Go — простой и забавный язык. Но в нём, как и в любых других языках, есть свои подводные камни. И во многих из них сам Go не виноват. Одни — это естественное следствие прихода программистов из других языков, другие возникают из-за ложных представлений и нехватки подробностей. Если вы найдёте время и почитаете официальные спецификации, вики, почтовые рассылки, публикации в блогах и исходный код, то многие из подводных камней станут для вас очевидны. Но далеко не каждый так начинает, и это нормально. Если вы новичок в Go, статья поможет сэкономить немало часов, которые вы бы потратили на отладку кода. Мы будем рассматривать версии Go 1.5 и ниже.

Содержание

Уровень: абсолютный новичок  
  
1. [Открывающую фигурную скобку нельзя размещать в отдельной строке](https://habr.com/ru/company/vk/blog/314804/#1)  
2. [Неиспользуемые переменные](https://habr.com/ru/company/vk/blog/314804/#2)  
3. [Неиспользуемые импорты](https://habr.com/ru/company/vk/blog/314804/#3)  
4. [Короткие объявления переменных можно использовать только внутри функций](https://habr.com/ru/company/vk/blog/314804/#4)  
5. [Переобъявления переменных с помощью коротких объявлений](https://habr.com/ru/company/vk/blog/314804/#5)  
6. [Нельзя использовать короткие объявления переменных для настройки значений полей](https://habr.com/ru/company/vk/blog/314804/#6)  
7. [Случайное сокрытие переменных](https://habr.com/ru/company/vk/blog/314804/#7)  
8. [Нельзя использовать nil для инициализации переменной без явного указания типа](https://habr.com/ru/company/vk/blog/314804/#8)  
9. [Использование nil-слайсов (slice) и хеш-таблиц (map)](https://habr.com/ru/company/vk/blog/314804/#9)  
10. [Ёмкость хеш-таблиц](https://habr.com/ru/company/vk/blog/314804/#10)  
11. [Строки не могут быть nil](https://habr.com/ru/company/vk/blog/314804/#11)  
12. [Передача массивов в функции](https://habr.com/ru/company/vk/blog/314804/#12)  
13. [Неожиданные значения в выражениях range в слайсах и массивах](https://habr.com/ru/company/vk/blog/314804/#13)  
14. [Одномерность слайсов и массивов](https://habr.com/ru/company/vk/blog/314804/#14)  
15. [Обращение к несуществующим ключам в map](https://habr.com/ru/company/vk/blog/314804/#15)  
16. [Неизменяемость строк](https://habr.com/ru/company/vk/blog/314804/#16)  
17. [Преобразование строк в байт-слайсы (Byte Slices), и наоборот](https://habr.com/ru/company/vk/blog/314804/#17)  
18. [Строки и оператор индекса](https://habr.com/ru/company/vk/blog/314804/#18)  
19. [Строки — не всегда текст в кодировке UTF-8](https://habr.com/ru/company/vk/blog/314804/#19)  
20. [Длина строк](https://habr.com/ru/company/vk/blog/314804/#20)  
21. [Отсутствующая запятая в многострочных литералах slice/array/map](https://habr.com/ru/company/vk/blog/314804/#21)  
22. [log.Fatal и log.Panic не только журналируют](https://habr.com/ru/company/vk/blog/314804/#22)  
23. [Несинхронизированные операции встроенных структур данных](https://habr.com/ru/company/vk/blog/314804/#23)  
24. [Итерационные значения для строк в выражениях range](https://habr.com/ru/company/vk/blog/314804/#24)  
25. [Итерирование хеш-таблиц (map) с помощью выражения for range](https://habr.com/ru/company/vk/blog/314804/#25)  
26. [Сбойное поведение в выражениях switch](https://habr.com/ru/company/vk/blog/314804/#26)  
27. [Инкременты и декременты](https://habr.com/ru/company/vk/blog/314804/#27)  
28. [Побитовый NOT-оператор](https://habr.com/ru/company/vk/blog/314804/#28)  
29. [Различия приоритетов операторов](https://habr.com/ru/company/vk/blog/314804/#29)  
30. [Неэкспортированные поля структур не кодируются](https://habr.com/ru/company/vk/blog/314804/#30)  
31. [Выход из приложений с помощью активных горутин](https://habr.com/ru/company/vk/blog/314804/#31)  
32. [При отправке в небуферизованный канал данные возвращаются по мере готовности получателя](https://habr.com/ru/company/vk/blog/314804/#32)  
33. [Отправка в закрытый канал приводит к panic](https://habr.com/ru/company/vk/blog/314804/#33)  
34. [Использование «nil»-каналов](https://habr.com/ru/company/vk/blog/314804/#34)  
35. [Методы, принимающие параметры по значению, не меняют исходных значений](https://habr.com/ru/company/vk/blog/314804/#35)  
  
Уровень: более опытный новичок  
  
36. [Закрытие тела HTTP-ответа](https://habr.com/ru/company/vk/blog/314804/#36)  
37. [Закрытие HTTP-соединений](https://habr.com/ru/company/vk/blog/314804/#37)  
38. [Десериализация (unmarshalling) JSON-чисел в интерфейсные значения](https://habr.com/ru/company/vk/blog/314804/#38)  
39. [Сравнение struct, array, slice и map](https://habr.com/ru/company/vk/blog/314804/#39)  
40. [Восстановление после panic](https://habr.com/ru/company/vk/blog/314804/#40)  
41. [Обновление и привязка значений полей в slice, array и map в выражениях for range](https://habr.com/ru/company/vk/blog/314804/#41)  
42. [«Скрытые данные» в слайсах](https://habr.com/ru/company/vk/blog/314804/#42)  
43. [«Повреждение» данных в слайсах](https://habr.com/ru/company/vk/blog/314804/#43)  
44. [«Устаревшие» слайсы](https://habr.com/ru/company/vk/blog/314804/#44)  
45. [Методы и объявления типов](https://habr.com/ru/company/vk/blog/314804/#45)  
46. [Как выбраться из кодовых блоков for switch и for select](https://habr.com/ru/company/vk/blog/314804/#46)  
47. [Итерационные переменные и замыкания в выражениях for](https://habr.com/ru/company/vk/blog/314804/#47)  
48. [Вычисление аргумента блока defer (Deferred Function Call Argument Evaluation)](https://habr.com/ru/company/vk/blog/314804/#48)  
49. [Вызов блока defer](https://habr.com/ru/company/vk/blog/314804/#49)  
50. [Ошибки при приведении типов](https://habr.com/ru/company/vk/blog/314804/#50)  
51. [Блокированные горутины и утечки ресурсов](https://habr.com/ru/company/vk/blog/314804/#51)  
  
Уровень: продвинутый новичок  
  
52. [Применение методов, принимающих значение по ссылке (pointer receiver), к экземплярам значений](https://habr.com/ru/company/vk/blog/314804/#52)  
53. [Обновление полей значений в хеш-таблице](https://habr.com/ru/company/vk/blog/314804/#53)  
54. [nil-интерфейсы и nil-интерфейсные значения](https://habr.com/ru/company/vk/blog/314804/#54)  
55. [Переменные стека и кучи](https://habr.com/ru/company/vk/blog/314804/#55)  
56. [GOMAXPROCS, согласованность (concurrency) и параллелизм](https://habr.com/ru/company/vk/blog/314804/#56)  
57. [Изменение порядка операций чтения и записи](https://habr.com/ru/company/vk/blog/314804/#57)  
58. [Диспетчеризация по приоритетам (Preemptive Scheduling)](https://habr.com/ru/company/vk/blog/314804/#58)

1. Открывающую фигурную скобку нельзя размещать в отдельной строке

В большинстве других языков, использующих фигурные скобки, вам нужно выбирать, где их размещать. Go выбивается из правила. За это вы можете благодарить автоматическую вставку точки с запятой (точка с запятой предполагается в конце каждой строки, без анализа следующей). Да, в Go есть точка с запятой!  
  
Неправильно:

**package** main

**import** "fmt"

**func** **main**()

{ // ошибка, нельзя выносить открывающую фигурную скобку в отдельную строку

fmt.Println("hello there!")

}

Ошибка компилирования:

/tmp/sandbox826898458/main.go:6: syntax error: unexpected semicolon or newline before {

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

fmt.Println("works!")

}

2. Неиспользуемые переменные

Если у вас есть неиспользуемые переменные, то код не скомпилируется. Исключение: переменные, которые объявляются внутри функций. Это правило не касается глобальных переменных. Также можно иметь неиспользуемые аргументы функций.  
  
Если вы присвоили неиспользуемой переменной новое значение, то ваш код всё равно не будет компилироваться. Придётся её где-то использовать, чтобы угодить компилятору.  
  
Неправильно:

**package** main

**var** gvar **int** // not an error

**func** **main**() {

**var** one **int** // ошибка, неиспользуемая переменная

two := 2 // ошибка, неиспользуемая переменная

**var** three **int** // ошибка, даже несмотря на присваивание значения 3 в следующей строке

three = 3

**func**(unused **string**) {

fmt.Println("Unused arg. No compile error")

}("what?")

}

Ошибки компилирования:

/tmp/sandbox473116179/main.go:6: one declared and not used /tmp/sandbox473116179/main.go:7: two declared and not used /tmp/sandbox473116179/main.go:8: three declared and not used

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

**var** one **int**

\_ = one

two := 2

fmt.Println(two)

**var** three **int**

three = 3

one = three

**var** four **int**

four = four

}

Другое решение: комментировать или удалять неиспользуемые переменные.

3. Неиспользуемые импорты

Если вы импортируете пакет и потом не используете какие-либо из его функций, интерфейсов, структур или переменных, то код не скомпилируется. Если нужно импортировать пакет, идентификатор «\_» в качестве его имени поможет избежать ошибок компилирования. Идентификатор «\_» чаще всего применяется для использования сайд-эффектов импортированных библиотек.  
  
Неправильно:

**package** main

**import** (

"fmt"

"log"

"time"

)

**func** **main**() {

}

Ошибки компилирования:

/tmp/sandbox627475386/main.go:4: imported and not used: "fmt" /tmp/sandbox627475386/main.go:5: imported and not used: "log" /tmp/sandbox627475386/main.go:6: imported and not used: "time"

Правильно:

**package** main

**import** (

\_ "fmt"

"log"

"time"

)

**var** \_ = log.Println

**func** **main**() {

\_ = time.Now

}

Другое решение: удалить или закомментировать неиспользуемые импорты. В этом поможет инструмент [goimports](http://godoc.org/golang.org/x/tools/cmd/goimports).

