

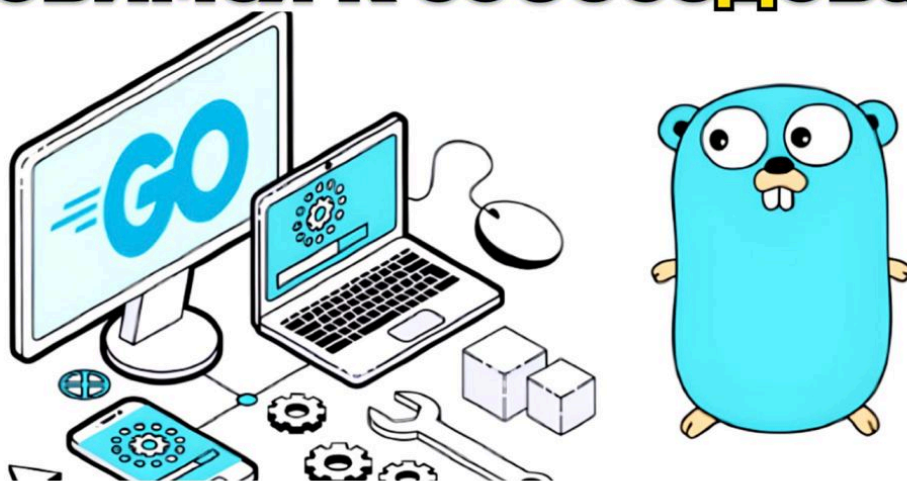


Go — 100 вопросов/заданий с собеседований. Подготовка к собеседованию Golang

🕒 15.06.2024

📁 ПОСТЫ

ГОТОВИМСЯ К СОБЕСЕДОВАНИЮ



@Golang_google

Приветствую тебя, гоффер!

В этой статье разберём более 100 вопросов, они покрывают львиную долю того, что могут спросить на собеседовании джуниор Go-разработчика с практически любой специализацией. Конечно же, в реальной работе на Go требуются *немного* другие скиллы, чем умение быстро ответить на любой вопрос. Однако сложилась *добрая* традиция делать из собеседования викторину с главным призом в виде трудоустройства — к этому нужно быть готовым.

Тем не менее, я уверен, что эта статья будет очень полезна и начинающим, и бывалым гоферам. К каждому вопросу я привёл подробный ответ, поэтому можно использовать этот список как справочник с самой основной теорией по Go. Вперемешку с вопросами также и задачи, есть простые/средние/сложные. Некоторые из этих заданий прямо классические и 99% попадутся на собесе, так что полезно будет их прорешать.

И пара сторонних ресурсов:

- [1900 вопросов с собеседований Go](#)
- [Нереально информативная шпаргалка по Go, всё в одном месте](#)
- [Самое полное интервью Golang Middle](#)
- Шикарный канал с массой годноты по Go — [@Golang_google](#)
- [Целая подборка отличного контента](#): тут и описание продвинутых библиотек Go, и масса полезных в работе инструментов
- [Шпаргалка по вопросам о Go](#)
- [Где бесплатно изучать Golang в 2024. Лучшие курсы, книги, практики](#) [ультимативный гайд!](#)
- [Решения задач с собеседований по Go](#)
- Бесплатные курсы Go

Кстати, начинающие Go-разрабы могут посмотреть примеры хорошего написания кода в этих репозиториях на гитхабе: [пример1](#), [пример2](#), [пример3](#), [пример4](#), [пример5](#). Там можно подтянуть конфиги линтеров, посмотреть как пишется документация к функциям, как проекты структурируются.

Меньше слов, поехали к вопросам!

Оглавление

1. [Как реализовано ООП в Go?](#)
2. [Особенности Go по сравнению с Python и Java, например](#)
3. [Преимущества и недостатки Go](#)
4. [Какие типы данных используются в Go?](#)
5. [Что такое рефлексия в go и чем она полезна?](#)

- [6. Что из себя представляют числовые константы в Go?](#)
- [7. Что такое lock-free структуры данных, и есть ли такие в Go?](#)
- [8. Что такое канал, и какие виды каналов бывают в Go?](#)
- [9. Как работают буферизованные и небуферизованные каналы?](#)
- [10. Можно ли в Go закрыть канал со стороны читателя?](#)
- [11. Расскажи про строки в Go?](#)
- [12. Как эффективно конкатенировать множество строк?](#)
- [13. Что из себя представляет стабы \(stubs\) и моки \(mock\) в контексте тестирования в Go?](#)
- [14. Что делает `runtime.newobject\(\)` ?](#)
- [15. Что такое `\$GOROOT` и `\$GOPATH` ?](#)
- [16. Какие численные типы есть в Go?](#)
- [17. Чем отличается `int` от `uint`?](#)
- [18. Что такое обычный `int` и какие есть нюансы его реализации?](#)
- [19. Какая есть проблема в этом коде?](#)
- [20. Как проверить тип переменной в среде выполнения?](#)
- [21. Как выполнить несколько условий в одном операторе `switch case`?](#)
- [22. Что такое `heap` и `stack` ?](#)
- [23. Где выделяется память под переменную? Можно ли этим управлять?](#)
- [24. Что такое указатель на указатель в Go?](#)
- [25. Реализовать структуру данных "стек" с функциональностью `pop`, `append` и `top`.](#)
- [26. Что такое слайс \(slice\) и массив \(array\)? Чем отличается массив от слайса?](#)
- [27. Как ведут себя срезы в Go на граничных значениях?](#)
- [28. Как работает `append` для слайсов? Можно ли применить к массивам? Напиши свою функцию `append`.](#)
- [29. Как можно добавить элементы в слайс? Что будет если элемент не вмещается в размер слайса?](#)
- [30. Как можно скопировать слайс? Что такое функция `copy`? Как добиться аналогичного поведения `copy` с помощью `append`?](#)
- [31. Как можно нарезать слайс? Какие есть нюансы, подводные камни?](#)
- [32. Что такое table-driven тесты и как их реализовать в Go?](#)
- [33. В каких случаях в Go могут возникнуть `deadlocks`?](#)
- [34. Что такое горутина? Как ее остановить?](#)
- [35. Как завершить много горутин?](#)
- [36. В чём различия горутин от потока системы?](#)
- [37. Реализовать функцию `reverse`, разворачивающую срез целых чисел без использования временного среза](#)
- [38. Что такое пакеты в Go?](#)
- [39. Что такое глобальная переменная?](#)

- [40. Реализовать алгоритм бинарного поиска](#)
- [41. Что выведет этот код?](#)
- [42. Что ты можешь сказать про структуру Reader?](#)
- [43. Как реализована map в Go?](#)
- [44. Что следует учитывать при добавлении элемента в map во время итерации, чтобы избежать недетерминированных результатов?](#)
- [45. Что важно помнить при использовании map типа `any` ?](#)
- [46. Что такое data race \(гонка данных\) в Go?](#)
- [47. Вывести все комбинации символов строки](#)
- [48. Как можно оптимизировать использование памяти в Go, особенно при работе с большими структурами данных?](#)
- [49. Что такое интерфейсы в Go?](#)
- [50. Как сообщить компилятору Go, что наш тип реализует интерфейс?](#)
- [51. На какой стороне описывать интерфейс — на передающей или принимающей?](#)
- [52. Написать функцию, находящую палиндром](#)
- [53. Зачем используется ключевое слово `defer` в Go?](#)
- [54. Что такое замыкания функций?](#)
- [55. Реализовать функцию, подсчитывающую количество гласных](#)
- [56. Что возвращает функция `len\(\)`, если ей передаётся строка в кодировке UTF-8?](#)
- [57. Расскажи про работу с ошибками в Go](#)
- [58. Реализовать функцию последовательности Фибоначчи](#)
- [59. Что такое контекст \(`context`\) в Go и для чего он применяется?](#)
- [60. Как в Go реализованы конструкции циклов?](#)
- [61. FizzBuzz](#)
- [62. Можно ли вернуть из функции несколько значений?](#)
- [63. Объясните разницу между конкурентностью и параллельностью в Go](#)
- [64. Реализуйте функции `min` и `max`](#)
- [65. Какие механизмы синхронизации доступны в Golang?](#)
- [66. Что такое атомарная операция и для чего предназначен пакет `atomic`?](#)
- [67. Как устроен мьютекс?](#)
- [68. Как работает управление памятью в Go?](#)
- [69. Как легче всего проверить срез на пустоту?](#)
- [70. Как можно создать веб-сервер с использованием Golang?](#)
- [71. Что нужно, чтобы две функции были одного типа?](#)
- [72. Реализовать сортировку слиянием, используя горутины и каналы](#)
- [73. Каков побочный эффект использования `time.After` в выражении `select` ?](#)
- [74. Расскажи про `recover`](#)
- [75. Реализовать пересечение двух слайсов](#)
- [76. В чем разница между методами `Time.Sub\(\)` и `Time.Add\(\)` пакета `time` ?](#)

- [77. Что такое теги структур?](#)
- [78. Исправь код](#)
- [79. Если в функции есть return, обязательно ли она вернет то, что указано в return?](#)
- [80. Что такое iota?](#)
- [81. Реализовать генератор случайных чисел](#)
- [82. Что такое псевдоним типа \(type alias\) в Go?](#)
- [83. Как отсортировать массив структур по алфавиту по полю Name ?](#)
- [84. Что такое сериализация? Зачем она нужна?](#)
- [85. Слить N каналов в один](#)
- [86. Как устроен сетевой ввод-вывод в Go?](#)
- [87. Какие побитовые операторы знаешь?](#)
- [88. Как работает init ?](#)
- [89. Сделать конвейер чисел](#)
- [90. Прерывание for/switch](#)
- [91. Дженерики — это про что?](#)
- [92. Написать WorkerPool с заданной функцией](#)
- [93. Что из себя представляет буферизованный и небуферизованный файловый ввод-вывод?](#)
- [94. Что насчёт линтеров?](#)
- [95. Что из себя представляет пакет semaphore в Go?](#)
- [96. Преимущества и недостатки ORM по сравнению с использованием встроенных возможностей для SQL?](#)
- [97. Реализовать обход ссылок из файла](#)
- [98. Поменять местами значения переменных без промежуточной](#)
- [99. Сумма квадратов чисел](#)
- [100. Как можно обработать JSON-данные в Golang?](#)
- [101. Как реализовать rate limiter на Go?](#)

Как реализовано ООП в Go?

Вообще, в Go нет классического ООП в полном смысле, но есть некоторые похожие возможности. В Go нет классов, объектов, исключений и шаблонов. Нет иерархии типов, но есть сами типы — то есть возможность описывать свои типы/структуры. Структурные типы (с методами) служат тем же целям, что и классы в других языках.

В Go мы можем выражать все прямолинейно, в отличие от использования классов, то есть отдельно описывать свойства, а отдельно поведение, и использовать *композицию* вместо привычного наследования, которого в Go нет.

В Go есть интерфейсы — типы, которые объявляют наборы методов. Подобно интерфейсам в других языках, они не имеют реализации. Объекты, которые реализуют все методы интерфейса, автоматически реализуют интерфейс.

Инкапсуляция в Go реализована на уровне пакетов. Имена, начинающиеся со строчной буквы, видны только внутри этого пакета (не являются экспортируемыми). И наоборот — все, что начинается с заглавной буквы — доступно извне пакета.

В Go нет наследования, но есть структуры — типы данных, которые могут включать в себя другие типы, в том числе и структуры (сам этот процесс называется встраивание). При этом и у родительских, и у дочерних структур могут быть свои методы. При встраивании реализация дочерних методов перезаписывает реализацию родительских, выглядит это примерно так:

```
type Parent struct{}

func (c *Parent) Print() {
    fmt.Println("parent")
}

type Child struct {
    Parent
}

func (p *Child) Print() {
    fmt.Println("child")
}

func main() {
    var x Child
    x.Print()
}

// child
```

Кстати, это "наследование" — это embedding. Обсудим некоторые особенности embedding:

- **Простота:** embedding очень прост в использовании — просто определяем один тип внутри другого.
- **Композиция вместо наследования:** вместо того чтобы наследовать методы и поля, Go предпочитает композицию, где один тип может включать в себя другой, дополняя его функциональностью.
- **Поведение и интерфейсы:** если встроенный тип реализует определенный интерфейс, то и тип, в который он встроен, автоматически реализует этот интерфейс.

И ещё один пример embedding:

```
type Engine struct {
    Power int
    Type  string
}

type Car struct {
    Engine
    Brand  string
    Model  string
}

func main() {
    c := Car{
        Engine: Engine{Power: 150, Type: "Petrol"},
        Brand:  "Ford",
        Model:  "Fiesta",
    }
    fmt.Println(c.Power)
}
```

А вот пример со встроенными методами:

```
type Writer interface {
    Write([]byte) (int, error)
}
```

```
}  
  
type Logger struct {  
    Writer  
}
```

Теперь `Logger` автоматически реализует интерфейс `Writer`, но только в том случае, если его встроенное поле `Writer` также реализует методы этого интерфейса.

Несколько важных особенностей:

- **Имена полей и конфликты:** если встроенный и внешний типы имеют поля или методы с одинаковыми именами, приоритет будет у внешнего типа.
- **Неявное поведение:** одним из возможных «подводных камней» является то, что методы встроенного типа становятся частью внешнего типа, что может быть не всегда очевидным при чтении кода.
- **Интерфейсы и встраивание:** в Go можно также встраивать интерфейсы, что позволяет создавать сложные интерфейсы на основе уже существующих.

