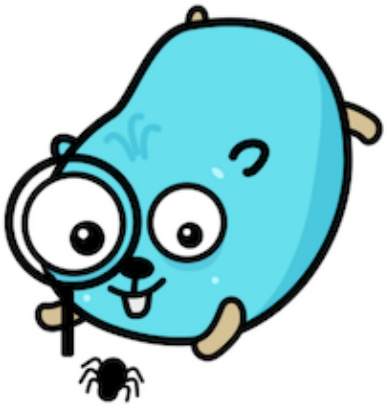
Линтеры и ошибки (часть 1)

В этом уроке мы поговорит о статическом анализе кода применительно к ошибкам в Go:

- какие существуют линтеры и что они проверяют;
- плюсы и минусы использования конкретных линтеров.

Часто, но не всегда, лучшие практики, сформировавшиеся в языке находят отражение в очередном линтере.

Посмотрим, так ли это с ошибками в Go.



Что такое линтер?

Согласно вики:

```
Lint, or a linter, is a static code analysis tool used to flag
programming errors,
bugs, stylistic errors, and suspicious constructs.The term originates
from a Unix

utility that examined C language source code.
```

Т.е. линтером называют программу, производящую статический анализ кода другой программы. Почему "статический"? Потому что анализ происходит именно текста программы, т.е. без её компиляции, запуска и т.д.

Go предоставляет "из коробки" статический анализатор **go vet**:

```
$ go tool vet help
vet is a tool for static analysis of Go programs.

vet examines Go source code and reports suspicious constructs,
such as Printf calls whose arguments do not align with the format
string. It uses heuristics that do not guarantee all reports are
genuine problems, but it can find errors not caught by the compilers.


...
```

Он реализует базовый набор проверок, спасающих Go-разработчика от частых ошибок, невидимых компилятору Go.

Но на данный момент стандартом de facto в статическим анализе Go кода является open source инструмент **golangci-lint** – мощный аккумулятор более чем 40 различных линтеров (включая **go vet**, представленный выше). О некоторых из них мы и поговорим далее.

Тест "nil pointer dereference "

Имеется простой код:

```
package main

import "fmt"

func main()  {
    var i *int
    *i = 5
    fmt.Println(5)
}
```

Какой из инструментов отловит здесь потенциальную ошибку разыменования `nil`-указателя?

Выберите один вариант из списка

- ☐ Компилятор Go
- ☐ go vet
- ☐ И компилятор и go vet
- ☐ Ни компилятор, ни go vet – упадём с паникой в рантайме

go vet: erroras

errorsas report passing non-pointer or non-error values to errors.As

В настоящее время в стандартном анализаторе поддержка ошибок представлена только одной проверкой **erroras**:

```
$ go tool vet help
vet is a tool for static analysis of Go programs.
...
Registered analyzers:
    ...
    errorsas     report passing non-pointer or non-error values to
errors.As

    ...
```

Мы помним, что `errors.As` требует в качестве `target` указатель на интерфейс или значение, реализующее `error`, иначе мы получим одну из паник:

```
func As(err error, target interface{}) bool {
    if target == nil {
        panic("errors: target cannot be nil")
    }

    val := reflectlite.ValueOf(target)
    typ := val.Type()
    if typ.Kind() != reflectlite.Ptr || val.IsNil() {
        panic("errors: target must be a non-nil pointer")
```

```
    }

    if e := typ.Elem(); e.Kind() != reflectlite.Interface &&
!e.Implements(errorType) {
        panic("errors: *target must be interface or implement error")
    }
    // ...
}
```

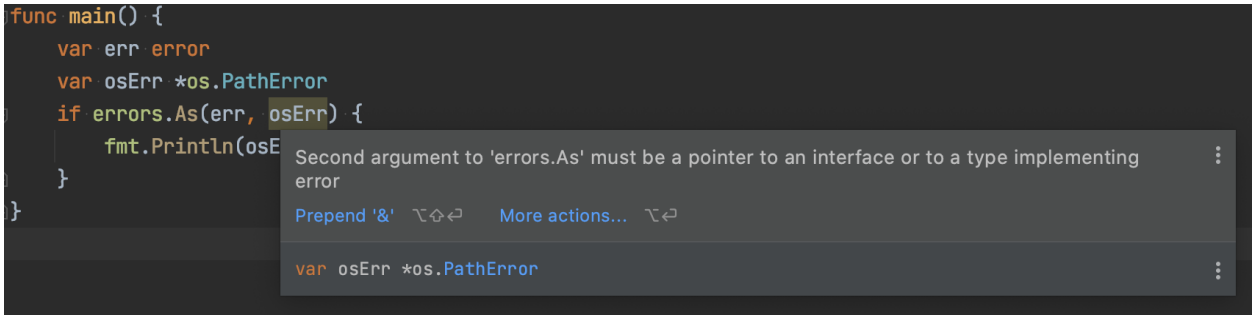
Например:

```
// https://goplay.tools/snippet/z9dSrxB9CWS
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {
    var err error
    var osErr *os.PathError
    if errors.As(err, osErr) { // Казалось бы, уже указатель!
        fmt.Println(osErr.Path) // До этой строчки не дойдёт :(
    }
}

/*
panic: errors: target must be a non-nil pointer

goroutine 1 [running]:
errors.As(0x0, 0x0, 0x10289cb00, 0x0, 0x1400010c058)
        /usr/local/go/src/errors/wrap.go:84 +0x4b4
main.main()
        examples/05-errors-best-practices/govet-erroras/main.go:12
+0x34

*/
```