4. Короткие объявления переменных можно использовать только внутри функций

Неправильно:

**package** main

myvar := 1 // ошибка

**func** **main**() {

}

Ошибка компилирования:

/tmp/sandbox265716165/main.go:3: non-declaration statement outside **function** body

Правильно:

**package** main

**var** myvar = 1

**func** **main**() {

}

5. Переобъявления переменных с помощью коротких объявлений

В одной области видимости выражения нельзя переобъявлять переменные, но это можно делать в объявлении нескольких переменных (multi-variable declarations), среди которых хотя бы одна — новая. Переобъявляемые переменные должны располагаться в том же блоке, иначе получится скрытая переменная (shadowed variable).  
Неправильно:

**package** main

**func** **main**() {

one := 0

one := 1 // ошибка

}

Ошибка компилирования:

/tmp/sandbox706333626/main.go:5: no new variables on left side of :=

Правильно:

**package** main

**func** **main**() {

one := 0

one, two := 1,2

one,two = two,one

}

6. Нельзя использовать короткие объявления переменных для настройки значений полей

Неправильно:

**package** main

**import** (

"fmt"

)

**type** info **struct** {

result **int**

}

**func** **work**() (**int**,error) {

**return** 13,nil

}

**func** **main**() {

**var** data info

data.result, err := work() // ошибка

fmt.Printf("info: %+v\n",data)

}

Ошибка компилирования:

prog.go:18: non-name data.result on left side of :=

Хотя разработчикам Go уже предлагали это исправить, не стоит надеяться на перемены: Робу Пайку нравится всё «как есть». Вам помогут временные переменные. Или предварительно объявляйте все свои переменные и используйте стандартный оператор присваивания.  
  
Правильно:

**package** main

**import** (

"fmt"

)

**type** info **struct** {

result **int**

}

**func** **work**() (**int**,error) {

**return** 13,nil

}

**func** **main**() {

**var** data info

**var** err error

data.result, err = work() // ok

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Printf("info: %+v\n",data) // выводит: info: {result:13}

}

7. Случайное сокрытие переменных

Синтаксис короткого объявления переменных так удобен (особенно для тех, кто пришёл в Go из динамических языков), что его легко принять за регулярную операцию присваивания. Если вы сделаете эту ошибку в новом блоке кода, то компилятор не выдаст ошибку, но приложение будет работать некорректно.

**package** main

**import** "fmt"

**func** **main**() {

x := 1

fmt.Println(x) // выводит 1

{

fmt.Println(x) // выводит 1

x := 2

fmt.Println(x) // выводит 2

}

fmt.Println(x) // выводит 1 (плохо, если нужно было 2)

}

Это очень распространённая ошибка даже среди опытных Go-разработчиков. Её легко совершить и трудно заметить. Для выявления подобных ситуаций можно использовать команду vet. По умолчанию она не выполняет проверку переменных на скрытость. Поэтому используйте флаг -shadow: go tool vet -shadow your\_file.go

8. Нельзя использовать nil для инициализации переменной без явного указания типа

Идентификатор nil можно использовать как «нулевое значение» (zero value) для интерфейсов, функций, указателей, хеш-таблиц (map), слайсов (slices) и каналов. Если не задать тип переменной, то компилятор не сможет завершить работу, потому что не сумеет угадать тип.  
  
Неправильно:

**package** main

**func** **main**() {

**var** x = nil // ошибка

\_ = x

}

Ошибка компилирования:

/tmp/sandbox188239583/main.go:4: use of untyped nil

Правильно:

**package** main

**func** **main**() {

**var** x **interface**{} = nil

\_ = x

}

9. Использование nil-слайсов (slice) и хеш-таблиц (map)

Можно добавлять элементы в nil-слайс, но если то же самое сделать с хеш-таблицей, то это приведёт к runtime panic.  
  
Правильно:

**package** main

**func** **main**() {

**var** s []**int**

s = append(s,1)

}

Неправильно:

**package** main

**func** **main**() {

**var** m **map**[**string**]**int**

m["one"] = 1 // ошибка

}

10. Ёмкость хеш-таблиц

Можно устанавливать ёмкость при создании хеш-таблиц, но нельзя применять к ним функцию cap().  
  
Неправильно:

**package** main

**func** **main**() {

m := make(**map**[**string**]**int**,99)

cap(m) // ошибка

}

Ошибка компилирования:

/tmp/sandbox326543983/main.go:5: invalid argument m (type map[string]int) **for** cap

11. Строки не могут быть nil

Это подводный камень для начинающих, которые присваивают строковым переменным nil-идентификаторы.  
  
Неправильно:

**package** main

**func** **main**() {

**var** x **string** = nil // ошибка

**if** x == nil { // ошибка

x = "default"

}

}

Ошибки компилирования:

/tmp/sandbox630560459/main.go:4: cannot use nil as type string **in** assignment /tmp/sandbox630560459/main.go:6: invalid operation: x == nil (mismatched types string and nil)

Правильно:  
  
package main

**func** **main**() {

**var** x **string** // возвращает значение по умолчанию "" (нулевое значение)

**if** x == "" {

x = "default"

}

}

12. Передача массивов в функции

Если вы разрабатываете на С/С++, то массивы для вас — указатели. Когда вы передаёте массивы функциям, функции ссылаются на ту же область памяти и поэтому могут обновлять исходные данные. В Go массивы являются значениями, так что, когда мы передаём их функциям, те получают копию исходного массива. Это может стать проблемой, если вы пытаетесь обновлять данные в массиве.

**package** main

**import** "fmt"

**func** **main**() {

x := [3]**int**{1,2,3}

**func**(arr [3]**int**) {

arr[0] = 7

fmt.Println(arr) // выводит [7 2 3]

}(x)

fmt.Println(x) // выводит [1 2 3] (плохо, если вам нужно было [7 2 3])

}

Если нужно обновить исходные данные в массиве, используйте типы указателей массива.

**package** main

**import** "fmt"

**func** **main**() {

x := [3]**int**{1,2,3}

**func**(arr \*[3]**int**) {

(\*arr)[0] = 7

fmt.Println(arr) // выводит &[7 2 3]

}(&x)

fmt.Println(x) // выводит [7 2 3]

}

Другое решение: слайсы. Хотя ваша функция получает копию переменной слайса, та всё ещё является ссылкой на исходные данные.

**package** main

**import** "fmt"

**func** **main**() {

x := []**int**{1,2,3}

**func**(arr []**int**) {

arr[0] = 7

fmt.Println(arr) // выводит [7 2 3]

}(x)

fmt.Println(x) // выводит [7 2 3]

}

13. Неожиданные значения в выражениях range в слайсах и массивах

Это может произойти, если вы привыкли к выражениям for-in или foreach в других языках. Но в Go выражение range отличается тем, что оно генерирует два значения: первое — это индекс элемента (item index), а второе — данные элемента (item data).  
  
Неправильно:

**package** main

**import** "fmt"

**func** **main**() {

x := []**string**{"a","b","c"}

**for** v := **range** x {

fmt.Println(v) // выводит 0, 1, 2

}

}

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

x := []**string**{"a","b","c"}

**for** \_, v := **range** x {

fmt.Println(v) // выводит a, b, c

}

}

14. Одномерность слайсов и массивов

Кажется, что Go поддерживает многомерные массивы и слайсы? Нет, это не так. Хотя можно создавать массивы из массивов и слайсы из слайсов. С точки зрения производительности и сложности — далеко не идеальное решение для приложений, которые выполняют числовые вычисления и основаны на динамических многомерных массивах.  
  
Можно создавать динамические многомерные массивы с помощью обычных одномерных массивов, слайсов из «независимых» слайсов, а также слайсов из слайсов «с совместно используемыми данными».  
  
Если вы используете обычные одномерные массивы, то при их росте вы отвечаете за индексирование, проверку границ и перераспределение памяти.  
  
Процесс создания динамического многомерного массива с помощью слайсов из «независимых» слайсов состоит из двух шагов. Сначала нужно создать внешний слайс, а затем разместить в памяти все внутренние слайсы. Внутренние слайсы не зависят друг от друга. Их можно увеличивать и уменьшать, не затрагивая другие.

**package** main

**func** **main**() {

x := 2

y := 4

table := make([][]**int**,x)

**for** i:= **range** table {

table[i] = make([]**int**,y)

}

}

Создание динамического многомерного массива с помощью слайсов из слайсов «с совместно используемыми данными» состоит из трёх шагов. Сначала нужно создать слайс, выполняющий роль «контейнера» данных, он содержит исходные данные (raw data). Затем — внешний слайс. В конце мы инициализируем каждый из внутренних слайсов, перенарезая слайс с исходными данными.

**package** main

**import** "fmt"

**func** **main**() {

h, w := 2, 4

raw := make([]**int**,h\*w)

**for** i := **range** raw {

raw[i] = i

}

fmt.Println(raw,&raw[4])

// выводит: [0 1 2 3 4 5 6 7] <ptr\_addr\_x>

table := make([][]**int**,h)

**for** i:= **range** table {

table[i] = raw[i\*w:i\*w + w]

}

fmt.Println(table,&table[1][0])

// выводит: [[0 1 2 3] [4 5 6 7]] <ptr\_addr\_x>

}

Предлагается разработать спецификацию на многомерные массивы и слайсы, но сейчас, судя по всему, у этой задачи низкий приоритет.

15. Обращение к несуществующим ключам в map

Эту ошибку совершают разработчики, которые при обращении к несуществующему ключу ожидают получить nil-значение (как это происходит в некоторых языках). Возвращаемое значение будет nil, если «нулевое значение» для соответствующего типа данных — nil. Но для других типов возвращаемое значение окажется другим. Определять, существует ли запись в хеш-таблице (map record), можно с помощью проверки на правильное «нулевое значение». Но это не всегда надёжно (например, что вы будете делать, если у вас есть таблица булевых значений, где «нулевое значение» — false). Самый надёжный способ узнать, существует ли запись, — проверить второе значение, возвращаемое операцией доступа к таблице.  
  
Плохо:

**package** main

**import** "fmt"

**func** **main**() {

x := **map**[**string**]**string**{"one":"a","two":"","three":"c"}

**if** v := x["two"]; v == "" { // некорректно

fmt.Println("no entry")

}

}

Хорошо:

**package** main

**import** "fmt"

**func** **main**() {

x := **map**[**string**]**string**{"one":"a","two":"","three":"c"}

**if** \_,ok := x["two"]; !ok {

fmt.Println("no entry")

}

}

16. Неизменяемость строк

Если вы попытаетесь обновить отдельные символы строковой переменной с помощь оператора индекса, то это не сработает. Строки — это байт-слайсы (byte slices), доступные только для чтения. Если вам все-таки нужно обновить строку, то стоит использовать байт-слайс и преобразовывать его в строку по необходимости.  
  