Особенности Go по сравнению с Python и Java, например

Сравнение с Java. Во-первых, Go компилируется в традиционном смысле этого слова, как и Java. В обоих этих языках строгая статическая типизация. Оба они поддерживают работу в многопоточном режиме.

Пожалуй одним из главных отличий от Go (кроме синтаксиса), является объектно-ориентированная природа языка Java и то, что, для достижения кроссплатформенности, она работает на виртуальной машине JVM (Java Virtual Machine). В то же время Go программа исполняется в своем внутреннем Runtime, и в Go нет классов с конструкторами. Вместо экземпляра методов, иерархии наследия классов и динамического метода, Go предоставляет структуры и интерфейсы.

Помимо этого, отличия между Go и Java в реализации параллелизма. В Go — горутины (сопроцессы, `goroutines`), каналы (`channels`), вся “тяжелая” работа лежит на планировщике (`sheduler`) исполняемой программы. В Java — потоки (`threads`), задачи

(tasks) и более абстрагированные concurrency API – исполнители (executors), callable и фьючерсы (future).



Сравнение с Python. Как и Python, Go имеет сравнительно простой синтаксис, что позволяет быстро реализовывать на нём фичи. В отличие от Python, Go — компилируемый язык (технически Python также компилируется, но не в традиционном смысле). Ожидается, это приводит к сокращению времени выполнения кода. Также в Go большое внимание уделено параллелизму — благодаря `goroutines` (собственным встроенным подпрограммам (сопроцессам)) проще реализовать высокоэффективные параллельные вычисления.

Преимущества и недостатки Go

К преимуществам можно отнести:

- **Простой синтаксис.** В Go нет наследования, классов, объектов и сложных функций. Всё лаконично и аккуратно — это позволяет просто писать на Go и читать чужой код. ~~Для понимания не понадобятся стандарты и комментарии~~ Почти всегда код выглядит читабельно.
- **Лёгкий для новичка.** [Основное руководство Go занимает всего 50 страниц](#). Благодаря строгости и простому синтаксису изучение языка Go — несложная задача даже для тех, у кого совсем нет опыта в разработке. Он построен так, что буквально ведёт разработчика за руку и защищает от ошибок и опечаток.
- **Много встроенных инструментов для разработчиков.** Внутрь языка встроены инструменты тестирования, утилита для создания документации, дополнения для поиска ошибок в коде и другие полезные функции. Поэтому разработка на языке

Go — довольно простой и приятный процесс, нет чувства, что нужно постоянно искать какие-то сторонние инструменты для облегчения работы.

- [typecheck](#) проверит соответствие типов в коде;
- [gas](#) найдет уязвимости;
- [go vet](#) поможет обнаружить ошибки в коде;
- [gofmt](#) правильно отформатирует код, проставит пробелы для выравнивания и табы для отступа;
- [godoc](#) найдет комментарии и подготовит из них мануал к программе, и другие.

Также в Go есть пакет профилирования [pprof](#). Он позволяет узнать, какие фрагменты кода выполняются очень долго, где программа сильно нагружает процессор или занимает много памяти. Результат работы представлен в виде текстового отчета, профайла. Для его использования нужна утилита [graphviz](#).

- **Большое количество библиотек.** Практически для каждой задачи есть готовые стандартные библиотеки внутри языка. Сторонние тоже есть, их список постоянно растёт. К коду на Go можно подключать библиотеки C и C++, которых очень много из-за популярности этих языков.
- **Высокая производительность.** Если переписать код с другого языка на Go (особенно с Python), можно даже без специальной оптимизации повысить производительность в 5–10 раз.
- **Надёжность.** Программы на Go довольно грамотно используют память и вычислительные ресурсы, поэтому работают стабильно (конечно адепты C, любящие контролировать каждый байт, не оценят)
- **Кроссплатформенность.** Мало кого этим сейчас удивишь, но всё же. Язык от Google поддерживается на Windows, [Linux](#), macOS, Android. Также он работает с FreeBSD, OpenBSD и другими UNIX-системами. Код также обладает переносимостью: программы, написанные для одной из этих операционных систем, могут быть легко с перекомпиляцией перенесены на другую ОС.
- **Поддержка UTF-8** (одна из наиболее полных среди всех ЯП)

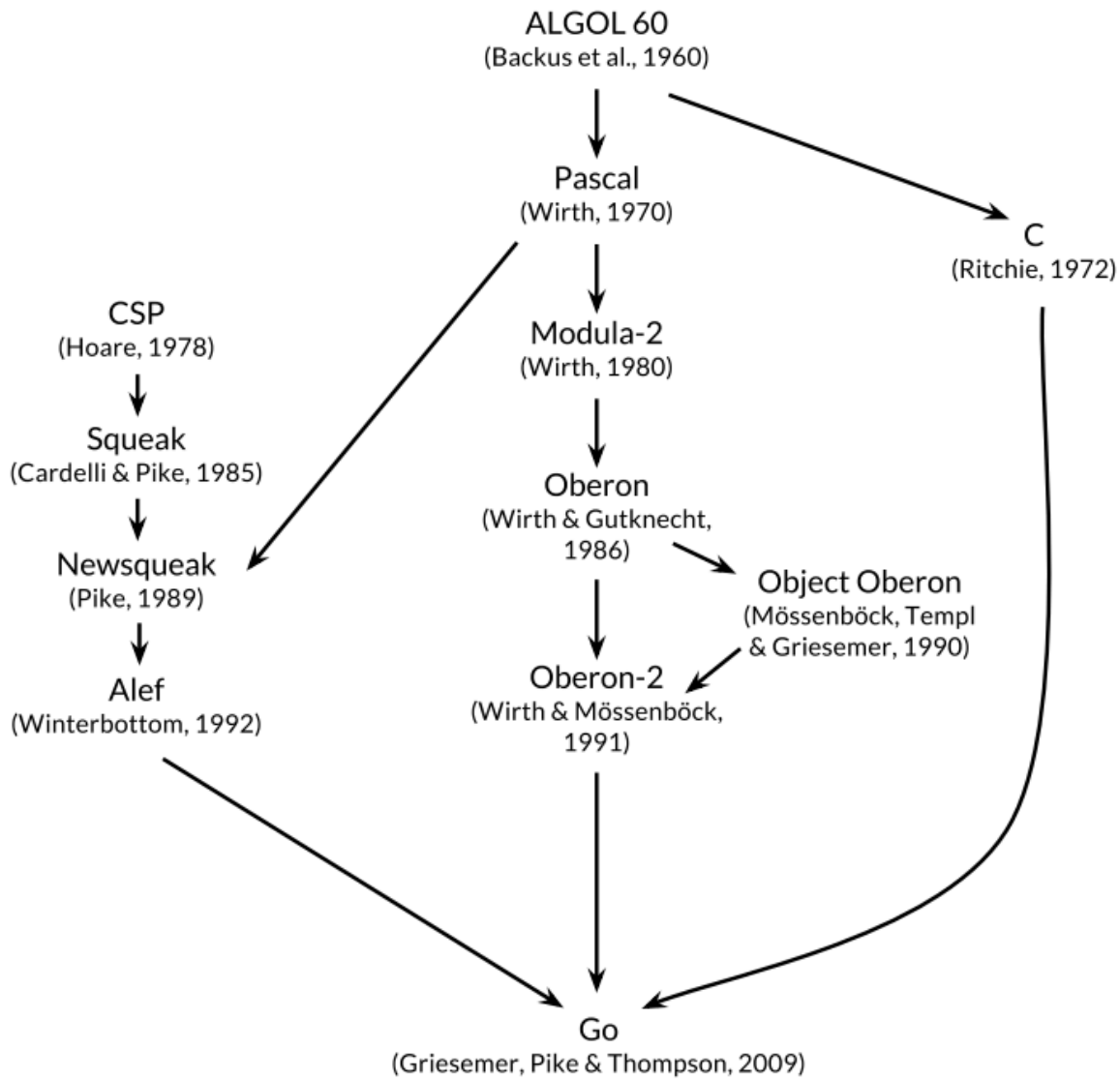
Недостатки:

- **Ограниченный функционал.** Кто бы что ни писал про универсальность, область широкого применения языка Go — это в основном сетевые и серверные приложения (судя по [вот этому опросу](#)). А вот с созданием графических интерфейсов он справляется плохо. Поэтому полностью написать на Go пользовательское приложение будет сложно из-за ограниченных возможностей,

да и в целом он неудобен для многих задач. Его нужно использовать с умом и там, где он действительно нужен.

- **Простота.** Да, это не только плюс, но и минус, поскольку простота вызвана отказом от некоторых особенностей (тот же ООП). Некоторые вещи, доступные на других языках, на Go сделать просто не выйдет. Например, разрабатывать большие проекты из-за отсутствия объектов, полезных для совместной работы с распределённым кодом.
- **Размер.** При компиляции даже простого кода Go легко получить файл в несколько Мб. Конечно, можно обрезать символы отладки и уменьшить объём с помощью упаковщика, но с этим нужно быть аккуратнее.
- **Работа с памятью.** Не существует средства ручного управления памятью; не получится настроить поведение сборщика мусора (Garbage collector).
- **Компилятор,** выбрасывающий в кучу локальные объекты, не способный инлайнить однократно применённую функцию больше 3-х операторов, не способный инлайнить методы из даже одного оператора, если применён defer и т.д.
- **Запятые.** Создатели хотели избавиться от точки-запятой, поэтому её в Go нет. Зато каждая третья строка кода заканчивается на запятую.
- Передача всего исключительно по значению.

Кстати, генеалогическое дерево Go выглядит так, можно блеснуть на собеседе:



Какие типы данных используются в Go?

Go работает со следующими типами:

- Method (метод)
- Boolean (логический тип)
- Numeric (численный)
- String (строковый)
- Array (массив)
- Slice (срезы)
- Struct (структура)

- Pointer (указатель)
- Function (функция)
- Interface (интерфейс)
- Map (карта)
- Channel (канал)

// строки

```
str := "Hello"
```

```
str := `Multiline  
string`
```

// числа

```
num := 3          // int
```

```
num := 3.          // float64
```

```
num := 3 + 4i      // complex128
```

```
num := byte('a')  // byte (alias for uint8)
```

```
var u uint = 7     // uint (unsigned)
```

```
var p float32 = 22.7 // 32-bit float
```

// массивы

```
var numbers [5]int
```

```
numbers := [...]int{0, 0, 0, 0, 0}
```

// срезы

```
slice := []int{2, 3, 4}
```

```
slice := []byte("Hello")
```

// указатели

```
func main () {
```

```
    b := *getPointer()
```

```
    fmt.Println("Value is", b)
```

```
}
```

```
func getPointer () (myPointer *int) {
```

```
    a := 234
```

```
    return &a
```

```
}
```

```
a := new(int)
*a = 234

// преобразование типов
i := 2
f := float64(i)
u := uint(i)
```

Что такое рефлексия в go и чем она полезна?

Рефлексия в Go реализована в пакете `reflect` и представляет собой механизм, позволяющий коду исследовать значения, типы и структуры во время выполнения, без заранее известной информации о них.

Рефлексия полезна в ситуациях, когда нам нужно работать с данными неизвестного типа, например, при сериализации/десериализации данных, реализации ORM систем и так далее.

С помощью рефлексии мы можем, например, определить тип переменной, прочитать и изменить её значения, вызвать методы динамически. Это делает код более гибким, но следует использовать рефлексию осторожно, так как она может привести к сложному и трудночитаемому коду, а также снизить производительность.

Простые примеры:

- **Определение типа переменной:**

```
package main

import (
    "fmt"
    "reflect"
)
```

```
func main() {  
    x := 42  
    fmt.Println("Тип переменной x:", reflect.TypeOf(x))  
}
```

В примере мы используем функцию `reflect.TypeOf()`, чтобы определить тип

- ****Чтение и изменение значений:****

```
```go  
package main

import (
 "fmt"
 "reflect"
)

func main() {
 x := 42
 v := reflect.ValueOf(&x).Elem() // Получаем reflect.Value

 fmt.Println("Исходное значение x:", x)
 v.SetInt(43) // Изменяем значение x
 fmt.Println("Новое значение x:", x)
}
```

Здесь мы используем `reflect.ValueOf()` для получения `reflect.Value` переменной `x`, а затем изменяем её значение с помощью `SetInt()`.

- **Динамический вызов методов:**

```
package main

import (
 "fmt"
```

```
"reflect"
)

type MyStruct struct {
 Field int
}

func (m *MyStruct) UpdateField(val int) {
 m.Field = val
}

func main() {
 x := MyStruct{Field: 10}

 // Получаем reflect.Value структуры
 v := reflect.ValueOf(&x)

 // Получаем метод по имени
 method := v.MethodByName("UpdateField")

 // Вызываем метод с аргументами
 method.Call([]reflect.Value{reflect.ValueOf(20)})

 fmt.Println("Обновленное значение поля:", x.Field)
}
```

В этом примере мы создаем экземпляр структуры `MyStruct`, получаем метод

[Что из себя представляют числовые константы в Go?](#)

**Числовые константы** в Go — это фиксированные значения, которые не из

Они принимают свой тип (например, `int`, `float64`) только когда это не

Простой пример:



```
```go
package main

import "fmt"

const (
    Big = 1 << 100
    Small = Big >> 99
)

func needInt(x int) int { return x*10 + 1 }
func needFloat(x float64) float64 { return x * 0.1 }

func main() {
    fmt.Println(needInt(Small))
    fmt.Println(needFloat(Small))
    fmt.Println(needFloat(Big))
}
```

Что такое lock-free структуры данных, и есть ли такие в Go?

Lock-free структуры данных — это тип структур данных, разработанных для многопоточных операций без использования традиционных блокировок, таких как мьютексы.