Запуск `go vet` предупреждает данную ситуацию:

```
$ go vet ./examples/05-errors-best-practices/govet-erroras
main.go:12:5:
    second argument to errors.As must be a non-nil pointer to either

    a type that implements error, or to any interface type
```

IDE в большинстве своём поддерживают **go vet** на уровне инспекций (подсвечивают место кода с потенциальной ошибкой), если же у вас не так, то рекомендуем настроить подобное (скриншот из GoLand):



---

go vet включен в **golangci-lint** по умолчанию, так что нет необходимости вызывать их по отдельности, достаточно только

```
$ golangci-lint run ./examples/05-errors-best-practices/govet-erroras
main.go:12:5:
    errorsas: second argument to errors.As must be a non-nil pointer
    to either a type that implements error, or to any interface type
(govet)

        if errors.As(err, osErr) {
```

errcheck

Errcheck is a program for checking for unchecked errors in go programs.

These unchecked errors can be critical bugs in some cases.

Линтер, **включенный** по умолчанию в **golangci-lint** и проверяющий, что вы не игнорируете возвращаемые ошибки.

Примечательно, что разработчики линтера подчёркивают: ***"These unchecked errors can be critical bugs in some cases".***

Конфигурация линтера выглядит следующим образом:

```
linters-settings:
  errcheck:
    # report about not checking of errors in type assertions: `a := b.(MyStruct)`;
    # default is false: such cases aren't reported by default.
    check-type-assertions: false

    # report about assignment of errors to blank identifier: `num, _ := strconv.Atoi(numStr)`;
    # default is false: such cases aren't reported by default.
    check-blank: false

    # list of functions to exclude from checking, where each entry is a single function to exclude.
    # see https://github.com/kisielk/errcheck#excluding-functions for details
    exclude-functions:
      - io/ioutil.ReadFile
      - io.Copy(*bytes.Buffer)

      - io.Copy(os.Stdout)
```

Мы рекомендуем выставить `check-type-assertions` и `check-black` в `true`, чтобы в каждом конкретном случае задумываться, стоит ли игнорировать ошибку.

В случае с **type assertion** допустимо паниковать, но с более понятным для разработчика текстом, например:

```
a, ok := b.(*MyStruct)
if !ok {
    panic(fmt.Sprintf("invalid some algo realization: expected b type is *MyStruct: got %T", b))

}
```

Также будьте внимательны, что по умолчанию **golangci-lint** работает с включенной опцией **-e/--exclude-use-default**, которая в конфигурации выглядит как:

```
issues:
  # Independently from option `exclude` we use default exclude
patterns,
  # it can be disabled by this option. To list all
  # excluded by default patterns execute `golangci-lint run --help`.
  # Default value for this option is true.

  exclude-use-default: true
```

И согласно этому, **errcheck** не будет проверять ошибки, возвращаемые рядом функций, что может быть опасно:

```
# EXC0001 errcheck: Almost all programs ignore errors on these
functions and in most cases it's ok
- Error return value of
      .((os\.)?std(out|err)\..*|
      .*Close|
      .*Flush|
      os\.Remove(All)?|
      .*print(f|ln)?|

      os\.(Un)?Setenv). is not checked
```

nilerr

nilerr *finds code which returns nil even though it checks that error is not nil.*

Полезный линтер, **выключенный** по умолчанию в **golangci-lint** и спасающий от досадных ошибок, связанных с возвратом `nil` вместо ошибоньки:

```
func f() error {
    if err := do(); err != nil {
        return nil // miss
    }
}

func f() error {
    if  err := do(); err == nil {
```

```
        return err // miss
    }

}
```

Мы не видим причин, чтобы не включать его, а наоборот крайне рекомендуем это делать.

rowserrcheck

rowserrcheck *is a static analysis tool which checks whether* `(*sql.Rows).Err` *is correctly checked.*