Неправильно:

**package** main

**import** "fmt"

**func** **main**() {

x := "text"

x[0] = 'T'

fmt.Println(x)

}

Ошибка компилирования:

/tmp/sandbox305565531/main.go:7: cannot assign to x[0]

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

x := "text"

xbytes := []**byte**(x)

xbytes[0] = 'T'

fmt.Println(**string**(xbytes)) // выводит Text

}

Стоит заметить, что это неправильный способ обновления символов в текстовой строке, потому что символ может состоять из нескольких байтов. В этом случае лучше конвертировать строку в слайс из «рун» (rune). Но даже внутри слайсов из «рун» одиночный символ может быть разбит на несколько рун, например если есть символ апострофа (grave accent). Такая непростая и запутанная природа «символов» является причиной того, что в Go строковые значения представляют собой последовательностей байтов.

17. Преобразование строк в байт-слайсы (Byte Slices), и наоборот

Когда вы преобразуете строку в байт-слайс (и наоборот), вы получаете полную копию исходных данных. Это не операция приведения (cast operation), как в других языках, и не перенарезка (reslicing), когда переменная нового слайса указывает на один и тот же массив, занятый исходным байт-слайсом.  
  
В Go есть несколько оптимизаций для преобразований из []byte в string и из string в []byte, позволяющих избегать дополнительных выделений памяти (ещё больше оптимизаций в списке todo).  
  
Первая оптимизация позволяет избежать дополнительного выделения памяти, когда ключи []byte используются для поиска записей в коллекциях map[string]: m[string(key)].  
  
Вторая оптимизация позволяет избегать дополнительного выделения в выражениях for range, когда строки преобразуются в []byte: for i,v := range []byte(str) {...}.

18. Строки и оператор индекса

Оператор индекса, применяемый к **строке**, возвращает байтовое значение (byte value), а не символ (как в других языках).

**package** main

**import** "fmt"

**func** **main**() {

x := "text"

fmt.Println(x[0]) // выводит 116

fmt.Printf("%T",x[0]) // выводит uint8

}

Если нужно обратиться к конкретным «символам» (кодовым точкам/рунам Unicode), то используйте выражение for range. Также вам будут полезны официальный пакет unicode/utf8 и экспериментальный utf8string (golang.org/x/exp/utf8string). utf8string включает в себя удобный метод At(). Можно также преобразовать **строку** в слайс рун (slice of runes).

19. Строки — не всегда текст в кодировке UTF-8

Строковые значения необязательно должны быть представлены в виде текста в кодировке UTF-8. Здесь возможен произвольный набор байтов. Единственный случай, когда строки должны быть в кодировке UTF-8, — когда они используются как строковые литералы. Но даже они могут включать в себя данные с экранированными последовательностями.  
  
Чтобы узнать кодировку строки, используйте функцию ValidString() из пакета unicode/utf8.

**package** main

**import** (

"fmt"

"unicode/utf8"

)

**func** **main**() {

data1 := "ABC"

fmt.Println(utf8.ValidString(data1)) // выводит: true

data2 := "A\xfeC"

fmt.Println(utf8.ValidString(data2)) // выводит: false

}

20. Длина строк

Допустим, вы разрабатываете на Python и у вас есть такой код:

data = u'♥'

print(len(data)) # выводит: 1

Если преобразовать его в аналогичный код на Go, то результат может вас удивить.

**package** main

**import** "fmt"

**func** **main**() {

data := "♥"

fmt.Println(len(data)) // выводит: 3

}

Встроенная функция len() возвращает не символ, а количество байт, как это происходит с Unicode-строками в Python.  
  
Чтобы получить такой же результат в Go, используйте функцию RuneCountInString() из пакета unicode/utf8.

**package** main

**import** (

"fmt"

"unicode/utf8"

)

**func** **main**() {

data := "♥"

fmt.Println(utf8.RuneCountInString(data)) // выводит: 1

Технически функция RuneCountInString() не возвращает количество символов, потому что один символ может занимать несколько рун.

**package** main

**import** (

"fmt"

"unicode/utf8"

)

**func** **main**() {

data := "é"

fmt.Println(len(data)) // выводит: 3

fmt.Println(utf8.RuneCountInString(data)) // выводит: 2

}

21. Отсутствующая запятая в многострочных литералах slice/array/map

Неправильно:

**package** main

**func** **main**() {

x := []**int**{

1,

2 // error

}

\_ = x

}

Ошибки компилирования:

/tmp/sandbox367520156/main.go:6: syntax error: need trailing comma before newline **in** composite literal /tmp/sandbox367520156/main.go:8: non-declaration statement outside **function** body /tmp/sandbox367520156/main.go:9: syntax error: unexpected }

Правильно:

**package** main

**func** **main**() {

x := []**int**{

1,

2,

}

x = x

y := []**int**{3,4,} // ошибки нет

y = y

}

Вы не получите ошибку компилирования, если оставите замыкающую запятую при объявлении в одну строчку.

22. log.Fatal и log.Panic не только журналируют

Библиотеки для логирования часто обеспечивают различные уровни для сообщений. В отличие от других языков, пакет логирования в Go делает больше. Если вызвать его функции Fatal\*() и Panic\*(), то приложение будет закрыто.

**package** main

**import** "log"

**func** **main**() {

log.Fatalln("Fatal Level: log entry") // здесь выполняется выход из приложения

log.Println("Normal Level: log entry")

}

23. Несинхронизированные операции встроенных структур данных

Некоторые возможности Go нативно поддерживают многозадачность (concurrency), но в их число не входят потокобезопасные коллекции (concurrency safe). Вы сами отвечаете за атомарность обновления коллекций. Для реализации атомарных операций рекомендуется использовать горутины и каналы, но можно задействовать и пакет sync, если это целесообразно для вашего приложения.

24. Итерационные значения для строк в выражениях range

Значение индекса (первое значение, возвращаемое операцией range) — это индекс первого байта текущего «символа» (кодовая точка/руна Unicode), возвращённый во втором значении. Это не индекс текущего «символа», как в других языках. Обратите внимание, что настоящий символ может быть представлен несколькими рунами. Если вам нужно работать именно с символами, то стоит использовать пакет norm (golang.org/x/text/unicode/norm).  
  
Выражения for range со строковыми переменными пытаются интерпретировать данные как текст в кодировке UTF-8. Если они не распознают какую-либо последовательность байтов, то возвращают руны 0xfffd (символы замены Unicode), а не реальные данные. Если в вашей строке хранятся произвольные данные (не UTF-8), то для сохранения преобразуйте их в байт-слайсы.

**package** main

**import** "fmt"

**func** **main**() {

data := "A\xfe\x02\xff\x04"

**for** \_,v := **range** data {

fmt.Printf("%#x ",v)

}

// выводит: 0x41 0xfffd 0x2 0xfffd 0x4 (нехорошо)

fmt.Println()

**for** \_,v := **range** []**byte**(data) {

fmt.Printf("%#x ",v)

}

// выводит: 0x41 0xfe 0x2 0xff 0x4 (хорошо)

}

25. Итерирование хеш-таблиц (map) с помощью выражения for range

На этот подводный камень натыкаются те, кто ожидают, что элементы будут располагаться в определённом порядке (например, отсортированные по значению ключа). Каждая итерация хеш-таблицы приводит к разным результатам. Среда исполнения (runtime) Go пытается сделать всё возможное, рандомизируя порядок итерирования, но ей это не всегда удаётся, поэтому вы можете получить несколько одинаковых итераций (например, пять) подряд.

**package** main

**import** "fmt"

**func** **main**() {

m := **map**[**string**]**int**{"one":1,"two":2,"three":3,"four":4}

**for** k,v := **range** m {

fmt.Println(k,v)

}

}

А если вы используете Go Playground (https://play.golang.org/), то всегда будете получать одинаковые результаты, потому что код не перекомпилируется, пока вы его не измените.

26. Сбойное поведение в выражениях switch

Блоки case в выражениях switch по умолчанию прерываются (break). В других языках поведение по умолчанию другое: переход (fall through) к следующему блоку case.

**package** main

**import** "fmt"

**func** **main**() {

isSpace := **func**(ch **byte**) **bool** {

**switch**(ch) {

**case** ' ': // ошибка

**case** '\t':

**return** true

}

**return** false

}

fmt.Println(isSpace('\t')) // выводит true (хорошо)

fmt.Println(isSpace(' ')) // выводит false (плохо)

}

Можно заставить блоки case переходить принудительно с помощью выражения fallthrough в конце каждого блока. Можно также переписать ваше выражение switch, чтобы в блоках использовались списки выражений.

**package** main

**import** "fmt"

**func** **main**() {

isSpace := **func**(ch **byte**) **bool** {

**switch**(ch) {

**case** ' ', '\t':

**return** true

}

**return** false

}

fmt.Println(isSpace('\t')) // выводит true (хорошо)

fmt.Println(isSpace(' ')) // выводит true (хорошо)

}

27. Инкременты и декременты

Во многих языках есть операторы инкрементирования и декрементирования. Но в Go не поддерживаются их префиксные версии. Также нельзя в одном выражении использовать оба этих выражения.  
  
Неправильно:

**package** main

**import** "fmt"

**func** **main**() {

data := []**int**{1,2,3}

i := 0

++i // error

fmt.Println(data[i++]) // ошибка

}

Ошибки компилирования:

/tmp/sandbox101231828/main.go:8: syntax error: unexpected ++ /tmp/sandbox101231828/main.go:9: syntax error: unexpected ++, expecting :

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

data := []**int**{1,2,3}

i := 0

i++

fmt.Println(data[i])

}

28. Побитовый NOT-оператор

Во многих языках символ ~ используется в качестве унарной NOT-операции (aka побитовое дополнение, bitwise complement), однако в Go для этого применяется XOR-оператор (^).  
  
Неправильно:

**package** main

**import** "fmt"

**func** **main**() {

fmt.Println(~2) // ошибка

}

Ошибка компилирования:

/tmp/sandbox965529189/main.go:6: the bitwise complement operator is ^

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

**var** d **uint8** = 2

fmt.Printf("%08b\n",^d)

}

Кого-то может запутать, что ^ в Go — это XOR-оператор. Если хотите, выражайте унарную NOT-операцию (например, NOT 0x02) с помощью бинарной XOR-операции (например, 0x02 XOR 0xff). Это объясняет, почему ^ используется для выражения унарной NOT-операции.  
  
Также в Go есть специальный побитовый оператор AND NOT (&^), который легко принять за оператор NOT. AND NOT выглядит как специальная функция/хак ради поддержки A AND (NOT B) без обязательного использования фигурных скобок.