Основная идея заключается в том, чтобы обеспечить безопасность потоков и избежать проблем, связанных с блокировками, включая взаимную блокировку (deadlock) и узкие места производительности (bottlenecks).

Lock-free структуры данных обычно используют атомарные операции, такие как CAS (compare-and-swap), для обеспечения согласованности данных между потоками. Эти операции позволяют потокам соревноваться за изменение данных, но гарантируют, что только один поток сможет успешно изменить данные в любой момент времени.

В Go, языке с поддержкой конкурентности, есть несколько примеров lock-free или почти lock-free структур данных, особенно в стандартной библиотеке. Например:

1. **Каналы:** хотя каналы в Go не являются полностью lock-free, они предоставляют высокоуровневый способ обмена данными между горутинами без явного использования блокировок.
2. **Атомарные операции:** пакет `sync/atomic` в Go предоставляет примитивы для атомарных операций, которые являются ключевыми компонентами для создания lock-free структур данных.
3. `sync.Map` : предназначен для использования в кейсах, где ключи в основном не меняются, и он использует оптимизации для уменьшения необходимости блокировок.

Что такое канал, и какие виды каналов бывают в Go?

Каналы — это инструменты коммуникации между горутинами.

Технически это конвейер/труба, откуда можно считывать или помещать данные. То есть одна горутина может отправить данные в канал, а другая — считать помещенные в этот канал данные.

Для создания канала в Go есть ключевое слово `chan`. Канал может передавать данные только одного типа.

```
package main

import "fmt"

func main() {
    var c chan int
    fmt.Println(c)
}
```

При простом определении переменной канала она имеет значение `nil`, то есть по сути канал неинициализирован. Для инициализации применяется функция `make()`.

В зависимости от определения емкости канала он может быть **буферизированным** или **небуферизированным**.

Для создания небуферизированного канала вызывается функция `make()` без указания емкости канала:

```
var intCh chan int = make(chan int)
```

Буферизированные каналы также создаются с помощью функции `make()`, только в качестве второго аргумента в функцию передается емкость канала. Если канал пуст, то получатель ждет, пока в канале появится хотя бы один элемент.

```
chanBuf := make(chan bool, 3)
```

С каналом можно произвести 4 действия:

- создать канал
- записать данные в канал
- вычитать что-то из канала
- закрыть канал

Однонаправленные каналы: в Go можно определить канал, как доступный только для отправки данных или только для получения данных.

Канал может быть возвращаемым значением функции. Однако следует внимательно подходить к операциям записи и чтения в возвращаемом канале.

[Анатомия каналов в Go](#)

Как работают буферизованные и небуферизованные каналы?

Буферизованные каналы позволяют вам быстро помещать задания в очередь, чтобы вы могли работать с большим количеством запросов и обрабатывать их позже. Кроме того, буферизованные каналы можно использовать в качестве семафоров, ограничивая пропускную способность вашего приложения.

Суть: все входящие запросы перенаправляются на канал, который обрабатывает их по очереди. Завершая обработку запроса, канал отправляет исходному, вызвавшему сообщение о готовности обработать новый запрос. Таким образом, ёмкость буфера канала ограничивает количество одновременных запросов, которые он может хранить.

Вот так выглядит код, который реализует данный метод:

```
package main

import (
    "fmt"
)

func main() {
    numbers := make(chan int, 5)
    // канал numbers не может хранить более пяти целых чисел — это буфер
    counter := 10
    for i := 0; i < counter; i++ {
        select {
            // здесь происходит обработка
            case numbers <- i * i:
                fmt.Println("About to process", i)
            default:
                fmt.Print("No space for ", i, " ")
        }
    }
    // мы начинаем помещать данные в numbers, однако когда канал заполнен
}
fmt.Println()
for {
    select {
        case num := <- numbers:
            fmt.Print("*", num, " ")
    }
}
```

```
default:
    fmt.Println("Nothing left to read!")
    return
}
}
```

Аналогично, мы пытаемся считывать данные из `numbers`, используя цикл `for`. Когда все данные из канала считаны, выполнится ветка `default` и программа завершится с помощью оператора `return`.

При выполнении кода выше мы получаем такой вывод:

```
$ go run bufChannel.go
About to process 0
. . .
About to process 4
No space for 5 No space for 6 No space for 7 No space for 8 No space
for 9
*0 *1 *4 *9 *16 Nothing left to read!
```

В общем:

- **буферизированный канал** заблокирует горутину только в том случае, если весь буфер забит. И происходит попытка еще одной записи. Как только будет выполнено чтение из канала – горутина разблокируется. В случае, если горутина всего одна (только функция `main`) и канал её заблокирует — программа выпадет с ошибкой, так как все горутинны заблокированы и выполнять нечего.
- **небуферизированный канал** заблокирует горутину до момента, пока с него ничего не прочитают.

Можно ли в Go закрыть канал со стороны читателя?

Закрытие канала обычно выполняется отправителем, а не получателем. Это связано с тем, что закрытие канала со стороны получателя может привести к панике при попытке отправителя записать в уже закрытый канал.

Однако, в некоторых случаях, получатель может определить, что данные больше не нужны, и хочет уведомить отправителя о прекращении отправки. В таком случае, обычно используется дополнительный канал, называемый каналом управления или сигнальным каналом, который получатель может использовать для отправки сигнала об остановке. После получения сигнала, отправитель может корректно закрыть основной канал данных.

Простой пример:

```
func main() {
    dataCh := make(chan int)
    stopCh := make(chan struct{})

    go func() {
        for {
            select {
            case data, ok := <-dataCh:
                if !ok {
                    // Канал закрыт, прекращаем обработку
                    return
                }
                // Обработка данных
                fmt.Println(data)
            case <-stopCh:
                // Получен сигнал остановки, закрываем канал dataCh
                close(dataCh)
                return
            }
        }
    }()

    // Отправка данных в канал
    dataCh <- 1
    dataCh <- 2
}
```

```
// Отправка сигнала остановки
stopCh <- struct{}{}

}
```

`stopCh` используется для уведомления горутины о необходимости закрыть канал `dataCh`. Это безопасный способ обеспечить корректное управление жизненным циклом канала.

Расскажи про строки в Go?

Для представления строк в Go поддерживается тип данных `string`. Строка Go — это просто массив байт. Исходя из этого, если применим функцию `len()` к строке, то получим количество байт. В одном байте может храниться любой символ ASCII, однако для хранения одного символа Unicode обычно требуется несколько байтов. В общем, чтобы посчитать именно количество символов, необходимо преобразовать строку в тип `rune`. Еще одним способом определения длины строки является функция `RuneCountInString` пакета `utf8`.

Вообще, руна (тип данных `rune`) — это значение `int32`, которое используется для представления одного кодового пункта Unicode. Руна представляет собой целое значение и используется для представления отдельных символов Unicode или, реже, для предоставления информации о форматировании.

Объявление и инициализация руны:

```
var r rune = 'A'
```

Преобразование строки в срез рун:

```
s := "Привет"
runes := []rune(s)
```

Итерация по рунам в строке:

```
for _, r := range "Привет" {
    fmt.Printf("%c ", r)
}
// П р и в е т
```

Обратное преобразование среза рун в строку:

```
runes := []rune{'П', 'р', 'и', 'в', 'е', 'т'}
s := string(runes) // "Привет"
```

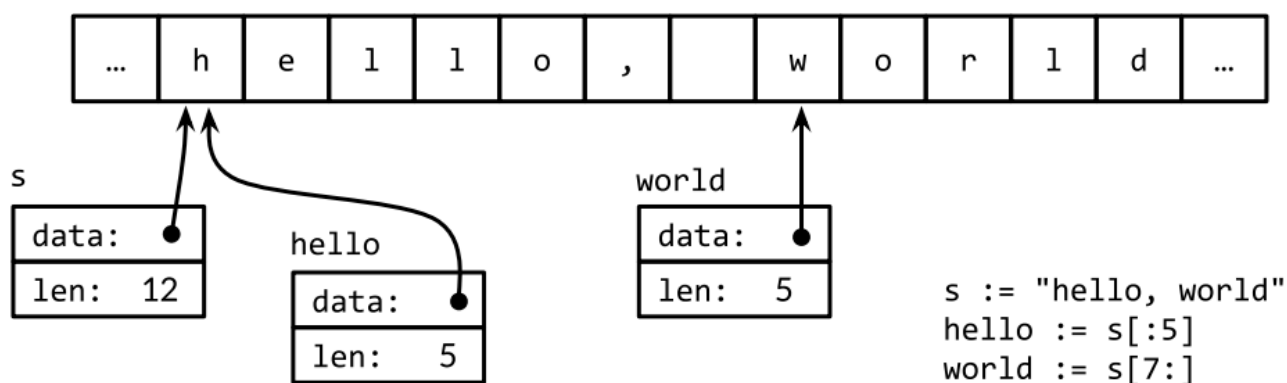
Получение Unicode-кода руны:

```
r := 'А'
code := int32(r) // 65
```

Проверка длины строки в рунах:

```
s := "Привет"
length := utf8.RuneCountInString(s) // 6
```

Строка "hello, world" и 2 её подстроки:



Как эффективно конкатенировать множество строк?

Строки в Go можно складывать (конкатенировать). Для многих операций есть стандартные пакеты, к примеру `strings`, `fmt`. В целом, есть 2 основных варианта конкатенации, но один из них в некоторых условиях может быть очень неэффективным.

Напишем код с функцией `concat`, которая объединяет все строковые элементы среза с помощью оператора `+=`:

```
func concat(values []string) string {
    s := ""
    for _, value := range values {
        s += value
    }
    return s
}
```

В этой реализации мы забываем базовую характеристику строки: ее *неизменность*. Следовательно, с каждой итерацией `s` не обновляется, вместо этого в памяти создается новая строка, что сильно влияет на время выполнения этой функции.

Благо есть пакет `strings` и структура `Builder`, чуть переделаем нашу функцию `concat`:

```
func concat(values []string) string {
    sb := strings.Builder{} // создается strings.Builder
    for _, value := range values {
        _, _ = sb.WriteString(value) // добавляется строка
    }
    return sb.String() // возвращается результирующая строка
}
```

А теперь напишем другую реализацию `concat`, используя `Grow` из `strings.Builder`:

```
func concat(values []string) string {
    total := 0
    for i := 0; i < len(values); i++ { // проводятся итерации по каждой
        total += len(values[i])
    }
    sb := strings.Builder{}
    sb.Grow(total) // вызывается Grow с аргументом, равным этому общему
    for _, value := range values {
        _, _ = sb.WriteString(value)
    }
    return sb.String()
}
```

Перед началом итераций мы вычисляем общее количество байтов, которое будет содержать окончательная строка, и присваиваем это значение переменной `total`. Обратите внимание, что нас интересует не количество рун, а количество байтов, поэтому мы используем функцию `len`. Затем мы вызываем `Grow`, чтобы гарантировать наличие места для байтов `total`, прежде чем проводить итерации по строкам.

Запустим бенчмарк для 3 версий нашей функции `concat` (v1 с использованием `+=`, v2 с использованием `strings.Builder{} без предварительного резервирования места в памяти` и v3 с использованием `strings.Builder{} с предварительным резервированием`). Входной срез содержит 1000 строк, и каждая строка содержит 1000 байт:

```
BenchmarkConcatV1-4 16 72291485 ns/op
BenchmarkConcatV2-4 1188 878962 ns/op
BenchmarkConcatV3-4 5922 190340 ns/op
```

Как видим, последний способ самый эффективный. Почему двукратное итерирование по входному срезу может ускорить код? Дело в том, что если для среза с заданной длиной или емкостью не выделено место заранее, то этот срез будет продолжать

расти каждый раз, когда окажется заполненным, что приведет к дополнительным выделениям памяти и копиям.

`strings.Builder` — рекомендуемое решение для конкатенации списка строк. Обычно это решение следует использовать в циклах.

Если просто нужно объединить несколько строк, использование `strings.Builder` не рекомендуется, так как это сделает код менее читаемым, чем использование оператора `+=` или `fmt.Sprintf`.

Классная книга по теме: «100 ошибок Go и как их избежать» — Тейва Харшани

Что из себя представляет стабы (stubs) и моки (mock) в контексте тестирования в Go?

Стабы (stubs) и моки (mocks) являются техниками, используемыми для изоляции тестируемого кода от внешних зависимостей во время тестирования в Go.

Стабы — это фейковые объекты, которые предоставляют предопределенные ответы на вызовы методов во время тестирования.

```
package main

import "fmt"

type DatabaseStub struct{}

func (db *DatabaseStub) GetUserName(id int) string {
    return "Alice"
}

type Database interface {
    GetUserName(id int) string
}
```

```
func PrintUserName(db Database, id int) {  
    name := db.GetUserName(id)  
    fmt.Println(name)  
}  
  
func main() {  
    dbStub := &DatabaseStub{  
        PrintUserName(dbStub, 1)  
    }  
}
```

Моки — это более продвинутые фейковые объекты, которые, кроме предоставления предопределенных ответов, также проверяют, как и когда методы были вызваны в тестах, что помогает в проверке взаимодействия между объектами.

```
package main  
  
import (  
    "github.com/stretchr/testify/mock"  
    "testing"  
)  
  
type DatabaseMock struct {  
    mock.Mock  
}  
  
func (db *DatabaseMock) GetUserName(id int) string {  
    args := db.Called(id)  
    return args.String(0)  
}  
  
func TestPrintUserName(t *testing.T) {  
    dbMock := new(DatabaseMock)  
    dbMock.On("GetUserName", 1).Return("Alice")  
  
    name := dbMock.GetUserName(1)  
}
```

```
dbMock.AssertExpectations(t)
}
```

В первом примере создается стаб `DatabaseStub`, который имеет метод `GetUserName`. Во втором примере создается мок `DatabaseMock` с использованием библиотеки `testify`, который проверяет, был ли метод `GetUserName` вызван с правильным аргументом.

