# Slide 1

Shadowing and errors

2 May 2019

Denys Goldiner

Senior software engineer

SoftServe

### introduction ###

# Slide 2

\* Introduction

.image images/go\_logo.svg.png \_ 1000

: Go молодой язык программирования, разработанный что бы вы могли просто изложить свою идею

: и сделать ваш код простым, понятным и надежным

: однако некоторые возможности нашего языка программирования могут вызвать сложности которые дадут обратный эффект

# Slide 3

\* Real experience

- making mistakes

- proceed analysis

- generate rules

- prevent mistakes

Experience is the ability to prevent mistakes

: ключевым моментом тут является опыт

: опыт состоит из ошибок, анализа и желания не совершать ошибок вновь

: опыт - это умение предовращать ошибки

# Slide 4

\* Base checklist

- check shadowing

- check returns

- check error handling

- check defers

- check closures

- ...

: что бы помочь не допускать ошибок другим членам вашей команды и делиться с ними опытом

: нужен процесс ревью MR

: И для того что бы сделать его максимально эффективным, мы зачастую делаем

: check list. В мой чек лист входит много пунтков

: вот некоторые из них о которых я бы хотел поговорить

: давайте разбираться, почему они заслужили там место

### Shadowing ###

# Slide 5

\* What is variable shadowing

.play examples/example1.go /^func main/,/^}/

: для начала что вообле такое variable shadowing

: если запустить данный код, мы увидим что переменная v внутри условия и вне его имеет разное значение

: в го переменные имеют диапазон действия, и вы можете переопределять переменные во вложенных блоках

: порою это может приводить к неожиданным последствиям

# Slide 6

\* Named return variables shadowing

.play examples/example2.go /begin shadowing/,/end shadowing/

: в го есть фича под названием named return parameters

: она позволяет инициализировать переменные и не указывать явно их при возврате из функции

: вместо этого рантайм автоматически подставит последнее присвоенное значение этой переменной

: однако они всегда создаются во внешнем поле действия функции

: и в случае если в вашем коде в одном из блоков переопределяется (затеняется) переменная

: то вы можете думать что работаете с то же переменной что была указанна в nsmed return parameters

: однако это будет совсем другая переменная что приведет к ошибке, которую довольно сложно отследить

# Slide 7

\* Multiple value shadowing

.play examples/example13.go /begin show/,/end show/

: и ситуация становится на много сложнее когда из функции возвращается множество переменных

: а так же с ростом размера функции становится все сложнее отслеживать какая переменная в каком

: скоупе работает и какое значение в конце концов будет возвращено

# это пример https://go-traps.appspot.com/#watchman-2

# Slide 8

\* Empty Returns

.code examples/example16.go /^func foo/,/^}/

.code examples/example16.go /^func bar/,/^}/

: кроме прочего код должен быть понятным с первого взгляда

: я считаю что на много понятнее, по какому сценарию мы пошли (успешному или критическому)

: если вы явно указываете значение по умолчанию для всех сущьностей таких как

: nil, 0, true, false, значения пустых структур итд

: это сделает ваш код более понятным в чтении

# Slide 9

\* Correct named returns usecase

.play examples/example8.go /begin show/,/end show/

: лично я вижу единственным допустимым применением именованных возвращаемых параметров

: в контексте оооооочень коротких функций, для написания документации

# Slide 10

\* Defer shadowing

.play examples/example3.go /^func main/,/^}/

: так же опастность может быть при обработке затененных переменных в defer болке

: переменная в которую установили значение ошибки и переменная которая проверяется в defer

: это две разные переменные и у нас будет некорректно обработана ошибка

# Slide 11

\* Http handler

.code examples/example14.go /begin show/,/end show/

: предположим мы решили сократить число кода и переместить обработку ошибки в имплементации http.Handler

: в блок derfer, и в этом случае у нас затенения не будет и все будет работать как ожидается

# Slide 12

\* Http handler shadowing

.image images/dif.png \_ 800

: и вот к нам на ревью приходит вот такой вот диф, мы в нем не видим ничего критичного