Выключен по умолчанию в **golangci-lint.** Проверяет, что вы не забыли обработать ошибку, которая могла произойти при получении очередной порции данных из базули:

```
rows, err := db.Query("select id from users")
if err != nil {
    // ...
}
defer rows.Close()

for rows.Next() {
    // ...
}

// Забыли `if rows.Err() != nil {` !

return nil
```

Несмотря на то, что сейчас уже нечасто встретишь работу с сырым `*sql.Rows`, мы считаем, что линтер крайне полезный и нужно включать. Правда на момент написания курса он в редких случаях выдавал false positives.

go-errorlint

```
go-errorlint is a source code linter for Go software that can be used to find code
```

`that will cause problemswith the error wrapping scheme introduced in Go 1.13.`

Линтер, **выключенный** по умолчанию в **golangci-lint** и проверяющий, что вы пользуетесь возможностями Go 1.13 при работе с ошибками:

```
// bad
err == ErrFoo

// good

errors.Is(err, ErrFoo)

// bad
myErr, ok := err.(*MyError)

// good
var me MyError

ok := errors.As(err, &me)
```

Конфигурация линтера выглядит следующим образом:

```
linters-settings:
  errorlint:
    # Report non-wrapping error creation using fmt.Errorf
    errorf: true
    # Check for plain type assertions and type switches
    asserts: true
    # Check for plain error comparisons

    comparison: true
```

Флаг **-errorf** позволяет требовать от разработчиков использование `%w` при оборачивании ошибок:

```
// bad
fmt.Errorf("oh noes: %v", err)

// good
```

```
fmt.Errorf("oh noes: %w", err)
```

О плюсах и минусах подобного подхода мы говорили ранее на шаге **"Ошибки и границы API"**.

По умолчанию мы рекомендуем делать `errorf: false`.

go-err113

```
Golang linter to check the errors handling expressions.
```

Линтер, частично дублирующий функционал предыдущего линтера **go-errorlint** и созданный после него.

По умолчанию **выключен** в **golangci-lint** и не имеет конфигурации.

Из особенностей:

- Поддерживает отсутствие врапинга для `io.EOF` (хотя **go-errorlint** поддерживает большее количество таких sentinel ошибок, не только `io.EOF`).
- Не поддерживает type switch.
- Не поддерживает `errors.As`.
- Принудительно требует использования `%w` или создания статичных ошибок:

```
// main.go:12:9: do not define dynamic errors, use wrapped static errors instead

return fmt.Errorf("cannot create file %q: %v", f.Name(), e)
```

Исходя из вышесказанного использование данного линтера "как есть" вызывает вопросы.

wrapcheck

`Checks that errors returned from external packages are wrapped.`

Линтер, **выключенный** по умолчанию в **golangci-lint** и проверяющий, что вы оборачиваете в свой текст возвращаемые вами ошибки из стороннего пакета:

```
if _, err := tx.Exec(sql, name, email, city); err != nil {
    // wrapcheck error: error returned from external package is
unwrapped.
    return err
}
```

```
if _, err := tx.Exec(sql, name, email, city); err != nil {
    // %v verb preferred to prevent error becoming part of external
API.
    return fmt.Errorf("failed to insert user: %v", err)

}
```

Примечательно, что автор линтера рекомендует использовать `%v`, а не `%w` – о причинах мы говорили ранее на шаге **"Ошибки и границы API"**.

Конфигурация линтера выглядит следующим образом:

```
wrapcheck:
  # An array of strings which specify substrings of signatures to
ignore. If this
  # set, it will override the default set of ignored signatures. You
can find the
  # default set at the top of ./wrapcheck/wrapcheck.go.
  ignoreSigs:
    - .Errorf(
    - errors.New(
    - errors.Unwrap(
```

```
    - .Wrap(
    - .Wrapf(
    - .WithMessage(
    - .WithMessagef(
    - .WithStack(

  # An array of glob patterns which, if any match the package of the
function
  # returning the error, will skip wrapcheck analysis for this error.
This is
  # useful for broadly ignoring packages and/or subpackages from
wrapcheck
  # analysis. There are no defaults for this value.
  ignorePackageGlobs:
      - encoding/*

      - github.com/pkg/*
```

Как мы видим, она позволяет указать, какие способы врапинга/создания ошибки линтер должен считать валидными.

errwrap

Wrap and fix Go errors with the new `%w` verb directive.

Не входит в **golangci-lint**.

Простенький линтер, помогающий вам обнаружить отсутствие врапинга через `%w`:

```
-       return fmt.Errorf("failed for %s with error: %s", "foo", err)

+       return fmt.Errorf("failed for %s with error: %w", "foo", err)
```

По сути дублирует функционал опции **-errorf** линтера **go-errorlint** (обсуждали несколько шагов назад), поэтому полезность инструмента под вопросом. Но есть интересная заметка от автора: Whether to Wrap or not?.