****Что делает `runtime.newobject()`?****

Краткость — сестра таланта:

- `runtime.newobject()` выделяет память в куче.

Как ответ на собесе пойдёт, а подробнее в [официальных доках](#)

****Что такое `\$GOROOT` и `\$GOPATH`?****

`$GOROOT` — каталог для стандартной библиотеки, включая исполняемые файлы и исходный код. Короче, местоположение всей бинарной сборки Go и исходных кодов.

`$GOPATH` — каталог для внешних пакетов. Или, что то же самое, местоположение всей бинарной сборки Go и исходных кодов.

Какие численные типы есть в Go?

В Go есть несколько основных числовых типов, вот они:

Целые числа (Integers):

- `int8` – 8-битное знаковое целое число (-128 до 127).
- `int16` – 16-битное знаковое целое (-32768 до 32767).
- `int32` – 32-битное знаковое целое (-2147483648 до 2147483647).
- `int64` – 64-битное знаковое целое (-9223372036854775808 до 9223372036854775807).
- `int` – знаковое целое число, размер зависит от архитектуры (32-битное или 64-битное).
- `uint8` (беззнаковый) – 8-битное целое число (0 до 255).
- `uint16` (беззнаковый) – 16-битное целое (0 до 65535).
- `uint32` (беззнаковый) – 32-битное целое (0 до 4294967295).
- `uint64` (беззнаковый) – 64-битное целое (0 до 18446744073709551615).
- `uint` (беззнаковый) – целое число, размер которого зависит от архитектуры (32-битное или 64-битное).

Числа с плавающей запятой (Floating Point Numbers):

- `float32` – 32-битное число с плавающей запятой, представляющее значение с плавающей точкой одинарной точности.
- `float64` – 64-битное число с плавающей запятой, представляющее значение с плавающей точкой двойной точности.

Комплексные числа (Complex Numbers):

- `complex64` – содержит два значения `float32`, представляющих действительную и мнимую части комплексного числа.
- `complex128` – содержит два значения `float64`, представляющих действительную и мнимую части комплексного числа.

Псевдонимы (Aliases):

- `byte` – псевдоним для `uint8`.
- `rune` – псевдоним для `int32`, представляющий юникодный символ.

Чем отличается `int` от `uint`?

`int` содержит диапазон от отрицательных значений до положительных.

`uint` – это диапазон от 0 в сторону увеличения положительных значений.

`int64` : -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 ,

`uint64` : от 0 до 18 446 744 073 709 551 615.

Вот и всё отличие

Что такое обычный `int` и какие есть нюансы его реализации?

В зависимости от того, какая архитектура платформы, на которой мы исполняем код, компилятор преобразует `int` в `int32` для 32-разрядной архитектуры и в `int64` для 64-разрядной архитектуры. Из самого названия типа следует, что `int32` занимает 4 байта (32/8), `int64` занимает 8 байтов (64/8).

Ещё пару вещей про `int` :

- Деление `int` на 0 в Go невозможно и вызовет ошибку компилятора. Тогда как деление `float` на 0 дает в своем результате бесконечность.
- Для преобразования строки в `int` и наоборот необходимо использовать функции из пакета `strconv` стандартной библиотеки Go. При этом, для преобразования строк в/из `int` и `int64` используются разные функции, `strconv.Atoi` и `strconv.Itoa` для `int` , `strconv.ParseInt` и `strconv.FormatInt` для `int64` соответственно.

Какая есть проблема в этом коде?

```
var counter int
for i := 0; i < 1000; i++ {
    go func() {
        counter++
    }()
}
```

Тут у нас налицо проблема с синхронизацией доступа к переменной `counter`. Так как мы запускаем каждую итерацию цикла в отдельной горутине, то не гарантируется порядок выполнения операции инкремента, что может привести к непредсказуемым результатам.

Исправленный код с использованием мьютекса для синхронизации доступа к переменной `counter` выглядит так:

```
var counter int
var mu sync.Mutex

for i := 0; i < 1000; i++ {
    go func() {
        mu.Lock()
        counter++
        mu.Unlock()
    }()
}
```

Мы объявляем переменную `mu` типа `sync.Mutex`, которая используется для блокировки доступа к переменной `counter` в каждой горутине. Метод `Lock()` блокирует доступ к мьютексу, а метод `Unlock()` освобождает его после выполнения инкремента. Таким образом, мы гарантируем правильную работу с переменной `counter` в многопоточной среде.

Как проверить тип переменной в среде выполнения?

Лучшим способом проверки типа переменной при выполнении является `Type Switch` (переключатель типов). Он оценивает переменные по типу, а не значению. Каждый такой переключатель содержит не менее одного `case`, который выступает в роли инструкции условия, а также кейс `default`, которые выполняются, если ни один из кейсов не верен.

Например, можно создать `Type Switch`, проверяющий, содержит ли значение `i` интерфейса тип `int` или `string`:

```
package main
import "fmt"

func do(i interface{}) {
    switch v := i.(type) {
        case int:
            fmt.Printf("Double %v is %vn", v, v*2)
        case string:
            fmt.Printf("%q is %v bytes longn", v, len(v))
        default:
            fmt.Printf("I don't know type %T!n", v)
    }
}

func main() {
    do(21)
    do("hello")
    do(true)
}
```

Как выполнить несколько условий в одном операторе `switch case`?

Ну, во-первых, можно использовать несколько условий в одном операторе `switch case` , разделяя их запятыми. Например:

```
switch x {
case 1, 2, 3:
    fmt.Println("x is 1, 2, or 3")
case 4, 5, 6:
    fmt.Println("x is 4, 5, or 6")
default:
    fmt.Println("x is not in any of the above cases")
}
```

В этом примере мы проверяем значение переменной `x` на соответствие нескольким условиям: 1, 2 или 3 в первом `case` , 4, 5 или 6 во втором `case` . Если значение `x` не соответствует ни одному из этих условий, выполняется блок `default` .

Также можно использовать ключевое слово `fallthrough` для перехода к следующему `case` без проверки условия. Например:

```
switch x {
case 1:
    fmt.Println("x is 1")
    fallthrough
case 2:
    fmt.Println("x is 1 or 2")
case 3:
    fmt.Println("x is 3")
default:
    fmt.Println("x is not in any of the above cases")
}
```

Тут при `x=1` сначала выполнится блок кода в первом `case` , затем блок кода во втором `case` (без проверки условия), так как мы использовали `fallthrough` . При `x=2` выполнится только блок кода во втором `case` . При `x=3` выполнится блок кода в третьем `case` . При любом другом значении `x` выполнится блок `default` .

Что такое `heap` и `stack`?

Стек (stack) — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Стек быстрый, так как часто привязан к кэшу процессора. Размер стека ограничен, и задаётся при создании потока.

Куча (heap) — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные.

В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. То что попадает в кучу, живёт там пока не придёт GC.

Но почему стек так быстр? Основных причин две:

- Стеку не нужно иметь сборщик мусора (garbage collector). Как мы уже упоминали, переменные просто создаются и затем вытесняются, когда функция завершается. Не нужно запускать сложный процесс освобождения памяти от неиспользуемых переменных и т.п.
- Стек принадлежит одной горутине, переменные не нужно синхронизировать в сравнении с теми, что находятся в куче. Что также повышает производительность

Где выделяется память под переменную? Можно ли этим управлять?

Прямых инструментов для управления выделением памяти у нас нет. Но есть кое-что, позволяющее понять механизм выделения памяти, чтобы использовать её эффективно.

Память под переменную может быть выделена в куче (heap) или стеке (stack). Очень приблизительно:

- **Стек** содержит последовательность переменных для заданной горютины (как только функция завершила работу, переменные вытесняются из стека)
- **Куча** содержит общие (shared) переменные (глобальные и т.п.)

Давайте рассмотрим простой пример, в котором мы возвращаем значение:

```
func getFooValue() foo {  
    var result foo  
    // какое-нибудь действие  
    return result  
}
```

Здесь переменная `result` создаётся в текущей горютине. И эта переменная помещается в **стек**. Как только функция завершает работу, клиент получает **копию этой переменной**. Исходная переменная вытесняется из стека. Эта переменная всё ещё существует в памяти, до тех пор, пока не будет затёрта другой переменной, но к этой переменной **уже нельзя получить доступ**.

Теперь тот же пример, но с **указателем**:

```
func getFooPointer() *foo {  
    var result foo  
    // Do something
```

```
    return &result  
}
```

Переменная `result` также создаётся текущей горутинной, но клиент получает указатель (копию адреса переменной). Если `result` вытеснена из стека, клиент функции **не сможет получить доступ к переменной**.

В подобном сценарии компилятор Go вынужден переместить переменную `result` туда, где она может быть доступна (shared) – **в кучу (heap)**.

Хотя есть и исключение. Для примера:

```
func main() {  
    p := &foo{  
    f(p)  
}
```

Поскольку мы вызываем функцию `f()` в той же горутине, что и функцию `main()`, переменную `p` не нужно перемещать. Она просто находится в стеке и вложенная функция `f()` будет иметь к ней доступ.

В качестве заключения, когда мы создаём функцию — поведением по умолчанию должно быть использование **передачи по значению**, а не по указателю. Указатель должен быть использован только когда мы **действительно** хотим переиспользовать данные.

Что такое указатель на указатель в Go?

В Go, указатель хранит адрес памяти другой переменной. Указатель на указатель — это переменная, которая хранит адрес памяти другого указателя, указывающий на некоторое значение или объект.

В целом, ничего сложного, вот тут у нас указатель на указатель:

```
package main

import "fmt"

func main() {
    a := 100
    var b *int = &a // b — указатель на переменную a
    var c **int = &b // c — указатель на указатель b

    fmt.Println("Значение a:", a) // Исходное значение
    fmt.Println("Адрес a:", &a) // Адрес переменной a
    fmt.Println("Значение b:", b) // Адрес, хранящийся в b (адрес a)
    fmt.Println("Разыменование b:", *b) // Разыменование b (значение a)
    fmt.Println("Значение c:", c) // Адрес, хранящийся в c (адрес b)
    fmt.Println("Разыменование c:", *c) // Разыменование c (значение b)
    fmt.Println("Двойное разыменование c:", **c) // Двойное разыменовани
}
```

- * a — обычная переменная типа `int`.
- * b — указатель на `int`, который хранит адрес переменной `a`.
- * c — указатель на указатель на `int`, который хранит адрес переменной `b`.

Таким образом, `c` является указателем на указатель. Он не только позволяет нам получить доступ к значению `a` через двойное разыменование (`**c`), но и изменять адрес, на который указывает `b`, что может быть полезно в некоторых сценариях, например, при передаче указателя в функцию для его модификации.

Реализовать структуру данных “стек” с функциональностью **`**pop**`**, **`**append**`** и **`**top**`**.

Очень простая реализация с использованием слайсов.

```
type Stack struct {  
    items []int  
}
```

Сначала мы определим тип `Stack` с полем `items`. Этот стек отвечает за хранение целых чисел, но здесь может быть любой другой необходимый тип данных.

Два наиболее важных метода стека – `push` и `pop`. Помещение элемента в стек добавляет его в самую верхнюю позицию, а удаление из стека извлекает самый верхний элемент.

```
func (s *Stack) Push(data int) {  
    s.items = append(s.items, data)  
}  
  
func (s *Stack) Pop() {  
    if s.IsEmpty() {  
        return  
    }  
    s.items = s.items[:len(s.items)-1]  
}
```

Эти методы работают с указателями на тип `Stack`.

`Push` добавляет элемент в `s.items`.

`Pop` удаляет самый верхний элемент.

Определим еще три полезных метода.

```
func (s *Stack) Top() (int, error) {  
    if s.IsEmpty() {  
        return 0, fmt.Errorf("stack is empty")  
    }  
    return s.items[len(s.items)-1], nil  
}
```

```
func (s *Stack) IsEmpty() bool {
    if len(s.items) == 0 {
        return true
    }
    return false
}

func (s *Stack) Print() {
    for _, item := range s.items {
        fmt.Print(item, " ")
    }
    fmt.Println()
}
```

Top возвращает самый верхний элемент в стеке. Если стек пуст, он возвращает нулевое значение и ошибку, говорящую о том, что стек пуст.

IsEmpty возвращает true, если стек пуст, и false в противном случае.

Print итерируется по стеку и выводит элементы.

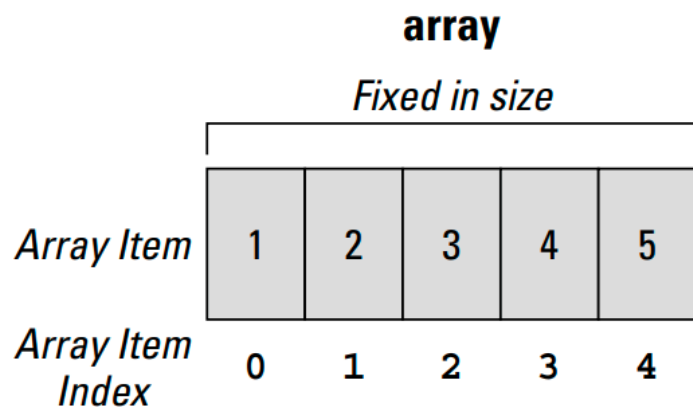
Готово, структура "стек" со всеми нужными методами реализована!