: мы людим простые, видим безобидный МР - ставим лайк

# Slide 13

\* Http handler

.code examples/example17.go /begin show/,/end show/

: а на самом деле наши переменные теперь затенены и теперь мы неправильно обрабатываем ошибки

: в этом и состоит основная опастность

: при изменении или расширении кода очень легко создать багу в неожиданном казалось бы месте

### defers ###

# Slide 14

\* defer

Defer is one of the most obvious features in go

.link https://about.sourcegraph.com/go/gophercon-2018-go-says-wat Go says what

: defer одна из самых неочевидных фич в нашем языке

: есть отличный доклад где большая часть whaaaat посвящена именно defer

: с ним стоит быть крайне осторожным

### Shadowing in parallelism ###

# Slide 15

\* Parallelism shadowing

.play examples/example4.go /^func main/,/^}/

: замыкания - это работа с переменными инициализированными вне анонимной фунции

: пологаю вы уже сталкивались с подобной особенностю, при работе с замыканиями в параллельно выполняющемся коде

: значение будет не то, которое имела переменная на этапе запуска горутины

: а то, которое имела переменная на момент выполнения кода внутри анонимной функции

# Slide 16

\* Parallelism shadowing

.play examples/example5.go /^func main/,/^}/

: на просторах интернета есть вот такое решение, применяется затенение переменной

: этот подход исправит проблему вызванную замыканием, однако выглядит это по крайней

# Slide 17

\* Avoid shadowing

- use all initialized variables only in the scope where they where created

- avoid using named return parameters

- always use explicit return values

- use defer very carefully

: или выбирайте разные имена для переменных, что бы избегать затенения

: или старайтесь что бы зоны действия переменных не пересекались

: всегда явно указывайте какую переменную вернуть и избегайте именованных возвращаемых параметров

: в общем стоит избегать затенения

: и используйте defer очень осторожно

### anonimous functions ###

# Slide 18

\* Parallelism shadowing

.play examples/example6.go /^func main/,/^}/

: более цивилизованным подходом является передача значения через параметры функции

: так мы избегаем и замыкания и затенения

# Slide 19

\* Anonymous functions

.play examples/example7.go /begin show/,/end show/

: однако на много правильнее будет вынести логику в отдельную именованную функцию

: вместо использования анонимной функции.

#: так вы гарантируете что никто не использует замыкание

#: и разгрузите логику, позволяя линейно читать ваш код.

#: а так же можно написать документацию и переиспользовать в будующем

# Slide 20

\* Anonymous functions

- the anonymous function gives a possibility to use closures

- anonymous functions are making code over complexed

- you can't create documentation for the anonymous function

- you can not reuse anonymous function

: так вы гарантируете что никто не использует замыкание

: и разгрузите логику, позволяя линейно читать ваш код.

: а так же можно написать документацию и переиспользовать в будующем

# Slide 21

\* Anonymous functions

.play examples/example15.go /begin show/,/end show/

: вот пример приблежонный к реальности что бы вы понимали, к чему приводит их применение

: замыкания, затенение переменных, именованные возвращаемые параметры, и пустые ретурны

: эта программа не читается линейно, так как надо постоянно подниматься наверх и смотреть что находится в анонимной функции

: затенения внутри функции делают проблематичным понять что и когда куда присваивается

: функция выполняет более одной обязанности

: и очени большая вероятность что при изменении этот код начнет сбоить

# Slide 22

\* Anonymous functions

.play examples/example18.go /begin show/,/end show/

: вот на столько проще выглядит код который просто не использует все выше упомянутые "фичи"

# Slide 23

\* Best practices

- don't use anonymous functions

- don't use closures

- if you need to path additional variables - use methods and structure fields

: в общем случае стоит избегать использования анонимных функций

: замыкания всегдя делают ваш код более сложным, усложняет рефакторинг

: следует использовать куда болеее гибкие механизмы такие как функции и методы

### panics ###

# Slide 24

\* Panic

- Panic is NOT an error

- panic in one handler can terminate execution of all program

- panic in some included code can lead to critical flow in all program

- no graceful shutdown in case of panic

- it is hard to debug code which panics

- it is hard to test the Panic

.link https://github.com/golang/go/wiki/CodeReviewComments#dont-panic Don't panic (Rob Pike)