**package** main

**import** "fmt"

**func** **main**() {

**var** a **uint8** = 0x82

**var** b **uint8** = 0x02

fmt.Printf("%08b [A]\n",a)

fmt.Printf("%08b [B]\n",b)

fmt.Printf("%08b (NOT B)\n",^b)

fmt.Printf("%08b ^ %08b = %08b [B XOR 0xff]\n",b,0xff,b ^ 0xff)

fmt.Printf("%08b ^ %08b = %08b [A XOR B]\n",a,b,a ^ b)

fmt.Printf("%08b & %08b = %08b [A AND B]\n",a,b,a & b)

fmt.Printf("%08b &^%08b = %08b [A 'AND NOT' B]\n",a,b,a &^ b)

fmt.Printf("%08b&(^%08b)= %08b [A AND (NOT B)]\n",a,b,a & (^b))

}

29. Различия приоритетов операторов

Помимо «довольно понятных» (bit clear) операторов (&^), в Go есть набор стандартных операторов, используемых многими другими языками. Но их приоритеты в данном случае не всегда такие же.

**package** main

**import** "fmt"

**func** **main**() {

fmt.Printf("0x2 & 0x2 + 0x4 -> %#x\n",0x2 & 0x2 + 0x4)

//prints: 0x2 & 0x2 + 0x4 -> 0x6

//Go: (0x2 & 0x2) + 0x4

//C++: 0x2 & (0x2 + 0x4) -> 0x2

fmt.Printf("0x2 + 0x2 << 0x1 -> %#x\n",0x2 + 0x2 << 0x1)

//prints: 0x2 + 0x2 << 0x1 -> 0x6

//Go: 0x2 + (0x2 << 0x1)

//C++: (0x2 + 0x2) << 0x1 -> 0x8

fmt.Printf("0xf | 0x2 ^ 0x2 -> %#x\n",0xf | 0x2 ^ 0x2)

//prints: 0xf | 0x2 ^ 0x2 -> 0xd

//Go: (0xf | 0x2) ^ 0x2

//C++: 0xf | (0x2 ^ 0x2) -> 0xf

}

30. Неэкспортированные поля структур не кодируются

Поля структур (struct fields), начинающиеся со строчных букв, не будут кодироваться (JSON, XML, GON и т. д.), так что при декодировании структуры вы получите в этих неэкспортированных полях нулевые значения.

**package** main

**import** (

"fmt"

"encoding/json"

)

**type** MyData **struct** {

One **int**

two **string**

}

**func** **main**() {

in := MyData{1,"two"}

fmt.Printf("%#v\n",in) // выводит main.MyData{One:1, two:"two"}

encoded,\_ := json.Marshal(in)

fmt.Println(**string**(encoded)) // выводит {"One":1}

**var** out MyData

json.Unmarshal(encoded,&out)

fmt.Printf("%#v\n",out) // выводит main.MyData{One:1, two:""}

}

31. Выход из приложений с помощью активных горутин

Приложение не будет ждать завершения ваших горутин. Новички часто об этом забывают. Все когда-то начинают — в таких ошибках нет ничего стыдного.

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

workerCount := 2

**for** i := 0; i < workerCount; i++ {

**go** doit(i)

}

time.Sleep(1 \* time.Second)

fmt.Println("all done!")

}

**func** **doit**(workerId **int**) {

fmt.Printf("[%v] is running\n",workerId)

time.Sleep(3 \* time.Second)

fmt.Printf("[%v] is done\n",workerId)

}

Вы увидите:

[0] is running

[1] is running

all done!

Одно из самых популярных решений — переменная WaitGroup. Это позволит главной горутине ожидать завершения работы всех рабочих горутин. Если ваше приложение использует долго выполняемые рабочие горутины с циклами обработки сообщений, то вам понадобится как-то сигнализировать им о том, что пора выходить. Можно отправлять каждой такой горутине сообщение kill. Или закрывать каналы, из которых рабочие горутины получают данные: это простой способ сигнализировать оптом.

**package** main

**import** (

"fmt"

"sync"

)

**func** **main**() {

**var** wg sync.WaitGroup

done := make(**chan** **struct**{})

workerCount := 2

**for** i := 0; i < workerCount; i++ {

wg.Add(1)

**go** doit(i,done,wg)

}

close(done)

wg.Wait()

fmt.Println("all done!")

}

**func** **doit**(workerId **int**,done <-**chan** **struct**{},wg sync.WaitGroup) {

fmt.Printf("[%v] is running\n",workerId)

**defer** wg.Done()

<- done

fmt.Printf("[%v] is done\n",workerId)

}

Если запустить это приложение, вы увидите:

[0] is running

[0] is done

[1] is running

[1] is done

Похоже, все горутины закончили работать до выхода главной горутины. Замечательно! Однако вы увидите и это:

fatal error: all goroutines are asleep - deadlock!

Нехорошо! Что происходит? Откуда взялась взаимоблокировка? Ведь все вышли и выполнили wg.Done(). Приложение должно работать.  
  
Блокировка возникает, потому что каждый рабочий получает копию исходной переменной WaitGroup. И когда все они выполняют wg.Done(), это никак не влияет на переменную WaitGroup в главной горутине.

**package** main

**import** (

"fmt"

"sync"

)

**func** **main**() {

**var** wg sync.WaitGroup

done := make(**chan** **struct**{})

wq := make(**chan** **interface**{})

workerCount := 2

**for** i := 0; i < workerCount; i++ {

wg.Add(1)

**go** doit(i,wq,done,&wg)

}

**for** i := 0; i < workerCount; i++ {

wq <- i

}

close(done)

wg.Wait()

fmt.Println("all done!")

}

**func** **doit**(workerId **int**, wq <-**chan** **interface**{},done <-**chan** **struct**{},wg \*sync.WaitGroup) {

fmt.Printf("[%v] is running\n",workerId)

**defer** wg.Done()

**for** {

**select** {

**case** m := <- wq:

fmt.Printf("[%v] m => %v\n",workerId,m)

**case** <- done:

fmt.Printf("[%v] is done\n",workerId)

**return**

}

}

}

Теперь всё работает правильно.

32. При отправке в небуферизованный канал данные возвращаются по мере готовности получателя

Отправитель не будет заблокирован, пока получатель обрабатывает ваше сообщение. В зависимости от машины, на которой выполняется код, получающая горутина может и не иметь достаточно времени на обработку сообщения, прежде чем продолжится выполнение отправителя.

**package** main

**import** "fmt"

**func** **main**() {

ch := make(**chan** **string**)

**go** **func**() {

**for** m := **range** ch {

fmt.Println("processed:",m)

}

}()

ch <- "cmd.1"

ch <- "cmd.2" // не будет обработано

}

33. Отправка в закрытый канал приводит к panic

Получение из закрытого канала безопасно. Возвращаемое значение ok в получаемом выражении (receive statement) станет false, что говорит о том, что никакие данные не были получены. Если вы получаете из буферизованного канала, то получите сначала буферизованные данные, а когда они закончатся, выражение ok станет false.  
  
Отправка данных в закрытый канал приводит к panic. Это задокументированное поведение, но оно не всегда интуитивно ожидаемо разработчиками, которые могут считать, что поведение при отправке будет аналогично поведению при приёме.

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

ch := make(**chan** **int**)

**for** i := 0; i < 3; i++ {

**go** **func**(idx **int**) {

ch <- (idx + 1) \* 2

}(i)

}

// get the first result

fmt.Println(<-ch)

close(ch) //нехорошо (у вас всё ещё есть другие отправители)

// do other work

time.Sleep(2 \* time.Second)

}

Решение зависит от вашего приложения. Это может быть небольшое изменение кода — или архитектуры, если потребуется. В любом случае удостоверьтесь, что приложение не пытается отправлять данные в закрытый канал.  
  
Пример с багом можно исправить, сигнализируя через специальный канал отмены (special cancellation channel) остальным рабочим горутинам, что их результаты больше не нужны.

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

ch := make(**chan** **int**)

done := make(**chan** **struct**{})

**for** i := 0; i < 3; i++ {

**go** **func**(idx **int**) {

**select** {

**case** ch <- (idx + 1) \* 2: fmt.Println(idx,"sent result")

**case** <- done: fmt.Println(idx,"exiting")

}

}(i)

}

// get first result

fmt.Println("result:",<-ch)

close(done)

// do other work

time.Sleep(3 \* time.Second)

}

34. Использование «nil»-каналов

В канале nil операции отправки и приёма блокируются навсегда. Это хорошо задокументированное поведение, но оно может стать сюрпризом для новичков.

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

**var** ch **chan** **int**

**for** i := 0; i < 3; i++ {

**go** **func**(idx **int**) {

ch <- (idx + 1) \* 2

}(i)

}

// get first result

fmt.Println("result:",<-ch)

// do other work

time.Sleep(2 \* time.Second)

}

При выполнении этого кода вы увидите ошибку runtime наподобие fatal error: all goroutines are asleep - deadlock!  
  
Это поведение можно использовать для динамического включения и отключения блоков case в выражении select.

**package** main

**import** "fmt"

**import** "time"

**func** **main**() {

inch := make(**chan** **int**)

outch := make(**chan** **int**)

**go** **func**() {

**var** in <- **chan** **int** = inch

**var** out **chan** <- **int**

**var** val **int**

**for** {

**select** {

**case** out <- val:

out = nil

in = inch

**case** val = <- in:

out = outch

in = nil

}

}

}()

**go** **func**() {

**for** r := **range** outch {

fmt.Println("result:",r)

}

}()

time.Sleep(0)

inch <- 1

inch <- 2

time.Sleep(3 \* time.Second)

}

35. Методы, принимающие параметры по значению, не меняют исходных значений

Параметры методов — это как обычные аргументы функций. Если они объявляются значением, то функция/метод получает копию вашего аргумента (receiver argument). Изменения в принятом значении не повлияют на исходное значение, если значение — переменная хеш-таблицы (map) или слайса и вы обновляете элементы коллекции или если обновляемые поля в значении — это указатели.

**package** main

**import** "fmt"

**type** data **struct** {

num **int**

key \***string**

items **map**[**string**]**bool**

}

**func** (this \*data) **pmethod**() {

this.num = 7

}

**func** (this data) **vmethod**() {

this.num = 8

\*this.key = "v.key"

this.items["vmethod"] = true

}

**func** **main**() {

key := "key.1"

d := data{1,&key,make(**map**[**string**]**bool**)}

fmt.Printf("num=%v key=%v items=%v\n",d.num,\*d.key,d.items)

// prints num=1 key=key.1 items=map[]

d.pmethod()

fmt.Printf("num=%v key=%v items=%v\n",d.num,\*d.key,d.items)

// prints num=7 key=key.1 items=map[]

d.vmethod()

fmt.Printf("num=%v key=%v items=%v\n",d.num,\*d.key,d.items)

// prints num=7 key=v.key items=map[vmethod:true]

}

36. Закрытие тела HTTP-ответа

Делая запрос с помощью стандартной HTTP-библиотеки, вы получаете переменную HTTP-ответа. Даже если вы не читаете тело ответа, всё равно нужно его закрыть. Обратите внимание: это относится и к пустым ответам. О них очень легко забыть, особенно новичкам.  
  