Что такое слайс (slice) и массив (array)? Чем отличается массив от слайса?

В Go массивы и срезы представляют собой структуры данных, состоящие из упорядоченных последовательностей элементов. Эти наборы данных очень удобно использовать, когда вам требуется работать с большим количеством связанных значений. Они позволяют хранить вместе связанные данные, концентрировать код и одновременно применять одни и те же методы и операции к нескольким значениям.

Хотя и массивы, и срезы в Go представляют собой упорядоченные последовательности элементов, между ними имеются существенные отличия.

Массив – это последовательно выделенная область памяти. Частью типа `array` является его размер, который в том числе является не изменяемым.



Массивы представляют собой структурированные наборы данных с заданным количеством элементов. Поскольку массивы имеют фиксированный размер, память для структуры данных нужно выделить только один раз, в то время как для структур данных переменной длины требуется динамическое выделение памяти в большем или меньшем объеме. Хотя из-за фиксированной длины массивов они не отличаются гибкостью в использовании, одноразовое выделение памяти позволяет повысить скорость и производительность вашей программы. В связи с этим, разработчики обычно используют массивы при оптимизации программ, в том числе, когда для структур данных не требуется переменное количество элементов.

```
var numbers [3]int
var strings [3]string
```

```
// Если вы не декларируете значения элементов массива, по умолчанию исп
// т. е. по умолчанию элементы массива будут пустыми.
// Это означает, что целочисленные элементы будут иметь значение 0, а с
fmt.Println(numbers) // [ 0 0 0 ]
fmt.Println(strings) // [ "" "" "" ]
```

Важно помнить, что в каждом случае декларирования нового массива создается отдельный тип. Поэтому, хотя `[2]int` и `[3]int` содержат целочисленные элементы, из-за разницы длины типы данных этих массивов несовместимы друг с другом.

Слайс (срез) – это структура go, которая включает в себя ссылку на базовый массив, а также две переменные `len` (length) и `cap` (capacity).

`len` – это длина слайса, количество элементов, которое в нём сейчас находится,
`cap` – это ёмкость слайса, количество элементов, которые мы можем записать в слайс сверх `len` без его дальнейшего расширения.

```
// структура слайса
type slice struct {
    array unsafe.Pointer
    len    int
    cap    int
}
```

Срез представляет собой мутируемую (изменяемую) упорядоченную последовательность элементов. Поскольку размер срезов не постоянный, а переменный, его использование сопряжено с дополнительной гибкостью. При работе с наборами данных, которые в будущем могут увеличиваться или уменьшаться, использование среза обеспечит отсутствие ошибок при попытке изменения размера набора. В большинстве случаев возможность изменения стоит издержек перераспределения памяти, которое иногда требуется для срезов, в отличие от массивов. Если вам требуется сохранить большое количество элементов или провести итерацию большого количества элементов, и при этом вам нужна возможность быстрого изменения этих элементов, вам подойдет тип данных среза.

```
// Создадим срез, содержащий элементы строкового типа данных:
seaCreatures := []string{"shark", "cuttlefish", "squid", "mantis shrimp"}

// Если вы хотите создать срез определенной длины без заполнения элементов
// вы можете использовать встроенную функцию make()
oceans := make([]string, 3) // output: [ "" "" "" ], len: 3, cap: 3

// Если вы хотите заранее выделить определенный объем памяти, вы можете
oceans := make([]string, 3, 5) // output: [ "" "" "" ], len: 3, cap: 5
```

Как ведут себя срезы в Go на граничных значениях?

Срез может быть создан с использованием выражения `a[low : high]`, где `a` — массив или другой срез, `low` — начальный индекс, а `high` — конечный индекс (не включительно). Если `low` равно 0, его можно опустить. Если `high` равно длине массива, его также можно опустить.

Значения `low` и `high` должны удовлетворять условиям $0 \leq low \leq high \leq \text{cap}(a)$, где `cap(a)` — это емкость исходного массива или среза. Попытка использовать индексы за пределами этих границ приведет к панике.

Если `low` и `high` равны, срез будет пустым, но валидным. Например, `a[2:2]` создаст пустой срез.

Если `low` или `high` выходят за границы допустимых значений, компилятор выдаст панику. Например, если `len(a)` равно 5, то `a[0:6]` вызовет панику, так как 6 превышает допустимую границу.

Срезы в Go являются ссылками на исходный массив. Это означает, что изменения в срезе отразятся на исходном массиве и на всех других срезах, сделанных из этого массива.

Срез, который не был инициализирован, имеет значение `nil`. Он отличается от пустого среза, который был инициализирован, но не содержит элементов. `nil` срез имеет длину и емкость 0, но пустой срез может иметь ненулевую емкость.

Если при добавлении элементов в срез его емкость оказывается недостаточной, Go автоматически создаст новый массив с большей емкостью и скопирует в него элементы из исходного среза.

Вот что можно сказать о поведении срезов в Go на граничных условиях.

Как работает append для слайсов? Можно ли применить к массивам? Напиши свою функцию append.

Функция принимает на вход слайс и переменное количество элементов для добавления в слайс.

`append` расширяет слайс за пределы его `len`, возвращая при этом новый слайс.

```
// функция append
func append(slice []Type, elems ...Type) []Type
```

Если количество элементов, которые мы добавляем в слайс, не будет превышать `cap`, вернется новый слайс, который ссылается на тот же базовый массив, что и предыдущий слайс. Если количество добавляемых элементов превысит `cap`, то вернется новый слайс, базовым для которого будет новый массив.

Пощупаем, как работает `append`.

```
// создаем слайс с capacity равным 3 и длиной 0
slice := make([]int, 0, 3) // len: 0, cap: 3

// далее заполняем слайс тремя элементами
slice = append(slice, 1) // len: 1, cap: 3
slice = append(slice, 2, 3) // len: 3, cap: 3

// получаем ожидаемый результат
fmt.Println(slice) // output [ 1, 2, 3 ]

// окей, теперь попробуем присводить слайс другому слайсу
// помним то, что слайс является структурой из трех элементов len, cap и
// поэтому в sliceCopy мы получаем скопированные значение len и cap, а
sliceCopy := slice
```

```
// пробуем менять первый элемент в новом слайсе
sliceCopy[1] = 10

// убеждаемся, что в обоих слайсах изменились значения, все из-за базового массива
fmt.Println(slice, sliceCopy) // output: slice: [ 1, 10, 3 ] sliceCopy: [ 1, 10, 3 ]

// хорошо, теперь пробуем добавить новый элемент в первый слайс
slice = append(slice, 4)
// тут у нас функция append "видит", что мест больше нет и увеличивает емкость
// и создает новый базовый массив с вместимостью в 6 элементов, что и видно
fmt.Println(slice) // output: [ 1, 10, 3, 4 ] len: 4, cap: 6
// но что случилось тут? ничего, просто ничего, теперь первая переменная sliceCopy
// все еще ссылается на старый массив
fmt.Println(sliceCopy) // output: [ 1, 2, 3 ] len: 3, cap: 3

// точно не связаны? ну давай убедимся! пробуем менять значения первых элементов
sliceCopy[0] = 50
slice[0] = 80

// убедились? :)
fmt.Println(slice, sliceCopy) // output: slice: [ 80, 10, 3, 4 ] sliceCopy: [ 50, 10, 3 ]
```

А вот с массивами функцию append использовать нельзя иначе получим ошибку: first argument to append must be slice; have T

```
array := [3]int{}
array = append(array, 3) // first argument to append must be a slice; have int
```

Теперь напишем свою функцию, тут всё в целом просто и понятно:

```
// она будет проще, только с добавлением одного элемента
func main() {
    fmt.Println(Append([]int{1, 2, 3}, 4))
}

func Append[T any](dst []T, el T) []T {
```

```
var res []T

resLen := len(dst) + 1
if resLen <= cap(dst) {
    res = dst[:resLen]
} else {
    resCap := resLen
    if resCap < 2*len(dst) {
        resCap = 2 * len(dst)
    }

    res = make([]T, resLen, resCap)
    copy(res, dst)
}

res[len(dst)] = el
return res
}
```

Как можно добавить элементы в слайс? Что будет если элемент не вмещается в размер слайса?

Один из способов добавления элементов в слайс мы уже обсудили выше, с использованием функции `append` :

```
slice := make([]int, 0, 10) // len: 0, cap: 10
for i := 0; i < 10; i++ {
    slice = append(slice, i*2)
}
```

Есть еще один способ — через индексы, выглядит это так:

```
slice := make([]int, 10) // len: 10, cap: 10
for i := 0; i < 10; i++ {
    slice[i] = i*2
}
```

У последнего способа есть недостаток, если количество элементов, которые мы хотим добавить в слайс превысит емкость исходного слайса, тогда мы получим панику:

```
panic: runtime error: index out of range [10] with length 10
```

```
// достаточно поменять условие на <=
slice := make([]int, 10) // len: 10, cap: 10
for i := 0; i <= 10; i++ {
    slice[i] = i * 2
}
```

в то время `append` расширил бы базовый массив слайса и продолжил дальше работать не паникуя.

Как можно скопировать слайс? Что такое функция `copy`? Как добиться аналогичного поведения `copy` с помощью `append`?

Встроенная функция `copy` копирует элементы в целевой срез `dst` из исходного среза `src`.

```
func copy(dst, src []Type) int
```

Возвращает количество скопированных элементов, которое будет минимумом `len(dst)` и `len(src)`. Результат не зависит от того, перекрываются ли аргументы.

```
// Копировать из одного среза в другой
var slice = make([]int, 3)
num := copy(slice, []int{0, 1, 2, 3})

fmt.Println(num, slice) // output: num == 3, slice == []int{0, 1, 2}
```

Второй способ копирования слайсов — использовать функцию `append`

```
slice := make([]byte, 0, len(a))
slice = append(c, []int{0, 1, 2, 3}...)

fmt.Println(slice) // output: slice == []int{0, 1, 2}
```

Как можно нарезать слайс?

Какие есть нюансы, подводные камни?

В Go можно сделать подслайс (сорри, кто как называет) из слайса или массива.

Делается так:

```
slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
subSlice := slice[3:8] // [ 4, 5, 6, 7, 8 ]
```

Окей, а что будет, если мы изменим значение под слайса или ещё хуже, добавим туда элементы через функцию `append` ?

```
subSlice[0] = 101
```



```
fmt.Println(slice) // [1 2 3 101 5 6 7 8 9 10]
fmt.Println(subSlice) // [101 5 6 7 8]
```

Видим, что в базовом слайсе тоже поменялись значения, а все потому, что у под слайса все тот же базовый массив, а для подслайса нулевой элемент это элемент под индексом 3 в базовом. Примерно такое же поведение наблюдается у функции `append`, если его применить к под слайсу базового слайса:

```
slice := make([]int, 10, 25)
subSlice := slice[3:5] // [ 0, 0, 0, 0, 0 ]

fmt.Println(len(slice), cap(slice)) // 10 25
fmt.Println(len(subSlice), cap(subSlice)) // 2 22

subSlice = append(subSlice, 11)

fmt.Println(slice) // [0 0 0 0 0 11 0 0 0 0]
fmt.Println(subSlice) // [0 0 11]
```

Причина данного поведения в том, что у обоих слайсов один базовый массив, а так же у под слайса своя "копия" слайса с полями `len` и `cap` и когда мы пытаемся добавить в дочерний слайс элемент, при условии, что в родительском хватает ёмкости, мы просто перезаписываем значение в базовом массива.

Что такое table-driven тесты и как их реализовать в Go?

Table-driven тесты в Go — это метод написания тестов, при котором тестовые кейсы организованы в виде таблицы данных.

Каждая строка таблицы представляет отдельный тестовый кейс с входными данными и ожидаемым результатом. Этот подход позволяет легко добавлять новые тестовые кейсы без необходимости дублирования кода.

Для реализации table-driven тестов в Go обычно используется следующий шаблон:

1. Определяем структуру, которая описывает тестовый кейс, включая входные данные и ожидаемый результат.
2. Создаем срез этих структур, где каждый элемент представляет отдельный тестовый кейс.
3. Используем цикл `for` для итерации по срезу тестовых кейсов.
4. Внутри цикла вызываем функцию, которую тестируем, и сравниваем результат с ожидаемым значением.

На практике выглядит это как-то так:

```
package mypackage

import "testing"

func TestMyFunction(t *testing.T) {
    cases := []struct {
        name string
        input int
        want int
    }{
        {"case1", 1, 2},
        {"case2", 2, 4},
        // ...
    }

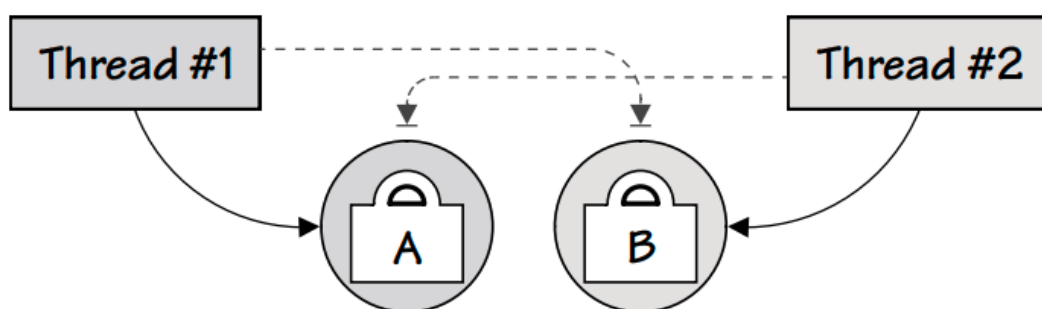
    for _, c := range cases {
        t.Run(c.name, func(t *testing.T) {
            got := MyFunction(c.input)
            if got != c.want {
                t.Errorf("MyFunction(%d) == %d, want %d", c.input, got,
                )
            }
        })
    }
}
```

`MyFunction` — это функция, которую мы тестируем. Для каждого тестового кейса в срезе `cases` мы запускаем тест, используя `t.Run`, что также обеспечивает хорошую

организацию вывода тестов и их независимость.