: первое что надо знать о паниках - паники - это не ошибки

: паника прерывает выполнение программы и свидетельствует о критической неисправности

: паника в одном хэндлере или просто незначительном месте может привести к остановке работы всей программы

: при завершении работы через панику, исключена возможность мягкого завершения работы приложения

: их сложно дебажить так как recover зачастую происходит не в том же месте где паника

: панику проблематично тестировать

# Slide 25

\* Panic (what to expect)

- recover in the highest level of the goroutine (handler) just in case

- recover on the lowest level near the dangerous code

- always be careful with side libraries, read the documentation

- always read documentation when you use reflect, runtime or unsafe packages

: вы должны знать что некоторые разработчики все же используют паники

: поэтому в случае работы с вэбом имеет смысл использовать recover на уровне хэндлера

: это позволит продолжить работу не поврежденных хэндлеров

: так же если вам известно, что какие то функции могут паниковать, используйте recover сразу после их вызова

: переводя панику в ошибку с которой уже можно нормально работать

: всегда будь те осторожны и читайте документацию к пакетам которые вы используете и к пакетам таким как

: reflect, unsafe, runtime и.т.д.

### errors ###

# Slide 26

\* Custom errors

- custom codes

- stack trace

- logging purpose

- response purpose

: зачем могут понадобиться кастомные ошибки ?

: на пример в случае если вам надо оперировать кодами ошибок

: или добавить стек вызова функции где произошла ошибка

: вы можете захотеть разделить сообщения ошибки на те которые логируются

: и те, которые возвращаются клиенту

# Slide 27

\* Custom errors

.code examples/example9.go /begin custom error/,/end custom error/

: вот пример кастомной ошибки которая умеет дополнять сообщение в зависимости от кода

# Slide 28

\* Custom errors

.play examples/example11.go /begin show/,/end show/

: однако, как видим при попытке обернуть нашу кастомную ошибку с помощью пакета fmt

: наше сообщение выглядит не так, как мы хотели бы

# Slide 29

\* fmt.Errorf

fmt/print.go handleMethods()

.code examples/example19.go /begin show/,/end show/

: это происходит потому что fmt.Errorf() вызывает под копотом метод Error()

# Slide 30

\* Custom errors

.code examples/example10.go /begin interface/,/end interface/

.code examples/example10.go /begin custom error/,/end custom error/

: поэтому мы будем работать через наш враппер

# Slide 31

\* Custom errors

.play examples/example12.go /begin show/,/end show/

: и при таком подходе мы решили проблему с врапанием ошибок,

: теперь мы можем без препятствий добавлять контекст как нам удобно

# Slide 32

\* Stacktrace

.code examples/example21.go /begin err/,/end err/

: иногда вам может понадобиться вывести стек трейс для ошибки

: это можно сделать следующим образом

: прошу заметить что стек выводится отдельным методом, а не в имплементации Error()

: это важно, так как позволит вам легче в дальнейшем писать тесты, вы сможете проверять

: ошибки сравнивая их текст, что не возможно сделать если выводится стектрейс

# Slide 33

\* Stacktrace

.play examples/example21.go /begin main/,/end main/

: а вот так будет выглядеть применение ошибки умеющей выводить стек

: за счет того что мы имплементировали интерфейс StackError мы можем к нему скаститься

: при этом избежать приведения к конкретному типу

# Slide 34

\* Custom errors

.play examples/example20.go /begin show/,/end show/

: как видим, значение по умолчанию для указателя на нашу кастомную ошибку nil

: но при этом у нас проверка err != nil в main функции прошла

: это связанно с тем, что интерфейсный тим равен nil только когда у него нет ни динамического значения

: ни динамического типа, который мы, в нашем примере ошибочно установили

# Slide 35

\* Custom errors

- don't add stack trace to Error() implementation

- wrap custom errors with special function

- cast to interfaces, not concrete types

- remember that error is an interface type

: итак, не добавляйте стек к имплементации Error()

: врапайте кастомные ошибки своими методами

: приводите к интерфейсам, а не к конкретным типам

: и помните что работаете с интерфейсным типом, когда провреяете ошибки

## story

## idea

## summary

## questions