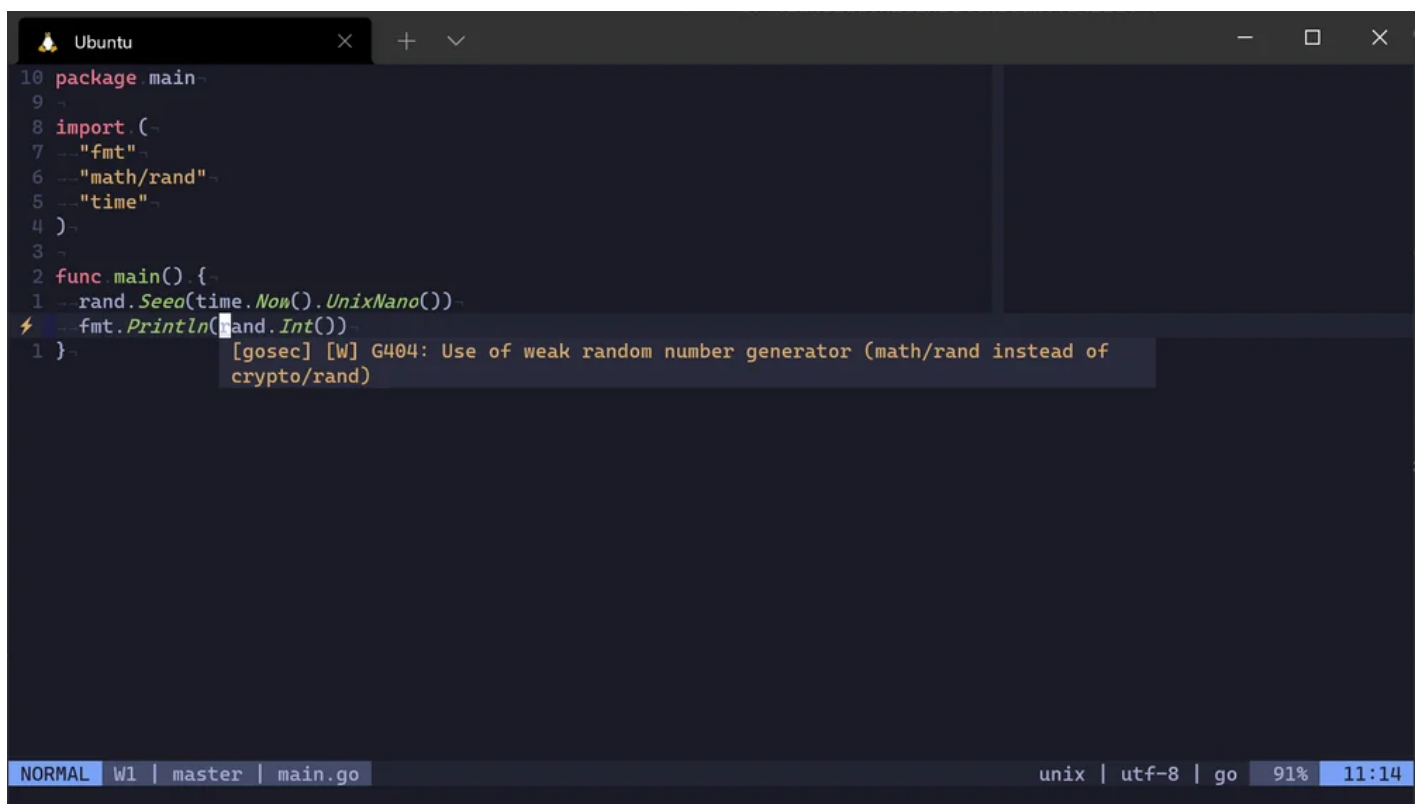
Vim users can integrate golangci-lint with a variety of plugins including vim-go, ALE, and Syntastic. You can also integrate it with coc.nvim, vim-lsp, or nvim.lspconfig with help of golangci-lint-langserver. Here's how I integrated golangci-lint in my editor with coc.nvim. First, install the language server:

```
$ go install github.com/nametake/golangci-lint-langserver@latest
```

Next, open the `coc.nvim` config file with `:CocConfig`, and add the following lines:

```
coc-settings.json
{
  "languageserver": {
    "golangci-lint-languageserver": {
      "command": "golangci-lint-langserver",
      "filetypes": ["go"],
      "initializationOptions": {
        "command": ["golangci-lint", "run", "--out-format", "json"]
      }
    }
  }
}
```

Save the config file, then restart `coc.nvim` with `:CocRestart`, or open a new instance of Vim. It should start working as soon as a Go file is open in the editor.

A screenshot of a Vim editor window on an Ubuntu system. The editor is displaying a Go file named `main.go`. The code includes a `package main` declaration, imports for `fmt`, `math/rand`, and `time`, and a `main` function. A linting error is shown as a tooltip: `[gosec] [W] G404: Use of weak random number generator (math/rand instead of crypto/rand)`. The status bar at the bottom indicates the editor is in `NORMAL` mode, on the `W1` window, at the `master` branch, editing `main.go`. The encoding is `unix`, `utf-8`, and the language is `go`. The battery level is at 91% and the time is 11:14.

voila

Refer to the [golangci-lint docs](#) for more information on how to integrate it with other editors.