В каких случаях в Go могут возникнуть deadlocks?

Deadlock — взаимоблокировка потоков:



Причины возникновения дедлоков в Go:

1. **Горутины, ожидающие друг друга:** горутины могут входить в состояние дедлока, если они ожидают ресурсы или сигналы друг от друга, образуя циклическую зависимость.
2. **Неправильное использование каналов:** попытка чтения из закрытого канала или блокировка на отправке/получении данных из-за отсутствия получателей/отправителей, может привести к дедлоку.
3. **Злоупотребление блокировками:** использование мьютексов и других примитивов синхронизации без должной осторожности может вызвать дедлоки. Например, попытка захватить мьютекс, который уже захвачен текущей горутиной, приведет к блокировке.

Что такое горутина? Как ее остановить?

Горутина — это функция или метод, которые выполняются *конкурентно* с любыми другими горутинами, используя специальный поток. Потоки горутин более легковесны, чем стандартные потоки, и большинство программ Go одновременно

используют тысячи горутин. Для создания горутины перед объявлением функции нужно добавить ключевое слово `go`.

```
go f(x, y, z)
```

Остановить горутины можно отправкой сигнала в специальный канал. При этом горутин могут отвечать на такие сигналы, только если им сказано выполнять проверку. Поэтому нужно будет включить проверки в подходящие места, например в начало цикла `for`.

```
package main

func main() {
    quit := make(chan bool)
    go func() {
        for {
            select {
            case <-quit:
                return
            default:
                // ...
            }
        }
    }()

    // ...
    quit <- true
}
```

Все функции, которые могут остановить горутины:

```
runtime.Gosched
runtime.gopark
runtime.notesleep
runtime.Goexit
```

Как завершить много горутин?

Горутины автоматически завершают работу при выходе потока управления из функции `main()`, поэтому важно дождаться окончания выполнения потоков. Для этого можно использовать функции `Sleep()` и `Scanln()`. Однако трудно назвать эти способы универсальными. Какое время указать внутри функции `Sleep()`, чтобы потоки успели полностью выполниться и при этом пользователю не пришлось ждать, если потоки завершились раньше этого времени? Угадать это значение невозможно. Одним из способов решения проблемы является использование структуры `WaitGroup` из пакета `sync`.

Структура `WaitGroup` содержит такие методы:

- `Add()` — добавляет указанное количество потоков к существующему значению счетчика. Значение может быть отрицательным. Если счетчик будет иметь отрицательное значение, то генерируется паника. Формат `Add()` :

```
(*sync.WaitGroup).Add(delta int)
```

- `Done()` — уменьшает значение счетчика на единицу. Этот метод следует вызывать при завершении потока. Формат `Done()` :

```
(*sync.WaitGroup).Done()
```

- `Wait()` — блокирует выполнение потока до тех пор, пока значение счетчика не станет равно нулю. Формат `Wait()` :

```
(*sync.WaitGroup).Wait()
```

А вот пример использования `WaitGroup` :

```
package main
```

```
import (  
    "fmt"
```

```
"sync"
"time"
)

func main() {
    fmt.Println("Начало функции main()")
    var wg sync.WaitGroup
    for i := 1; i < 4; i++ {
        wg.Add(1)           // Увеличиваем счетчик потоков на единицу
        go func(n int) {
            defer wg.Done() // Уменьшаем счетчик потоков на единицу
            for j := 1; j < 11; j++ {
                fmt.Println("Поток:", n, "j =", j)
                time.Sleep(time.Second) // Имитация выполнения задачи
            }
        }(i)
    }
    wg.Wait() // Ожидаем завершения всех потоков
    fmt.Println("Конец функции main()")
}
```

В чём различия горутины от потока системы?

Уровень абстракции:

- Горутины — это абстракции уровня языка, предоставляемые Go. Они позволяют выполнять функции или методы конкурентно.
- Потоки — это более традиционные сущности операционной системы для параллельного выполнения задач.

Размер стека:

- Горутины начинаются с очень маленького стека, который может динамически расти и сокращаться в зависимости от потребности (обычно начинается с 2KB).

- Потоки, в зависимости от ОС, обычно имеют стек фиксированного размера, который может быть значительно больше (обычно от 1MB и выше).

Создание и переключение:

- Горутины легко создать (просто используя ключевое слово `go` перед вызовом функции), и они дешевы в плане создания и переключения контекста.
- Потоки дороже по стоимости создания и контекстного переключения, так как это требует прямого взаимодействия с операционной системой.

Планировщик:

- Горутины управляются планировщиком Go, который работает в пользовательском пространстве (`user space`) и распределяет горутины по доступным ОС потокам (обычно один поток на ядро CPU).
- Потоки управляются планировщиком ОС.

Изоляция:

- Ошибка в одной горутине (например, паника) может повлиять на все другие горутины в той же программе.
- Ошибка в одном потоке (например, `segmentation fault`) обычно не влияет на другие потоки.

Ниже представлен пример создания горутины и потока в Go. Мы создаем горутину с помощью ключевого слова. Затем, меняя `GOMAXPROCS` , мы фактически заставляем Go использовать дополнительный поток ОС, что делает выполнение кода более похожим на многопоточное:

```
package main

import (
    "fmt"
    "runtime"
    "sync"
    "time"
)

func runGoroutine(id int) {
```

```
fmt.Println("Горутина", id)
}

func main() {
    // создаём горутину
    go runGoroutine(1)

    // создаём поток, установив максимальное количество используемых по-
    runtime.GOMAXPROCS(2)
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println("Поток (после изменения GOMAXPROCS)")
    }()
    wg.Wait()

    time.Sleep(1 * time.Second) // даём горутине время на выполнение
}
```

Реализовать функцию reverse, разворачивающую срез целых чисел без использования временного среза

Меньше слов, поехали сразу к коду:

```
package main
import "fmt"

func reverse(sw []int) {
    for a, b := 0, len(sw)-1; a < b; a, b = a+1, b-1 {
        sw[a], sw[b] = sw[b], sw[a]
    }
}
```



```
    }  
}  
  
func main() {  
    x := []int{3, 2, 1}  
    reverse(x)  
    fmt.Println(x)  
}
```

Цикл меняет местами значения каждого элемента среза. Значения будут следовать слева направо, и в итоге все элементы будут развернуты.

Что такое пакеты в Go?

В Go, **пакет** — это коллекция исходных файлов `.go` в одной директории и с одинаковой директивой `package`. В начале каждого такого файла объявляется зарезервированное слово `package`, а после него прописывается имя пакета. В рамках пакета все функции и глобальные переменные, объявленные как в верхнем, так и в нижнем регистре, видят друг друга.

Каждая программа на Go состоит из пакетов. Файлы, находящиеся в одном каталоге, должны относиться к одному пакету.

Пакеты в Go можно сравнить с неймспейсами в других языках. Все программы, написанные на Go, начинают работу в пакете `main`. В первой строке каждого файла `.go` используется оператор `package`, указывающий к какому пакету относится код данного файла.

Согласно соглашению, имя пакета совпадает с последним элементом пути импорта. Например, пакет `math/rand` состоит из файлов, которые начинаются с оператора `package rand`.

Импорт пакетов осуществляется с помощью оператора `import` и позволяет нам вызывать функции, которые не встроены в Go. Некоторые пакеты являются частью стандартной библиотеки Go, а некоторые нужно устанавливать с помощью инструмента `go get`.

Что такое глобальная переменная?

Глобальная переменная – это переменная уровня пакета, то есть объявленная вне функции. Глобальная переменная также может быть доступна за рамками пакета, конечно только в том случае, если ее наименование начинается в верхнем регистре.

К глобальным переменным можно получить доступ из любого места пакета, не прибегая к необходимости явно передавать их в функцию, и они могут меняться, если только не были определены как константы с использованием ключевого слова `const`.

Важный факт о названии глобальной переменной. Регистр первой буквы в названии глобальных переменных, функций и структур имеет специальное значение. Если первая буква названия является строчной, то такой идентификатор будет виден только в пределах пакета, внутри которого он объявлен. Если первая буква названия является заглавной, то такой идентификатор будет общедоступным, поэтому все названия функций, которые мы уже рассматривали, начинаются с заглавной буквы.

Область видимость глобальных и локальных переменных можно проиллюстрировать на этом коде:

```
package main

import "fmt"

var x int = 10                                // Глобальная переменная

func main() {
    test()
    // Вывод значения глобальной переменной x
    fmt.Println(x)                            // 10
    { // Блок
        z := 30                                // Локальная переменная
        fmt.Println(z)                        // 30
    }
    // Переменная z здесь уже не видна!!!
}
```

```
    for i := 0; i < 10; i++ {  
        fmt.Println(i)                // 30  
    }  
    // Переменная i здесь уже не видна!!!  
}  
  
func test() {  
    var x int = 5                      // Локальная переменная  
    // Вывод значения локальной переменной x  
    fmt.Println(x)                    // 5  
}
```

Реализовать алгоритм бинарного поиска

Задача: реализовать алгоритм бинарного поиска. Также известен как метод деления пополам или дихотомия – классический алгоритм поиска элемента в отсортированном массиве (слайсе), использующий дробление массива (слайса) на половины. На входе может быть слайс вида `[]int{1, 3, 4, 6, 8, 10, 55, 56, 59, 70, 79, 81, 91, 10001}`

Вернуть: индекс элемента 55 (то есть вернуть 6)

Принцип алгоритма бинарного поиска очень прост. Мы всегда имеем дело с упорядоченным массивом. Это позволяет прибегать к хитрости – на каждой итерации цикла, который мы запускаем, мы вычисляем индекс среднего элемента. Этот элемент сравниваем с искомым.

Если элемент из середины массива оказался меньше, чем тот, что мы ищем, значит нужный нам находится правее по массиву, ведь массив упорядочен. Соответственно, нам нужно перейти в следующую итерацию цикла, “оставляя” ей лишь правую часть массива – в ней снова будет найден средний элемент и алгоритм повторится.

Если же элемент из середины массива оказался больше искомого, то мы переходим к следующей итерации, отбрасывая правую часть массива, а оставляя левую.

Если же элемент совпадает с искомым, мы выходим из цикла.

```
package main


func BinarySearch(in []int, searchFor int) (int, bool) {
    if len(in) == 0 {
        return 0, false
    }

    var first, last = 0, len(in) - 1

    for first <= last {
        var mid = ((last - first) / 2) + first

        if in[mid] == searchFor {
            return mid, true
        } else if in[mid] > searchFor { // нужно искать в "левой" части слайда
            last = mid - 1
        } else if in[mid] < searchFor { // нужно искать в "правой" части слайда
            first = mid + 1
        }
    }

    return 0, false
}
```



Конечно, вместо цикла мы могли бы использовать рекурсию.

****Что выведет этот код?****

```
package main

import (
    "fmt"
)

func main() {
```

```
test1 := []int{1, 2, 3, 4, 5}
test1 = test1[:3]
test2 := test1[3:]
fmt.Println(test2[:2])
}
```

Что ж, обсудим, что тут происходит.

1. импортируем пакет `fmt`
2. определяем функцию `main`
3. создаем массив целых чисел `test1` со значениями `[1, 2, 3, 4, 5]`
4. берём из `test1` только первые 3 элемента вот так: `test1 = test1[:3]`
5. создаем новый срез `test2`, нарезая `test1` с индекса 3
6. печатаем первые два элемента `test2` — это `[4 5]`

Вот собственно и всё.

Что ты можешь сказать про структуру `Reader`?

Структура `Reader` описывает байтовый буфер доступный только для чтения и реализует все методы из интерфейсов `io.Reader`, `io.ReaderAt`, `io.WriterTo`, `io.Seeker`, `io.ByteScanner` и `io.RuneScanner`. Это означает, что мы можем передать буфер везде, где ожидаются эти интерфейсы, например, в качестве потока для ввода данных.