Некоторые новички пытаются закрывать тело ответа, но в неправильном месте.

**package** main

**import** (

"fmt"

"net/http"

"io/ioutil"

)

**func** **main**() {

resp, err := http.Get("https://api.ipify.org?format=json")

**defer** resp.Body.Close()// неправильно

**if** err != nil {

fmt.Println(err)

**return**

}

body, err := ioutil.ReadAll(resp.Body)

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(**string**(body))

}

Этот код будет работать с успешными HTTP-запросами, но в случае сбоя переменная resp может быть nil, что приведёт к runtime panic.  
  
Самый распространённый способ закрыть тело ответа — с помощью вызова defer после проверки ошибочности HTTP-ответа.

**package** main

**import** (

"fmt"

"net/http"

"io/ioutil"

)

**func** **main**() {

resp, err := http.Get("https://api.ipify.org?format=json")

**if** err != nil {

fmt.Println(err)

**return**

}

**defer** resp.Body.Close()// допустимо, в большинстве случаев :-)

body, err := ioutil.ReadAll(resp.Body)

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(**string**(body))

}

В большинстве случаев, когда возникают сбои HTTP-запросов, переменная resp будет nil, а переменная err — non-nil. Но при сбое переадресации обе переменные будут non-nil. Это означает возникновение утечки.  
  
Её можно предотвратить, добавив вызов для закрытия тел ответов non-nil в блоке обработки ошибок HTTP-запросов. Другое решение: использовать один вызов defer для закрытия тел ответов для всех сбойных и успешных запросов.

**package** main

**import** (

"fmt"

"net/http"

"io/ioutil"

)

**func** **main**() {

resp, err := http.Get("https://api.ipify.org?format=json")

**if** resp != nil {

**defer** resp.Body.Close()

}

**if** err != nil {

fmt.Println(err)

**return**

}

body, err := ioutil.ReadAll(resp.Body)

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(**string**(body))

}

Исходная реализация resp.Body.Close() также считывает и отклоняет данные оставшихся тел ответов. Благодаря этому HTTP-соединение может быть повторно использовано для другого запроса, если включено поведение keep alive. Поведение самого последнего HTTP-клиента отличается. Теперь вы ответственны за чтение и отклонение оставшихся данных ответов. Если этого не сделать, то HTTP-соединение вместо повторного использования может быть закрыто. Надеюсь, этот маленький подводный камень будет задокументирован в Go 1.5.  
  
Если для вашего приложения важно повторно использовать HTTP-соединения, то в конце логики обработки ответа может понадобиться добавить что-то вроде этого:

\_, err = io.Copy(ioutil.Discard, resp.Body)

Это будет необходимо, если вы не считываете всё тело ответа немедленно, например при обработке ответов JSON API с помощью подобного кода:

json.NewDecoder(resp.Body).Decode(&data)

37. Закрытие HTTP-соединений

Некоторые HTTP-серверы какое-то время держат сетевые соединения открытыми (согласно спецификации HTTP 1.1 и серверной конфигурации keep alive). По умолчанию стандартная HTTP-библиотека закрывает соединения, только когда об этом просит целевой HTTP-сервер. Тогда при определённых условиях в вашем приложении могут закончиться сокеты / файловые дескрипторы.  
  
Можно попросить библиотеку закрывать соединение после завершения вашего запроса, задав значение true в поле Close переменной запроса.  
  
Другое решение: добавить заголовок запроса Connection и задать ему значение close. Целевой HTTP-сервер тоже должен ответить заголовком Connection: close. Когда библиотека его увидит, она закроет соединение.

**package** main

**import** (

"fmt"

"net/http"

"io/ioutil"

)

**func** **main**() {

req, err := http.NewRequest("GET","http://golang.org",nil)

**if** err != nil {

fmt.Println(err)

**return**

}

req.Close = true

//or do this:

//req.Header.Add("Connection", "close")

resp, err := http.DefaultClient.Do(req)

**if** resp != nil {

**defer** resp.Body.Close()

}

**if** err != nil {

fmt.Println(err)

**return**

}

body, err := ioutil.ReadAll(resp.Body)

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(len(**string**(body)))

}

Можно ещё глобально отключить повторное использование HTTP-соединений. Для этого создайте кастомную конфигурацию HTTP-транспорта.

**package** main

**import** (

"fmt"

"net/http"

"io/ioutil"

)

**func** **main**() {

tr := &http.Transport{DisableKeepAlives: true}

client := &http.Client{Transport: tr}

resp, err := client.Get("http://golang.org")

**if** resp != nil {

**defer** resp.Body.Close()

}

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(resp.StatusCode)

body, err := ioutil.ReadAll(resp.Body)

**if** err != nil {

fmt.Println(err)

**return**

}

fmt.Println(len(**string**(body)))

}

Если вы отправляете на один сервер много запросов, то этого достаточно для сохранения соединения открытым. Но если приложение за короткое время шлёт один-два запроса на много разных серверов, то лучше закрывать соединения сразу после получения ответов. Также можно увеличить лимит на количество открытых файлов. Что лучше — зависит от вашего приложения.

38. Десериализация (unmarshalling) JSON-чисел в интерфейсные значения

Когда вы декодируете/десериализуете JSON-данные в интерфейс, Go по умолчанию обращается с числовыми значениями в JSON как с числами float64. Значит, вот такой код вызовет panic:

**package** main

**import** (

"encoding/json"

"fmt"

)

**func** **main**() {

**var** data = []**byte**(`{"status": 200}`)

**var** result **map**[**string**]**interface**{}

**if** err := json.Unmarshal(data, &result); err != nil {

fmt.Println("error:", err)

**return**

}

**var** status = result["status"].(**int**) // ошибка

fmt.Println("status value:",status)

}

Runtime Panic:  
  
panic: interface conversion: interface is float64, not int  
  
Если JSON-значение, которое вы пытаетесь декодировать, целочисленное, есть несколько вариантов.

* Использовать значение с плавающей запятой как есть :-)
* Преобразовать значение с плавающей запятой в целочисленный тип, который вам нужен.
* **package** main
* **import** (
* "encoding/json"
* "fmt"
* )
* **func** **main**() {
* **var** data = []**byte**(`{"status": 200}`)
* **var** result **map**[**string**]**interface**{}
* **if** err := json.Unmarshal(data, &result); err != nil {
* fmt.Println("error:", err)
* **return**
* }
* **var** status = **uint64**(result["status"].(**float64**)) // хорошо
* fmt.Println("status value:",status)
* }
* Использовать тип Decoder для десериализации JSON и представления JSON-чисел с помощью интерфейсного типа Number.
* **package** main
* **import** (
* "encoding/json"
* "bytes"
* "fmt"
* )
* **func** **main**() {
* **var** data = []**byte**(`{"status": 200}`)
* **var** result **map**[**string**]**interface**{}
* **var** decoder = json.NewDecoder(bytes.NewReader(data))
* decoder.UseNumber()
* **if** err := decoder.Decode(&result); err != nil {
* fmt.Println("error:", err)
* **return**
* }
* **var** status,\_ = result["status"].(json.Number).Int64() // хорошо
* fmt.Println("status value:",status)
* }

Можно использовать строковое представление вашего значения Number, чтобы десериализовать его в другой числовой тип:

**package** main

**import** (

"encoding/json"

"bytes"

"fmt"

)

**func** **main**() {

**var** data = []**byte**(`{"status": 200}`)

**var** result **map**[**string**]**interface**{}

**var** decoder = json.NewDecoder(bytes.NewReader(data))

decoder.UseNumber()

**if** err := decoder.Decode(&result); err != nil {

fmt.Println("error:", err)

**return**

}

**var** status **uint64**

**if** err := json.Unmarshal([]**byte**(result["status"].(json.Number).String()), &status); err != nil {

fmt.Println("error:", err)

**return**

}

fmt.Println("status value:",status)

}

* Использовать тип struct, который преобразует (maps) числовое значение в нужный вам числовой тип.
* **package** main
* **import** (
* "encoding/json"
* "bytes"
* "fmt"
* )
* **func** **main**() {
* **var** data = []**byte**(`{"status": 200}`)
* **var** result **struct** {
* Status **uint64** `json:"status"`
* }
* **if** err := json.NewDecoder(bytes.NewReader(data)).Decode(&result); err != nil {
* fmt.Println("error:", err)
* **return**
* }
* fmt.Printf("result => %+v",result)
* //prints: result => {Status:200}
* }
* Использовать struct для преобразования числового значения в тип json.RawMessage, если требуется отложить декодирование значения.  
    
  Это полезно, если вы должны выполнить декодирование условного JSON-поля (conditional field) в условиях возможности изменения структуры или типа поля.
* **package** main
* **import** (
* "encoding/json"
* "bytes"
* "fmt"
* )
* **func** **main**() {
* records := [][]**byte**{
* []**byte**(`{"status": 200, "tag":"one"}`),
* []**byte**(`{"status":"ok", "tag":"two"}`),
* }
* **for** idx, record := **range** records {
* **var** result **struct** {
* StatusCode **uint64**
* StatusName **string**
* Status json.RawMessage `json:"status"`
* Tag **string** `json:"tag"`
* }
* **if** err := json.NewDecoder(bytes.NewReader(record)).Decode(&result); err != nil {
* fmt.Println("error:", err)
* **return**
* }
* **var** sstatus **string**
* **if** err := json.Unmarshal(result.Status, &sstatus); err == nil {
* result.StatusName = sstatus
* }
* **var** nstatus **uint64**
* **if** err := json.Unmarshal(result.Status, &nstatus); err == nil {
* result.StatusCode = nstatus
* }
* fmt.Printf("[%v] result => %+v\n",idx,result)
* }
* }

39. Сравнение struct, array, slice и map

Можно использовать оператор эквивалентности == для сравнения переменных структур, если каждое поле структуры можно сравнить с помощью этого оператора.

**package** main

**import** "fmt"

**type** data **struct** {

num **int**

fp **float32**

complex **complex64**

str **string**

char **rune**

yes **bool**

events <-**chan** **string**

handler **interface**{}

ref \***byte**

raw [10]**byte**

}

**func** **main**() {

v1 := data{}

v2 := data{}

fmt.Println("v1 == v2:",v1 == v2) // выводит: v1 == v2: true

}

Если хоть одно из полей несравниваемо, то применение оператора эквивалентности приведёт к ошибке компилирования. Обратите внимание, что сравнивать массивы можно только тогда, когда сравниваемы их данные.