Setting up a pre-commit hook



Image source

Running `golangci-lint` as part of your Git pre-commit hooks is a great way to ensure that all Go code that is checked into source control is linted properly. If you haven't set up a pre-commit hook for your project, here's how to set one up with pre-commit, a language-agnostic tool for managing Git hook scripts.

Install the `pre-commit` package manager by following the instructions on this page, then create a `.pre-commit-config.yaml` file in the root of your project, and populate it with the following contents:

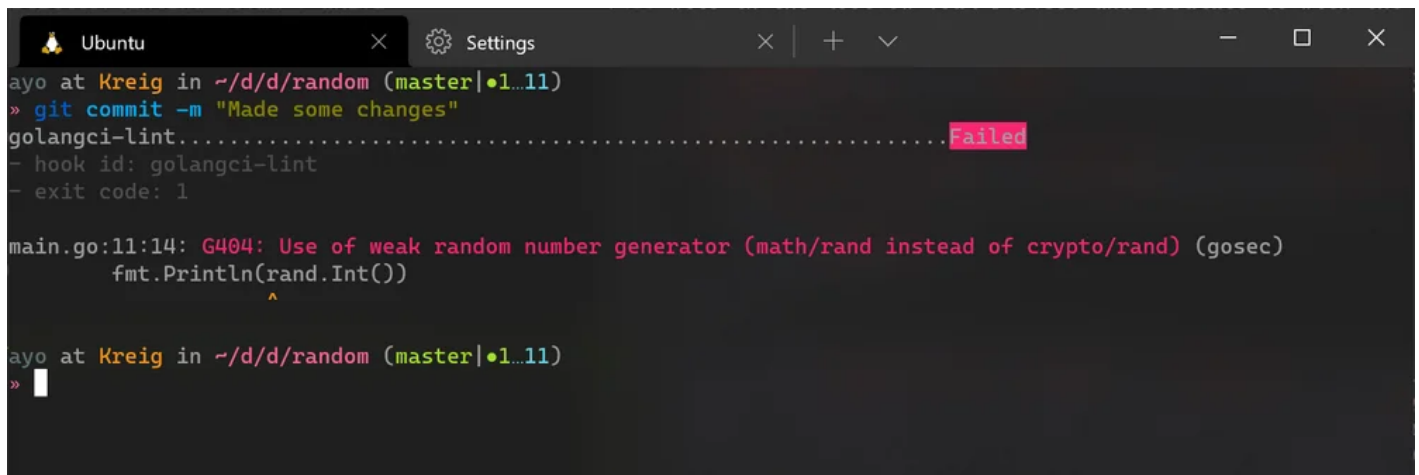
```
.pre-commit-config.yaml
repos:
- repo: https://github.com/tekvizely/pre-commit-golang
  rev: v0.8.3 # change this to the latest version
  hooks:
  - id: golangci-lint
```

This configuration file extends the pre-commit-golang repository which supports various hooks for Go projects. The `golangci-lint` hook targets staged files only, which is handy for when introducing `golangci-lint`

to an existing project so that you don't get overwhelmed with so many linting issues at once. Once you've saved the file, run `pre-commit install` to set up the git hook scripts in the current repository.

```
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

On subsequent commits, the specified hooks will run on all staged .go files and halt the committing process if errors are discovered. You'll need to fix all the linting issues before you'll be allowed to commit. You can also use the `pre-commit run` command if you want to test the pre-commit hook without making a commit.

A terminal window on Ubuntu showing a failed pre-commit hook. The user runs 'git commit -m "Made some changes"'. The 'golangci-lint' hook fails with exit code 1. The error message is 'main.go:11:14: G404: Use of weak random number generator (math/rand instead of crypto/rand) (gosec)'. The code snippet shown is 'fmt.Println(rand.Int())'. The terminal window has tabs for 'Ubuntu' and 'Settings'.

Continuous Integration (CI) workflow

Running your project's linting rules on each pull request prevents code that is not up to standards from slipping through into your codebase. This can also be automated by adding golangci-lint to your Continuous Integration process. If you use GitHub Actions, the official Action should be preferred over a simple binary installation for performance reasons. After setting it up, you'll get an inline display of any reported issues on pull requests.

```
309 +  
310 + func (cl *ContextLoader) testGithubActions() {  
  
X Check failure on line 310 in pkg/lint/load.go  
GitHub Actions / golangci-lint  
pkg/lint/load.go#L310  
func `(*ContextLoader).testGithubActions` is unused (unused)  
  
311 + a := 1  
  
X Check failure on line 311 in pkg/lint/load.go  
GitHub Actions / golangci-lint  
pkg/lint/load.go#L311  
ineffectual assignment to `a` (ineffassign)  
  
312 + a = 2  
313 + cl.log.Infof("a is %d", a)  
314 + }
```

During the setup process, ensure to pin the golangci-lint version that is being used so that it yields consistent results with your local environment. The project is being actively developed, so updates may deprecate some linters, or report more errors than previously detected for the same source code.

Conclusion

Linting your programs is a sure fire way to ensure consistent coding practices amongst all contributors to a project. By adopting the tools and processes discussed in this article, you'll be well on your way to doing just that.

Thanks for reading, and happy coding!

Revision #2

Created 16 April 2022 09:03:30 by gasick

Updated 16 April 2023 19:36:18 by gasick