Создать объект буфера позволяет функция `NewReader()`. Формат функции:

```
bytes.NewReader(b []byte) *bytes.Reader
```

Пример:

```
// import "os"
buf := bytes.NewReader([]byte("test"))
buf.WriteTo(os.Stdout) // test
```

Reader. Получение содержимого буфера

Получить содержимое буфера позволяют следующие методы:

- `Read()` — при каждом вызове записывает в слайс `b` следующие `len(b)` байтов. Метод возвращает два значения. Через `n` доступно число считанных байтов. Через `err` можно получить информацию об ошибке. Если в буфере больше нет данных, то значением `err` будет `io.EOF`. Если ошибки не возникло, то значением `err` будет `nil`. Формат метода:

```
(*bytes.Reader).Read(b []byte) (n int, err error)
```

Пример с `Read()` :

```
arr := []byte{0, 0}
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.Read(arr)) // 2 <nil>
fmt.Println(arr)           // [116 101]
fmt.Println(buf.Read(arr)) // 2 <nil>
fmt.Println(arr)           // [115 116]
fmt.Println(buf.Read(arr)) // 0 EOF
```

- `ReadAt()` — записывает в слайс `b` `len(b)` байтов, начиная с позиции `off`. Метод возвращает два значения. Через `n` доступно число считанных байтов. Через `err` можно получить информацию об ошибке. Если в буфере больше нет данных, то значением `err` будет `io.EOF`. Если ошибки не возникло, то значением `err` будет `nil`. Формат метода:

```
(*bytes.Reader).ReadAt(b []byte, off int64) (n int, err error)
```

Пример с `ReadAt()` :

```
arr := []byte{0, 0}
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.ReadAt(arr, 0)) // 2 <nil>
fmt.Println(arr)                // [116 101]
fmt.Println(buf.ReadAt(arr, 2)) // 2 <nil>
fmt.Println(arr)                // [115 116]
fmt.Println(buf.ReadAt(arr, 4)) // 0 EOF
```

- `ReadByte()` — при каждом вызове через первое возвращаемое значение доступен следующий байт. Через второе возвращаемое значение можно получить информацию об ошибке. Если в буфере больше нет данных, то значением будет `io.EOF`. Если ошибки не возникло, то значением будет `nil`.
Формат метода:

```
(*bytes.Reader).ReadByte() (byte, error)
```

Пример с `ReadByte()` :

```
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.ReadByte()) // 116 <nil>
fmt.Println(buf.ReadByte()) // 101 <nil>
fmt.Println(buf.ReadByte()) // 115 <nil>
fmt.Println(buf.ReadByte()) // 116 <nil>
fmt.Println(buf.ReadByte()) // 0 EOF
```

- `UnreadByte()` — отменяет чтение последнего байта. Формат метода:

```
(*bytes.Reader).UnreadByte() error
```

Пример `UnreadByte()` :

```
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.ReadByte()) // 116 <nil>
```

```
fmt.Println(buf.UnreadByte()) // <nil>
fmt.Println(buf.ReadByte())    // 116 <nil>
```

- `ReadRune()` — при каждом вызове через первое возвращаемое значение доступен следующий символ. Через второе возвращаемое значение доступно число байтов. Через третье возвращаемое значение можно получить информацию об ошибке. Если в буфере больше нет данных, то значением будет `io.EOF`. Если ошибки не возникло, то значением будет `nil`. Формат метода:

```
(*bytes.Reader).ReadRune() (ch rune, size int, err error)
```

Пример с `ReadRune()` :

```
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.ReadRune()) // 116 1 <nil>
fmt.Println(buf.ReadRune()) // 101 1 <nil>
fmt.Println(buf.ReadRune()) // 115 1 <nil>
fmt.Println(buf.ReadRune()) // 116 1 <nil>
fmt.Println(buf.ReadRune()) // 0 0 EOF
```

- `UnreadRune()` — отменяет чтение последнего символа. Формат метода:

```
(*bytes.Reader).UnreadRune() error
```

Пример с `UnreadRune()` :

```
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.ReadRune()) // 116 1 <nil>
fmt.Println(buf.UnreadRune()) // <nil>
fmt.Println(buf.ReadRune()) // 116 1 <nil>
```

- `Seek()` — позволяет задать позицию указателя внутри буфера. Формат метода:

```
(*bytes.Reader).Seek(offset int64, whence int) (int64, error)
```


Пример с Seek :

```
arr := []byte{0, 0}
buf := bytes.NewReader([]byte("test"))
fmt.Println(buf.Read(arr)) // 2 <nil>
fmt.Println(arr)           // [116 101]
// Перемещаем указатель в начало буфера
fmt.Println(buf.Seek(0, 0)) // 0 <nil>
fmt.Println(buf.Read(arr))  // 2 <nil>
fmt.Println(arr)           // [116 101]
```

Прочитать содержимое буфера позволяют также функции `Fscan()` , `Fscanln()` и `Fscanf()` из пакета `fmt`

```
buf := bytes.NewReader([]byte("10 20"))
x, y := 0, 0
n, err := fmt.Fscanf(buf, "%d %d", &x, &y)
fmt.Println(n, err) // 2 <nil>
fmt.Println(x, y)   // 10 20
```

Как реализована тар в Go?

Представление хеш-таблицы: `map[string]int`

Массив

0	→	Ключ	Значение
1		"two"	2
2		"six"	6
3			

`map` представляет собой неупорядоченную коллекцию пар ключ-значение, в которой все ключи различны. Под капотом `map` основана на структуре данных хеш-таблицы, которая в свою очередь представляет собой массив *бакетов*, где каждый бакет — это указатель на массив пар ключ-значение.

Как создать `map`?

- с помощью ключевого слова `map` с последующим указанием типа данных ключа в квадратных скобках `[]` и типа данных значения. Пары ключ-значение заключаются в фигурные скобки `{ }`: `map[key]value{ }`
- функция `make` представляет альтернативный вариант создания `map`. Она создает пустую хеш-таблицу: `m := make(map[string]int)`

Что будет, если попытаться получить значение по несуществующему ключу из `map`?

- мы получим нулевое значение для типа значений `map`. Например, если это `map[string]int`, то значение будет `0`. Если это `map[string]*SomeStruct`, значение будет `nil`.

Как проверить, существует ли ключ в `map`?

- при получении значения из `map` можно использовать второй возвращаемый аргумент, который будет булевым значением, указывающим, существует ли ключ: `value, exists := m["key"]`

Является ли `map` потокобезопасным типом данных?

- нет, `map` не является потокобезопасным, и для доступа к нему из нескольких горутин одновременно может потребоваться синхронизация, например, с помощью `sync.Mutex`.

Немного об оптимизации: если мы заранее знаем количество элементов, которые будет содержать `map`, эффективнее будет создать ее, указав начальный размер. Это позволяет избежать потенциального расширения `map`, что довольно сложно с точки зрения вычислений, поскольку требует перераспределения достаточного пространства памяти и перебалансировки всех элементов.

Интересный вопрос: если ключ или значение типа `map` имеют размер более 128 байт, каким образом Go их будет хранить?

- Если ключ или значение карты превышает 128 байт, Go не сохранит его непосредственно в бакете карты. Вместо этого Go сохраняет указатель на ключ или значение.

Что следует учитывать при добавлении элемента в карту во время итерации, чтобы избежать недетерминированных результатов?

В примере ниже проводятся итерации по `map[int]bool`. Если значение пары равно `true`, мы добавляем еще один элемент.

```
m := map[int]bool {
    0: true,
    1: false,
    2: true, }
for k, v := range m {
    if v {
        m[10+k] = true
    }
}
fmt.Println(m)
```

Результат непредсказуем:

```
map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true]
map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true 32:true]
map[0:true 1:false 2:true 10:true 12:true 20:true]
```



Вот что говорится в спецификации Go по поводу создания нового элемента карты во время итераций:

Если во время итерации создается элемент карты, он может быть обработан во время итерации или пропущен. Выбор может *варьироваться* для каждого созданного элемента и от одной итерации к другой.

Когда элемент добавляется к карте во время итерации, он может быть либо создан, либо нет при последующей итерации. В Go нет возможности как-то «навязать» поведение кода. Оно может *варьироваться* от одной итерации к другой, и поэтому мы трижды получали разные результаты.

Важно помнить о таком поведении, чтобы код не выдавал непредсказуемых результатов. Если нужно обновить карту во время итерации по ней, то одним из решений будет работа с *копией* карты:

```
m := map[int]bool{
    0: true,
    1: false,
```

```
    2: true,
}
m2 := copyMap(m) // Создается копия первоначальной карты

for k, v := range m {
    m2[k] = v
    if v {
        m2[10+k] = true // Обновляется m2 вместо m
    }
}
fmt.Println(m2)
```

В этом примере мы отделяем *читаемую* карту от *обновляемой*. Мы продолжаем итерировать по `m`, но все обновления делаются на `m2`. Новая версия кода ведет к предсказуемому и повторяемому результату:

```
map[0:true 1:false 2:true 10:true 12:true]
```

В общем, при работе с картой не следует полагаться:

- на то, что данные упорядочиваются по ключам
- на то, что порядок вставки сохранится
- на детерминированность порядка итераций
- на то, что элемент будет создан во время той же итерации, во время которой он был добавлен

Что важно помнить при использовании карты типа `any`?

При демаршалинге (десериализация, JSON → структуры Go) данных мы можем иметь дело с картой вместо структуры. Когда ключи и значения не определены, работа с картой, а не со статической структурой, дает некоторую гибкость. Но есть правило, о котором следует помнить, чтобы избежать неверных предположений и возможной паники.

Возьмем простой пример:

```
b := getMessage()
var m map[string]any
err := json.Unmarshal(b, &m)
if err != nil {
    return err
}
```

Добавим следующий JSON:

```
{
    "id": 32,
    "name": "foo"
}
```

Поскольку мы используем общую мапу `map[string]any`, она автоматически парсит все поля: `map[id:32 name:foo]`

При использовании мапы типа `any` важно помнить:

- любое числовое значение, независимо от того, содержит оно десятичное число или нет, преобразуется в тип `float64`.

Выведем тип `m["id"]` и убедимся в этом:

```
fmt.Printf("%Tn", m["id"])

// float64
```

Важно: не делать ошибочных предположений и не ожидать, что числовые значения без десятичных знаков будут по умолчанию преобразованы в целые числа.

Что такое data race (гонка данных) в Go?

Гонки данных — одни из наиболее распространенных и самых сложных для отладки типов ошибок в конкурентных системах. Гонка данных возникает, когда две горуты одновременно обращаются к одной и той же переменной, и хотя бы одно из обращений — запись. Чтобы избежать эту проблемы, в Go предоставляются различные примитивы синхронизации.

Race Condition (состояние гонки) — более широкое понятие, чем гонка данных. Оно описывает ситуацию, когда поведение программы зависит от относительного порядка выполнения операций. Гонка данных — один из видов состояний гонки, но не единственный.

Пример гонки данных, которая может привести к сбоям и повреждению памяти:

```
func main() {
    c := make(chan bool)
    m := make(map[string]string)
    go func() {
        m["1"] = "a" // Первый конфликтный доступ
        c <- true
    }()
    m["2"] = "b" // Второй конфликтный доступ
    <-c
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

Чтобы помочь диагностировать такие ошибки, Go включает встроенный детектор гонок данных. Для его использования добавьте флаг `-race` в команду `go` :

```
$ go test -race mypkg
$ go run -race mysrc.go
$ go build -race mycmd
$ go install -race mypkg
```

Переменная окружения `GORACE` устанавливает параметры детектора гонок данных, например:

```
$ GORACE="log_path=/tmp/race/report Strip_path_prefix=/my/go/sources/" {
```

Не могу порекомендовать ничего лучше по теме, чем [официальные доки Go](#)

Вывести все комбинации символов строки

Нужно: реализовать функцию `perm()`, принимающую срез или строку и выводящую все возможные комбинации символов.

Решение может быть таким:

```
package main
import "fmt"

// Perm вызывает f с каждой пермутацией a.
func Perm(a []rune, f func([]rune)) {
    perm(a, f, 0)
}

// пермутируем значения в индексе i на len(a)-1.
func perm(a []rune, f func([]rune), i int) {
    if i > len(a) {
        f(a)
        return
    }
    perm(a, f, i+1)
    for j := i + 1; j < len(a); j++ {
        a[i], a[j] = a[j], a[i]
        perm(a, f, i+1)
        a[i], a[j] = a[j], a[i]
    }
}
```



```
}  
}  
  
func main() {  
    Perm([]rune("abc"), func(a []rune) {  
        fmt.Println(string(a))  
    })  
}
```

Мы используем типы `rune` для обработки и срезов, и строк. `runes` являются кодовыми точками из Unicode, а значит могут парсить строки и срезы одинаково.

Как можно оптимизировать использование памяти в Go, особенно при работе с большими структурами данных?

Для оптимизации использования памяти в Go необходимо выполнять некоторые рекомендации, в частности:

1. **Избегать глобальных переменных:** глобальные переменные остаются в памяти на протяжении всего времени выполнения программы. Используйте их, когда это действительно необходимо.
2. **Использовать правильные типы данных:** например, вместо использования `int` для небольших чисел можно использовать `int8` / `int16` и т. д., в зависимости от диапазона значений.
3. `sync.Pool` : если в программе часто создаются и удаляются большие объекты, мы можем использовать `sync.Pool` для их повторного использования.
4. **Ленивая инициализация:** инициализировать сложные структуры данных или большие массивы желательно только тогда, когда они действительно нужны.

5. **Использовать указатели на структуры:** вместо передачи копии структуры мы можем передать указатель на нее. Важно знать, что это правило работает не всегда и не везде (подробнее можно прочитать [здесь](#)).
6. **Срезы vs массивы:** срезы могут менять свой размер и динамически выделять память. Если размер данных известен, лучше использовать массив.
7. **Освобождать ресурсы:** временные большие структуры данных, которые больше не нужны, следует явно освобождать, присваивая им значение `nil`, чтобы сборщик мусора мог быстрее их убрать.
8. **Использовать буферизацию:** буферизированный ввод/вывод или буферизированные каналы могут сократить количество выделений и освобождений памяти.
9. **Оптимизировать структуры:** структуры в Go выровнены по памяти. Переупорядочивание полей структуры может уменьшить ее размер.

В целом, советы очевидны и просты, но если бы все им следовали — жизнь была бы прекрасней)

Что такое интерфейсы в Go?

Интерфейсы в Go предоставляют способ указания поведения объекта.

1. **Определение интерфейса.** Интерфейс в Go представляет собой набор методов, для которых не указаны конкретные реализации:

```
type Writer interface {  
    Write([]byte) (int, error)  
}
```

1. **Реализация интерфейса.** Если определенный тип предоставляет методы, соответствующие всем методам интерфейса, считается, что этот тип реализует данный интерфейс (та самая утиная типизация). В Go не требуется явно указывать, что тип реализует интерфейс — это определяется неявно.
2. **Пустой интерфейс.** Интерфейс без методов называется пустым и записывается как `interface{}`. Любой тип удовлетворяет пустому интерфейсу, что делает его полезным для создания универсальных функций и структур.

3. **Встраивание интерфейсов.** Можно комбинировать несколько интерфейсов, встраивая один интерфейс в другой:

```
type ReaderWriter interface {  
    Reader  
    Writer  
}
```

1. **Интерфейсы и методы со значениями и указателями.** Методы, определенные с получателем-указателем, могут быть частью интерфейса только если используется указатель на тип. Это важно учитывать при проектировании интерфейсов.
2. **Использование интерфейсов.** Интерфейсы позволяют задавать требования к поведению типов, обеспечивая полиморфное поведение. Таким образом, функции могут принимать параметры интерфейсного типа, что дает большую гибкость при работе с различными типами.
3. **type assertion и type switch .** При работе с интерфейсами иногда требуется приведение типов или определение конкретного типа значения интерфейса. Для этих задач используются операции `type assertion` и `type switch` .
4. **Значение интерфейса** в Go состоит из двух компонентов: указателя на конкретное значение и указателя на таблицу методов этого типа.
5. `nil` может быть допустимым значением интерфейса. Если интерфейс содержит `nil` и на нем вызывается метод, это вызовет ошибку времени выполнения.

Пощупаем интерфейс на практике. Допустим, мы хотим определить интерфейс для геометрических фигур, которые могут вычислять свою площадь.

- У нас есть интерфейс `Shape` с методом `Area` .
- `Circle` и `Square` — две структуры, которые реализуют этот интерфейс.
- В функции `main` мы создаем экземпляры `Circle` и `Square` , добавляем их в срез `shapes` типа `Shape` , а затем итерируемся по этому срезу, выводя площадь каждой фигуры.