**package** main

**import** "fmt"

**type** data **struct** {

num **int** // ok

checks [10]**func**() **bool** // несравниваемо

doit **func**() **bool** // несравниваемо

m **map**[**string**] **string** // несравниваемо

bytes []**byte** // несравниваемо

}

**func** **main**() {

v1 := data{}

v2 := data{}

fmt.Println("v1 == v2:",v1 == v2)

}

Go предоставляет несколько вспомогательных функций для сравнения переменных, которые нельзя сравнивать с помощью операторов сравнения.  
  
Самое популярное решение: использовать функцию DeepEqual() из пакета reflect.

**package** main

**import** (

"fmt"

"reflect"

)

**type** data **struct** {

num **int** // ok

checks [10]**func**() **bool** // несравниваемо

doit **func**() **bool** // несравниваемо

m **map**[**string**] **string** // несравниваемо

bytes []**byte** // несравниваемо

}

**func** **main**() {

v1 := data{}

v2 := data{}

fmt.Println("v1 == v2:",reflect.DeepEqual(v1,v2)) // prints: v1 == v2: true

m1 := **map**[**string**]**string**{"one": "a","two": "b"}

m2 := **map**[**string**]**string**{"two": "b", "one": "a"}

fmt.Println("m1 == m2:",reflect.DeepEqual(m1, m2)) // prints: m1 == m2: true

s1 := []**int**{1, 2, 3}

s2 := []**int**{1, 2, 3}

fmt.Println("s1 == s2:",reflect.DeepEqual(s1, s2)) // prints: s1 == s2: true

}

Помимо невысокой скорости (что может быть критично для вашего приложения), DeepEqual() имеет свои собственные подводные камни.

**package** main

**import** (

"fmt"

"reflect"

)

**func** **main**() {

**var** b1 []**byte** = nil

b2 := []**byte**{}

fmt.Println("b1 == b2:",reflect.DeepEqual(b1, b2)) // prints: b1 == b2: false

}

DeepEqual() не считает пустой слайс эквивалентным nil-слайсу. Это поведение отличается от того, что вы получите при использовании функции bytes.Equal(): она считает эквивалентными nil и пустые слайсы.

**package** main

**import** (

"fmt"

"bytes"

)

**func** **main**() {

**var** b1 []**byte** = nil

b2 := []**byte**{}

fmt.Println("b1 == b2:",bytes.Equal(b1, b2)) // prints: b1 == b2: true

}

DeepEqual() не всегда идеальна при сравнении слайсов.

**package** main

**import** (

"fmt"

"reflect"

"encoding/json"

)

**func** **main**() {

**var** str **string** = "one"

**var** in **interface**{} = "one"

fmt.Println("str == in:",str == in,reflect.DeepEqual(str, in))

//prints: str == in: true true

v1 := []**string**{"one","two"}

v2 := []**interface**{}{"one","two"}

fmt.Println("v1 == v2:",reflect.DeepEqual(v1, v2))

//prints: v1 == v2: false (not ok)

data := **map**[**string**]**interface**{}{

"code": 200,

"value": []**string**{"one","two"},

}

encoded, \_ := json.Marshal(data)

**var** decoded **map**[**string**]**interface**{}

json.Unmarshal(encoded, &decoded)

fmt.Println("data == decoded:",reflect.DeepEqual(data, decoded))

//prints: data == decoded: false (not ok)

}

Если ваши байт-слайсы (или строки) содержат текстовые данные, то, когда понадобится сравнить значения без учёта регистра, вы можете использовать ToUpper() или ToLower() из пакетов bytes и strings (прежде чем прибегнуть к ==, bytes.Equal() или bytes.Compare()). Это сработает для англоязычных текстов, но не для многих других языков. Так что лучше выбрать strings.EqualFold() и bytes.EqualFold().  
  
Если ваши байт-слайсы содержат секретные данные (криптографические хеши, токены и т. д.), которые нужно сравнивать с предоставленной пользователями информацией, обойдитесь без reflect.DeepEqual(), bytes.Equal() или bytes.Compare(). Эти функции сделают приложение уязвимым к атакам по времени. Чтобы избежать утечки информации о времени, используйте функции из пакета crypto/subtle (например, subtle.ConstantTimeCompare()).

40. Восстановление после panic

Функцию recover() можно использовать для поимки/перехвата panic. Это получится, только если вызвать её в блоке defer.  
  
Некорректно:

**package** main

**import** "fmt"

**func** **main**() {

recover() // ничего не делает

panic("not good")

recover() // не будет выполнено :)

fmt.Println("ok")

}

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

**defer** **func**() {

fmt.Println("recovered:",recover())

}()

panic("not good")

}

Вызов recover() сработает, только если будет выполнен в блоке defer.  
  
Неправильно:

**package** main

**import** "fmt"

**func** **doRecover**() {

fmt.Println("recovered =>",recover()) // prints: recovered => <nil>

}

**func** **main**() {

**defer** **func**() {

doRecover() // восстановление panic не произошло

}()

panic("not good")

}

41. Обновление и привязка значений полей в slice, array и map в выражениях for range

Сгенерированные в выражениях range значения данных — это копии реальных элементов коллекций, а не ссылки на исходные элементы. Стало быть, обновление значений не изменит исходные данные. Кроме того, если взять адрес значения, то вы не получите указатель на исходные данные.

**package** main

**import** "fmt"

**func** **main**() {

data := []**int**{1,2,3}

**for** \_,v := **range** data {

v \*= 10 // оригинал не изменился

}

fmt.Println("data:",data) // выводит: [1 2 3]

}

Если вам нужно обновить исходное значение записи в коллекции, то для доступа к данным воспользуйтесь индексным оператором.

**package** main

**import** "fmt"

**func** **main**() {

data := []**int**{1,2,3}

**for** i,\_ := **range** data {

data[i] \*= 10

}

fmt.Println("data:",data) // выводит: [10 20 30]

}

Если коллекция содержит значения указателей, то правила немного меняются. Вам всё ещё нужно использовать индексный оператор, чтобы исходная запись указывала на другое значение. Но вы можете обновлять данные, хранящиеся в целевом месте, с помощью второго значения в выражении for range.

**package** main

**import** "fmt"

**func** **main**() {

data := []\***struct**{num **int**} {{1},{2},{3}}

**for** \_,v := **range** data {

v.num \*= 10

}

fmt.Println(data[0],data[1],data[2]) // prints &{10} &{20} &{30}

}

42. «Скрытые данные» в слайсах

При перенарезке получившийся слайс будет ссылаться на массив исходного слайса. Не забывайте об этом. Иначе может возникнуть непредсказуемое потребление памяти, когда приложение разместит в ней крупные временные слайсы и создаст из них новые, чтобы ссылаться на небольшие куски исходных данных.

**package** main

**import** "fmt"

**func** **get**() []**byte** {

raw := make([]**byte**,10000)

fmt.Println(len(raw),cap(raw),&raw[0]) // выводит: 10000 10000 <byte\_addr\_x>

**return** raw[:3]

}

**func** **main**() {

data := get()

fmt.Println(len(data),cap(data),&data[0]) // выводит: 3 10000 <byte\_addr\_x>

}

Чтобы избежать этой ошибки, удостоверьтесь, что копируете нужные данные из временного слайса (вместо перенарезки).

**package** main

**import** "fmt"

**func** **get**() []**byte** {

raw := make([]**byte**,10000)

fmt.Println(len(raw),cap(raw),&raw[0]) // выводит: 10000 10000 <byte\_addr\_x>

res := make([]**byte**,3)

copy(res,raw[:3])

**return** res

}

**func** **main**() {

data := get()

fmt.Println(len(data),cap(data),&data[0]) // выводит: 3 3 <byte\_addr\_y>

}

43. «Повреждение» данных в слайсах

Допустим, вам нужно переписать путь (хранящийся в слайсе). Чтобы ссылаться на каждую папку, вы его перенарезаете, изменяя имя первой папки, а затем комбинируете имена в новый путь.

**package** main

**import** (

"fmt"

"bytes"

)

**func** **main**() {

path := []**byte**("AAAA/BBBBBBBBB")

sepIndex := bytes.IndexByte(path,'/')

dir1 := path[:sepIndex]

dir2 := path[sepIndex+1:]

fmt.Println("dir1 =>",**string**(dir1)) // выводит: dir1 => AAAA

fmt.Println("dir2 =>",**string**(dir2)) // выводит: dir2 => BBBBBBBBB

dir1 = append(dir1,"suffix"...)

path = bytes.Join([][]**byte**{dir1,dir2},[]**byte**{'/'})

fmt.Println("dir1 =>",**string**(dir1)) // выводит: dir1 => AAAAsuffix

fmt.Println("dir2 =>",**string**(dir2)) // выводит: dir2 => uffixBBBB (not ok)

fmt.Println("new path =>",**string**(path))

}

Так не сработает. Вместо AAAAsuffix/BBBBBBBBB вы получите AAAAsuffix/uffixBBBB. Причина в том, что слайсы обеих папок ссылаются на один и тот же массив данных из исходного слайса пути. То есть исходный путь тоже изменился. Это может быть проблемой для вашего приложения.  
  
Ее можно решить, разместив в памяти новые слайсы и скопировав туда нужные данные. Другой выход: использовать полное выражение слайса (full slice expression).

**package** main

**import** (

"fmt"

"bytes"

)

**func** **main**() {

path := []**byte**("AAAA/BBBBBBBBB")

sepIndex := bytes.IndexByte(path,'/')

dir1 := path[:sepIndex:sepIndex] // полное выражение слайса

dir2 := path[sepIndex+1:]

fmt.Println("dir1 =>",**string**(dir1)) // выводит: dir1 => AAAA

fmt.Println("dir2 =>",**string**(dir2)) // выводит: dir2 => BBBBBBBBB

dir1 = append(dir1,"suffix"...)

path = bytes.Join([][]**byte**{dir1,dir2},[]**byte**{'/'})

fmt.Println("dir1 =>",**string**(dir1)) // выводит: dir1 => AAAAsuffix

fmt.Println("dir2 =>",**string**(dir2)) // выводит: dir2 => BBBBBBBBB (ok now)

fmt.Println("new path =>",**string**(path))

}

Дополнительный параметр в полном выражении управляет ёмкостью нового слайса. Добавление этого параметра к новому слайсу запустит размещение в памяти нового буфера вместо перезаписи данных во второй слайс.

44. «Устаревшие» слайсы

На одни и те же данные могут ссылаться несколько слайсов. Например, когда вы создаёте новый слайс на основе имеющегося. Если такое поведение важно для вашего приложения, позаботьтесь об «устаревших» слайсах.  
  
В какой-то момент добавление данных в один из слайсов приведёт к размещению в памяти нового массива, потому что в старом не хватит места для новых данных. Теперь на старый массив (со старыми данными) ссылаются несколько слайсов.

**import** "fmt"

**func** **main**() {

s1 := []**int**{1,2,3}

fmt.Println(len(s1),cap(s1),s1) // выводит 3 3 [1 2 3]

s2 := s1[1:]

fmt.Println(len(s2),cap(s2),s2) // выводит 2 2 [2 3]

**for** i := **range** s2 { s2[i] += 20 }

// всё ещё ссылается на тот же массив

fmt.Println(s1) // выводит [1 22 23]

fmt.Println(s2) // выводит [22 23]

s2 = append(s2,4)

**for** i := **range** s2 { s2[i] += 10 }

//s1 is now "stale"

fmt.Println(s1) // выводит [1 22 23]

fmt.Println(s2) // выводит [32 33 14]

}

45. Методы и объявления типов

Когда вы определяете новый тип на основе существующего (не интерфейсного), тем самым вы создаёте объявление типа и не наследуете методы, объявленные в существующем типе.  
  
Неправильно:

**package** main

**import** "sync"

**type** myMutex sync.Mutex

**func** **main**() {

**var** mtx myMutex

mtx.Lock() // ошибка

mtx.Unlock() // ошибка

}

Ошибки компилирования:

/tmp/sandbox106401185/main.go:9: mtx.Lock undefined (type myMutex has no field or method Lock) /tmp/sandbox106401185/main.go:10: mtx.Unlock undefined (type myMutex has no field or method Unlock)

Если вам нужны методы из исходного типа, вы можете задать новый тип структуры, встроив исходный в качестве анонимного поля.  
  
Правильно:

**package** main

**import** "sync"

**type** myLocker **struct** {

sync.Mutex

}

**func** **main**() {

**var** lock myLocker

lock.Lock() // ok

lock.Unlock() // ok

}

Объявления интерфейсных типов также сохраняют свои наборы методов.  
  
Правильно:

**package** main

**import** "sync"

**type** myLocker sync.Locker

**func** **main**() {

**var** lock myLocker = new(sync.Mutex)

lock.Lock() // ok

lock.Unlock() // ok

}

46. Как выбраться из кодовых блоков for switch и for select

* уровень: более опытный

Выражение break без метки (label) выводит вас только из внутреннего блока switch/select. Если использовать выражение return — не вариант, тогда лучший выход — задать метку для внешнего цикла.

**package** main

**import** "fmt"

**func** **main**() {

loop:

**for** {

**switch** {

**case** true:

fmt.Println("breaking out...")

**break** loop

}

}

fmt.Println("out!")

}

То же самое и с выражением goto…

47. Итерационные переменные и замыкания в выражениях for

Самая распространённая проблема в Go. Итерационные переменные в выражении for снова используются для каждой итерации. Это значит, что каждое замыкание (aka функциональный литерал), созданное в вашем цикле for, будет ссылаться на ту же переменную (и они получат значение переменной в тот момент, когда начнётся исполнение их горутин).  
  
Некорректно:

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

data := []**string**{"one","two","three"}

**for** \_,v := **range** data {

**go** **func**() {

fmt.Println(v)

}()

}

time.Sleep(3 \* time.Second)

// горутины выводят: three, three, three

}

Простейшее решение (не требующее менять горутины): сохранить текущее значение итерационной переменной в локальной переменной внутри блока цикла for.  
  
Правильно:

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

data := []**string**{"one","two","three"}

**for** \_,v := **range** data {

vcopy := v //

**go** **func**() {

fmt.Println(vcopy)

}()

}

time.Sleep(3 \* time.Second)

// горутины выводят: one, two, three

}

Другое решение: передать текущую итерационную переменную анонимной горутине в виде параметра.  
  
Правильно:

**package** main

**import** (

"fmt"

"time"

)

**func** **main**() {

data := []**string**{"one","two","three"}

**for** \_,v := **range** data {

**go** **func**(in **string**) {

fmt.Println(in)

}(v)

}

time.Sleep(3 \* time.Second)

// горутины выводят: one, two, three

}

Здесь чуть более сложная версия ловушки.  
  
Некорректно:

**package** main

**import** (

"fmt"

"time"

)

**type** field **struct** {

name **string**

}

**func** (p \*field) **print**() {

fmt.Println(p.name)

}

**func** **main**() {

data := []field{{"one"},{"two"},{"three"}}

**for** \_,v := **range** data {

**go** v.print()

}

time.Sleep(3 \* time.Second)

// горутины выводят: three, three, three

}

Правильно:

**package** main

**import** (

"fmt"

"time"

)

**type** field **struct** {

name **string**

}

**func** (p \*field) **print**() {

fmt.Println(p.name)

}

**func** **main**() {

data := []field{{"one"},{"two"},{"three"}}

**for** \_,v := **range** data {

v := v

**go** v.print()

}

time.Sleep(3 \* time.Second)

// горутины выводят: one, two, three

}

Как вы думаете, что вы увидите (и почему), запустив этот код?

**package** main

**import** (

"fmt"

"time"

)

**type** field **struct** {

name **string**

}

**func** (p \*field) **print**() {

fmt.Println(p.name)

}

**func** **main**() {

data := []\*field{{"one"},{"two"},{"three"}}

**for** \_,v := **range** data {

**go** v.print()

}

time.Sleep(3 \* time.Second)

}

48. Вычисление аргумента блока defer (Deferred Function Call Argument Evaluation)

Аргументы для вызовов отложенных функций вычисляются тогда же, когда и выражение defer (а не когда на самом деле выполняется функция).

**package** main

**import** "fmt"

**func** **main**() {

**var** i **int** = 1

**defer** fmt.Println("result =>",**func**() **int** { **return** i \* 2 }())

i++

//выводит: result => 2 (not ok if you expected 4)

}

49. Вызов блока defer

Отложенные вызовы исполняются в конце содержащей их функции, а не в конце содержащего их кодового блока. Новички часто ошибаются, путая правила исполнения отложенного кода с правилами определения области видимости переменной. Это может стать проблемой, если вы долго выполняете функцию с циклом for, которая во время каждой итерации пытается отложить (defer) вызовы очистки ресурсов.

**package** main

**import** (

"fmt"

"os"

"path/filepath"

)

**func** **main**() {

**if** len(os.Args) != 2 {

os.Exit(-1)

}

start, err := os.Stat(os.Args[1])

**if** err != nil || !start.IsDir(){

os.Exit(-1)

}

**var** targets []**string**

filepath.Walk(os.Args[1], **func**(fpath **string**, fi os.FileInfo, err error) **error** {

**if** err != nil {

**return** err

}

**if** !fi.Mode().IsRegular() {

**return** nil

}

targets = append(targets,fpath)

**return** nil

})

**for** \_,target := **range** targets {

f, err := os.Open(target)

**if** err != nil {

fmt.Println("bad target:",target,"error:",err) //выводит ошибку: too many open files

**break**

}

**defer** f.Close() // не будет закрыто в конце этого блока

// сделай что-нибудь с файлом...

}

}

Один из способов решения проблемы — обернуть кодовый блок в функцию.

**package** main

**import** (

"fmt"

"os"

"path/filepath"

)

**func** **main**() {

**if** len(os.Args) != 2 {

os.Exit(-1)

}

start, err := os.Stat(os.Args[1])

**if** err != nil || !start.IsDir(){

os.Exit(-1)

}

**var** targets []**string**

filepath.Walk(os.Args[1], **func**(fpath **string**, fi os.FileInfo, err error) **error** {

**if** err != nil {

**return** err

}

**if** !fi.Mode().IsRegular() {

**return** nil

}

targets = append(targets,fpath)

**return** nil

})

**for** \_,target := **range** targets {

**func**() {

f, err := os.Open(target)

**if** err != nil {

fmt.Println("bad target:",target,"error:",err)

**return**

}

**defer** f.Close() // ok

// сделай что-нибудь с файлом...

}()

}

}

Другое решение: избавиться от выражения defer :-)

50. Ошибки при приведении типов

Сбойные утверждения типов возвращают «нулевое значение» для целевых типов, использованных в операторе утверждения. Наложившись на сокрытие переменных, это может привести к непредсказуемому поведению.  
  
Некорректно:

**package** main

**import** "fmt"

**func** **main**() {

**var** data **interface**{} = "great"

**if** data, ok := data.(**int**); ok {

fmt.Println("[is an int] value =>",data)

} **else** {

fmt.Println("[not an int] value =>",data)

//выводит: [not an int] value => 0 (not "great")

}

}

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

**var** data **interface**{} = "great"

**if** res, ok := data.(**int**); ok {

fmt.Println("[is an int] value =>",res)

} **else** {

fmt.Println("[not an int] value =>",data)

// выводит: [not an int] value => great (as expected)

}

}

51. Блокированные горутины и утечки ресурсов

В выступлении «Go Concurrency Patterns» на конференции Google I/O в 2012-м Роб Пайк рассказал о нескольких фундаментальных concurrency-шаблонах. Один из них — извлечение первого результата.

**func** **First**(query **string**, replicas ...Search) **Result** {

c := make(**chan** Result)

searchReplica := **func**(i **int**) { c <- replicas[i](query) }

**for** i := **range** replicas {

**go** searchReplica(i)

}

**return** <-c

}

Для каждой копии (replica) поиска функция запускает отдельную горутину. Каждая из горутин отправляет свои поисковые результаты в канал результатов. Возвращается первое значение из канала.  
  
А что с результатами от других горутин? И что насчёт них самих?  
  
В функции First() канал результатов не буферизован. Это значит, что возвращается только первая горутина. Все остальные застревают в попытке отправить свои результаты. Получается, что если у вас более одной копии (replica), то при каждом вызове происходит утечка ресурсов.  
  
Чтобы этого избежать, все горутины должны завершиться (exit). Одно из возможных решений: использовать достаточно большой буферизованный канал результатов, способный вместить все результаты.

**func** **First**(query **string**, replicas ...Search) **Result** {

c := make(**chan** Result,len(replicas))

searchReplica := **func**(i **int**) { c <- replicas[i](query) }

**for** i := **range** replicas {

**go** searchReplica(i)

}

**return** <-c

}

Другое решение: использовать выражение select со сценарием (case) default и буферизованный канал на одно значение. Сценарий default позволяет быть уверенным, что горутина не застряла, даже если канал результатов не может принимать сообщения.

**func** **First**(query **string**, replicas ...Search) **Result** {

c := make(**chan** Result,1)

searchReplica := **func**(i **int**) {

**select** {

**case** c <- replicas[i](query):

**default**:

}

}

**for** i := **range** replicas {

**go** searchReplica(i)

}

**return** <-c

}

Также можно использовать специальный канал отмены (special cancellation channel) для прерывания рабочих горутин.

**func** **First**(query **string**, replicas ...Search) **Result** {

c := make(**chan** Result)

done := make(**chan** **struct**{})

**defer** close(done)

searchReplica := **func**(i **int**) {

**select** {

**case** c <- replicas[i](query):

**case** <- done:

}

}

**for** i := **range** replicas {

**go** searchReplica(i)

}

**return** <-c

}

Почему в презентации есть такие баги? Роб Пайк просто не хотел усложнять слайды (slides) своей презентации. Такое объяснение имеет смысл, но это может быть проблемой для новичков, которые используют код, не думая о вероятных проблемах.

52. Применение методов, принимающих значение по ссылке (pointer receiver), к экземплярам значений

Пока значение адресуемо (addressable), к нему можно применять метод, принимающий значение по ссылке. Иными словами, в некоторых случаях вам не нужно иметь версию метода, принимающего параметр по значению.  
  
Но не каждая переменная адресуема. Элементы хеш-таблицы (map) неадресуемы. Переменные, на которые ссылаются через интерфейсы, тоже неадресуемы.

**package** main

**import** "fmt"

**type** data **struct** {

name **string**

}

**func** (p \*data) **print**() {

fmt.Println("name:",p.name)

}

**type** printer **interface** {

print()

}

**func** **main**() {

d1 := data{"one"}

d1.print() //ok

**var** in printer = data{"two"} // ошибка

in.print()

m := **map**[**string**]data {"x":data{"three"}}

m["x"].print() //ошибка

}

Ошибки компилирования:

/tmp/sandbox017696142/main.go:21: cannot use data literal (type data) as type printer **in** assignment: data does not implement printer (print method has pointer receiver)

/tmp/sandbox017696142/main.go:25: cannot call pointer method on m["x"] /tmp/sandbox017696142/main.go:25: cannot take the address of m["x"]

53. Обновление полей значений в хеш-таблице

Если у вас есть таблица, состоящая из структур, то вы не можете обновлять отдельные структурные поля.  
  
Неправильно:

**package** main

**type** data **struct** {

name **string**

}

**func** **main**() {

m := **map**[**string**]data {"x":{"one"}}

m["x"].name = "two" // error

}

Ошибка компилирования:

/tmp/sandbox380452744/main.go:9: cannot assign to m["x"].name

Это не работает, потому что элементы таблицы не адресуемы.  
  
Новичков может дополнительно путать то, что элементы слайсов — адресуемы.

**package** main

**import** "fmt"

**type** data **struct** {

name **string**

}

**func** **main**() {

s := []data {{"one"}}

s[0].name = "two" // ok

fmt.Println(s) // prints: [{two}]

}

Обратите внимание, что когда-то в одном из компиляторов (gccgo) можно было обновлять поля элементов таблицы. Но это быстро пофиксили :-) Также считалось, что такая возможность появится в Go 1.3. Но в то время это было не так важно, так что фича всё ещё висит в списке todo.  
  
Первое обходное решение: использовать временную переменную.

**package** main

**import** "fmt"

**type** data **struct** {

name **string**

}

**func** **main**() {

m := **map**[**string**]data {"x":{"one"}}

r := m["x"]

r.name = "two"

m["x"] = r

fmt.Printf("%v",m) //выводит: map[x:{two}]

}

Второе обходное решение: использовать хеш-таблицу с указателями.

**package** main

**import** "fmt"

**type** data **struct** {

name **string**

}

**func** **main**() {

m := **map**[**string**]\*data {"x":{"one"}}

m["x"].name = "two" //ok

fmt.Println(m["x"]) //выводит: &{two}

}

Кстати, что будет, если выполнить этот код?

**package** main

**type** data **struct** {

name **string**

}

**func** **main**() {

m := **map**[**string**]\*data {"x":{"one"}}

m["z"].name = "what?" //???

}

54. nil-интерфейсы и nil-интерфейсные значения

Это вторая по распространённости ловушка Go. Интерфейсы — не указатели, даже если они так выглядят. Интерфейсные переменные будут nil только тогда, когда их типы и поля значений будут nil.  
  
Интерфейсный тип и поля значений заполняются на основе типа и значения переменной, использованной для создания соответствующей интерфейсной переменной. Если вы попытаетесь проверить, эквивалентна ли переменная nil, то это может привести к непредсказуемому поведению.

**package** main

**import** "fmt"

**func** **main**() {

**var** data \***byte**

**var** in **interface**{}

fmt.Println(data,data == nil) // выводит: <nil> true

fmt.Println(in,in == nil) // выводит: <nil> true

in = data

fmt.Println(in,in == nil) // выводит: <nil> false

//'data' является 'nil', но 'in' — не 'nil'

}

Остерегайтесь этой ловушки, когда у вас есть функция, возвращающая интерфейсы.  
  
Некорректно:

**package** main

**import** "fmt"

**func** **main**() {

doit := **func**(arg **int**) **interface**{} {

**var** result \***struct**{} = nil

**if**(arg > 0) {

result = &**struct**{}{}

}

**return** result

}

**if** res := doit(-1); res != nil {

fmt.Println("good result:",res) // выводит: good result: <nil>

// 'res' не является 'nil', но его значение — 'nil'

}

}

Правильно:

**package** main

**import** "fmt"

**func** **main**() {

doit := **func**(arg **int**) **interface**{} {

**var** result \***struct**{} = nil

**if**(arg > 0) {

result = &**struct**{}{}

} **else** {

**return** nil // возвращает явный 'nil'

}

**return** result

}

**if** res := doit(-1); res != nil {

fmt.Println("good result:",res)

} **else** {

fmt.Println("bad result (res is nil)") // здесь — как и ожидалось

}

}

55. Переменные стека и кучи

Не всегда известно, находится ли переменная в стеке или в куче. Если в С++ создать переменную с помощью оператора new, то она всегда будет в куче. В Go место размещения переменной выбирает компилятор, даже если используются функции new() или make(). Компилятор делает выбор на основании размера и результата «анализа локальности» (escape analysis). Это также означает, что можно возвращать ссылки на локальные переменные, что недопустимо в других языках, например в С и С++.  
  
Если вы хотите знать, где находятся переменные, то передайте -gcflags -m в go build или go run (например, go run -gcflags -m app.go).

56. GOMAXPROCS, согласованность (concurrency) и параллелизм

Go 1.4 и ниже используют только один тред контекста исполнения / ОС. Это значит, что в каждый момент времени может исполняться лишь одна горутина. Начиная с Go 1.5 количество контекстов исполнения стало равно количеству логических процессорных ядер, возвращаемому runtime.NumCPU(). Оно может не совпадать с общим количеством логических ядер в системе, в зависимости от настроек привязки CPU для процесса. Количество можно настроить, изменив переменную среды GOMAXPROCS или вызвав функцию runtime.GOMAXPROCS().  
  
Существует распространённое заблуждение, что GOMAXPROCS представляет собой количество процессоров, которые Go будет использовать для запуска горутин. Документация к функции runtime.GOMAXPROCS() только добавляет неразберихи. Но в описании к переменной GOMAXPROCS (https://golang.org/pkg/runtime/) говорится именно о тредах ОС.  
  
Значение GOMAXPROCS может превышать количество ваших процессоров, верхний предел — 256.

**package** main

**import** (

"fmt"

"runtime"

)

**func** **main**() {

fmt.Println(runtime.GOMAXPROCS(-1)) // выводит: X (1 on play.golang.org)

fmt.Println(runtime.NumCPU()) // выводит: X (1 on play.golang.org)

runtime.GOMAXPROCS(20)

fmt.Println(runtime.GOMAXPROCS(-1)) // выводит: 20

runtime.GOMAXPROCS(300)

fmt.Println(runtime.GOMAXPROCS(-1)) // выводит: 256

}

57. Изменение порядка операций чтения и записи

Go может менять порядок некоторых операций, но общее поведение внутри горутины, где это происходит, не меняется. Однако сказанное не относится к порядку исполнения самих горутин.

**package** main

**import** (

"runtime"

"time"

)

**var** \_ = runtime.GOMAXPROCS(3)

**var** a, b **int**

**func** **u1**() {

a = 1

b = 2

}

**func** **u2**() {

a = 3

b = 4

}

**func** **p**() {

println(a)

println(b)

}

**func** **main**() {

**go** u1()

**go** u2()

**go** p()

time.Sleep(1 \* time.Second)

}

Если запустить этот код несколько раз, то можно увидеть такие комбинации переменных a и b:

1

2

3

4

0

2

0

0

1

4

Самая интересная комбинация — 02 — говорит о том, что b была обновлена раньше a.  
  
Если нужно сохранить порядок операций чтения и записи среди нескольких горутин, то используйте каналы или соответствующие конструкции из пакета sync.

58. Диспетчеризация по приоритетам (Preemptive Scheduling)

Могут появляться разбойничьи (rogue) горутины, не дающие другим горутинам выполняться. Такое случается, если у вас есть цикл for, не позволяющий запустить диспетчер.

**package** main

**import** "fmt"

**func** **main**() {

done := false

**go** **func**(){

done = true

}()

**for** !done {

}

fmt.Println("done!")

}

Цикл for не должен быть пустым. Проблема не исчезнет, пока в цикле содержится код, не запускающий исполнение диспетчера.  
  
Он запускается после сборки мусора, выражений go, операций блокирования каналов, блокирующих системных вызовов и операций блокирования. Также он может работать, когда вызвана невстроенная (non-inlined) функция.

**package** main

**import** "fmt"

**func** **main**() {

done := false

**go** **func**(){

done = true

}()

**for** !done {

fmt.Println("not done!") // не встроена

}

fmt.Println("done!")

}

Чтобы узнать, встроена ли вызываемая вами в цикле функция, передайте -gcflags –m в go build или go run (например, go build -gcflags -m).  
  
Другое решение: явно вызвать диспетчер. Это можно сделать с помощью функции Gosched() из пакета runtime.

**package** main

**import** (

"fmt"

"runtime"

)

**func** **main**() {

done := false

**go** **func**(){

done = true

}()

**for** !done {

runtime.Gosched()

}

fmt.Println("done!")

}

Если вы дочитали до конца и у вас есть комментарии или идеи, добро пожаловать в [дискуссию](https://www.reddit.com/r/golang/comments/360vlb/draft_traps_gotchas_and_common_mistakes_in_go/) на Reddit (и в комментарии здесь, на Хабре. — Примеч. пер.).