```
package main
```

```
import (
```

```
    "fmt"
    "math"
)

type Shape interface {
    Area() float64
}

type Cicle struct {
    Radius float64
}

func (c Cicle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

type Square struct {
    SideLength float64
}

func (s Square) Area() float64 {
    return s.SideLength * s.SideLength
}

func main() {
    circle := Cicle{Radius: 5}
    square := Square{SideLength: 4}

    shapes := []Shape{circle, square}

    for _, shape := range shapes {
        fmt.Printf("Area of %T: %fn", shape, shape.Area())
    }
}
```

Этот пример показывает силу интерфейсов в Go: они предоставляют общий способ работы с разными типами, имеющими общий функционал.

Как сообщить компилятору Go, что наш тип реализует интерфейс?

В Go интерфейсы реализуются неявно. Это означает, что нам не нужно явно указывать, что наш тип реализует интерфейс (да-да, та самая *утиная типизация*). Вместо этого, если наш тип определяет все методы, которые присутствуют в интерфейсе, то он считается его реализующим.

Простой пример, допустим, у нас есть следующий интерфейс:

```
type Speaker interface {  
    Speak() string  
}
```

и тип `Person` :

```
type Person struct {  
    Name string  
}  
  
func (p Person) Speak() string {  
    return "My name is " + p.Name  
}
```

Так как `Person` определяет метод `Speak()`, который присутствует в интерфейсе `Speaker`, `Person` автоматически реализует интерфейс `Speaker`. Нет необходимости в дополнительном коде или объявлении для подтверждения этого.

На какой стороне описывать интерфейс — на передающей

или принимающей?

Многое зависит от конкретного случая, но по умолчанию описывать интерфейсы следует на **принимающей** стороне — таким образом, ваш код будет меньше зависеть от какого-то другого кода/пакета/реализации.

Другими словами, если нам в каком-то месте требуется “что-то что умеет себя закрывать”, или — умеет метод `Close() error`, или (другими словами) удовлетворяет интерфейсу:

```
type something interface {  
    Close() error  
}
```

...то он (интерфейс) должен быть описан на **принимающей** стороне. Так принимающая сторона не будет ничего знать о том, что именно в неё может “прилететь”, но точно знает поведение этого “чего-то”. Таким образом реализуется инверсия зависимости, и код становится проще переиспользовать/тестировать.

Написать функцию, находящую палиндром

Задача: написать функцию, которая позволяет вернуть значение `true`, если строка является палиндромом, и `false` — если нет.

Палиндром — слово, предложение или последовательность символов, которая абсолютно одинаково читается как в привычном направлении, так и в обратном. Ну и понятно, что “Anna” — это палиндром, а “table” и “John” — нет.

Вариант №1: Сравнение символов

Один из самых простых способов проверки, является ли строка палиндромом, заключается в сравнении символов с начала и конца строки. Если все символы соответствуют, то строка является палиндромом.

```
func IsPalindrome(str string) bool {
    for i := 0; i < len(str)/2; i++ {
        if str[i] != str[len(str)-i-1] {
            return false
        }
    }

    return true
}
```

Вариант №2: Использование функций strings

В Golang есть функция `strings.Reverse`, которая переворачивает строку в обратном порядке. Мы можем сравнить оригинальную строку с перевернутой строкой, чтобы узнать, является ли она палиндромом.

```
import "strings"

func IsPalindrome(str string) bool {
    reversedStr := strings.Builder{}

    for i := len(str) - 1; i >= 0; i-- {
        reversedStr.WriteByte(str[i])
    }

    return str == reversedStr.String()
}
```

Вариант №3: Использование пакета bytes

В Golang есть пакет `bytes`, который предоставляет функцию `bytes.Equal`, которую мы можем использовать для сравнения двух срезов байтов.

```
import "bytes"

func IsPalindrome(str string) bool {
    reversedBytes := make([]byte, len(str))
```

```
for i := 0; i < len(str); i++ {
    reversedBytes[i] = str[len(str)-i-1]
}

return bytes.Equal([]byte(str), reversedBytes)
}
```

Вариант №4: Рекурсия

Еще один способ проверки, является ли строка палиндромом, – использование рекурсии. Если первый и последний символы строки равны, мы рекурсивно вызываем функцию `IsPalindrome` для подстроки без первого и последнего символов.

```
func IsPalindrome(str string) bool {
    if len(str) <= 1 {
        return true
    }

    if str[0] != str[len(str)-1] {
        return false
    }

    return IsPalindrome(str[1 : len(str)-1])
}
```

Зачем используется ключевое слово `defer` в Go?

`defer` в Go — ключевое слово, которое используется для отложенного выполнения функции или метода до тех пор, пока текущая функция не завершится. Когда встречается `defer`, Go добавляет вызов функции или метода в стек отложенных вызовов, а затем продолжает выполнение текущей функции.

При этом, место объявления одной инструкции `defer` в коде никак не влияет на то, когда та выполнится. Функция с `defer` всегда выполняется перед выходом из внешней функции, в которой `defer` объявлялась.

Некоторые применения и особенности `defer` :

- **Заккрытие ресурсов.** Один из самых распространенных примеров использования `defer` — убедиться, что ресурсы, такие как файлы, сетевые подключения или соединения с базой данных, будут закрыты после их использования.

```
file, err := os.Open("file.txt")
if err != nil {
    //обработка ошибки
}
defer file.Close()
```

- **Множественные отложенные вызовы:** мы можем использовать несколько операторов `defer` в одной функции. Они будут выполнены в порядке LIFO.

```
func example() {
    defer fmt.Println("1")
    defer fmt.Println("2")
    fmt.Println("Function body")
}
```

- **Передача аргументов:** аргументы функции, вызываемой с помощью `defer`, вычисляются в момент вызова `defer`, а не в момент выполнения отложенной функции.

```
func example(a int) {
    defer fmt.Println(a)
    a *= 2
    return
}
example(5) //5
```

- **Использование с паникой:** `defer` часто используется совместно с `recover()`, чтобы обрабатывать или логировать панику, которая может произойти в функции.

```
func mightPanic() {  
    defer func() {  
        if r := recover(); r != nil {  
            fmt.Println("Recovered from panic:", r)  
        }  
    }()  
    //код, который может вызвать панику  
}
```

- **Зависимость от контекста:** отложенные функции имеют доступ к локальным переменным и могут изменять их значения, что делает `defer` мощным инструментом для выполнения последних действий с переменными перед выходом из функции.
- **Затраты производительности:** хотя ключевое слово `defer` удобно и безопасно, использование его внутри интенсивных по производительности циклов может вызвать незначительные, но всё же заметные накладные расходы.
- **Порядок возврата при использовании несколько функций с `defer`:**

```
func main() {  
    fmt.Println("counting")  
  
    for i := 1; i < 4; i++ {  
        defer fmt.Println(i)  
    }  
    fmt.Println("done")  
}
```

`defer` добавляет переданную после него функцию в стек. При возврате внешней функции вызываются все добавленные в стек вызовы. Поскольку стек работает по принципу LIFO (last in first out), значения стека возвращаются в порядке от последнего к первому. Таким образом, функции с `defer` будут вызываться **в обратной последовательности** от их объявления во внешней функции.

- **Как передаются значения в функции, перед которыми указано defer?**

```
func main() {  
    nums := 1 << 5 // 32  
  
    defer fmt.Println(nums)  
  
    nums = nums >> 1 //16  
  
    fmt.Println("done")  
}
```

Аргументы функций, перед которыми указано ключевое слово `defer` оцениваются немедленно. То есть на тот момент, когда переданы в функцию.

Что такое замыкания функций?

Во-первых, функции в Go — обычные значения; с ними можно работать, как с любыми другими объектами. А значит их можно даже передавать и возвращать другим функциям.

Во-вторых, функции могут создаваться внутри других как *анонимные функции*, их тоже можно вызывать, передавать или использовать иным способом (анонимная функция — функция, которой не назначено имя)

Особенностью Go является доступность состояния внешней функции из анонимных функций, даже после ее завершения. Именно это позволяет определять *замыкания*.

Замыкание — вложенная функция, сохраняющая доступ к переменным внешней функции даже после завершения последней.

Возьмем функцию `incrementor`. Она имеет состояние в виде переменной `i` и возвращает анонимную функцию, которая увеличивает значение перед возвратом. Можно сказать, что возвращаемая функция «замкнута» на переменной `i`.

```
func incrementor() func() int {  
    i := 0
```

```
    return func() int {  
        i++  
        return i  
    }  
}
```

Вызов `incrementor` создаст свою локальную копию `i` и вернет новую анонимную функцию, увеличивающую значение этой копии. Последующие вызовы `incrementor` будут создавать новые копии `i`:

```
func main() {  
    increment := incrementer()  
    fmt.Println(increment()) // 1  
    fmt.Println(increment()) // 2  
    fmt.Println(increment()) // 3  
  
    newIncrement := incrementer()  
    fmt.Println(newIncrement()) // 1  
}
```

Или такой пример, тут `adder()` возвращает замыкание, привязанное к собственной переменной `sum`, на которую оно ссылается.

```
package main  
import "fmt"  
  
func adder() func(int) int {  
    sum := 0  
    return func(x int) int {  
        sum += x  
        return sum  
    }  
}  
  
func main() {  
    pos, neg := adder(), adder()  
    fmt.Println(pos(1), neg(1))  
}
```

```
for i := 0; i < 10; i++ {  
    fmt.Println(  
        pos(i),  
        neg(-2*i),  
    )  
}  
}
```

Реализовать функцию, подсчитывающую количество гласных

Ну, в целом всё просто:

```
func countVowels(s string) int {  
    count := 0  
    for _, char := range s {  
        switch char {  
            case 'a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U':  
                count++  
            }  
        }  
    }  
    return count  
}
```

Не удержусь, помещу сюда милую питонячую реализацию в 2 строки:

```
def count_vowels(word):  
    return sum([w in 'aAeEiIoOuU' for w in word])
```

Что возвращает функция `len()`, если ей передаётся строка в кодировке UTF-8?

В Go строки на самом деле представляют собой последовательности байтов. Это означает, что когда вы передаёте строку в кодировке UTF-8 функции `len()`, она считает байты, а не символы:

```
func main() {  
    s := "世界"  
    fmt.Println("Байт:", len(s)) // 6 байт  
    fmt.Println("Символов:", utf8.RuneCountInString(s)) // 2 символа  
}
```

Расскажи про работу с ошибками в Go

В Golang ошибки обрабатываются с помощью возврата значений ошибки из функций. Обычно функции, которые могут вернуть ошибку, возвращают два значения: *результат выполнения и значение ошибки*. Если выполнение функции прошло успешно, то значение ошибки равно `nil`. Если же произошла ошибка, то значение ошибки содержит соответствующую информацию.

А вот так можно обрабатывать ошибки:

```
goCopy code  
func divide(x, y int) (result int, err error) {  
    if y == 0 {  
        err = errors.New("division by zero")  
        return  
    }  
    result = x / y  
}
```

```
    return
}

func main() {
    result, err := divide(10, 2)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Result:", result)
}
```

В приведенном примере функция `divide()` возвращает результат деления и ошибку, если делитель равен 0. В функции `main()` проверяется значение ошибки и выводится соответствующее сообщение.

Отличается ли обработка ошибок в Go от других ЯП? Если да, то чем?

Обработка ошибок в Go существенно отличается от других ЯП и имеет некоторые ключевые особенности:

1. **Явная обработка ошибок:** в Go нет механизма исключений, как во многих других языках. Вместо этого функции, которые могут вызвать ошибку, обычно возвращают значение ошибки как один из своих возвращаемых результатов.
2. **Множественные возвращаемые значения:** функции часто возвращают результат (или результаты) и ошибку. Это позволяет легко проверять наличие ошибки после каждого вызова функции.

```
val, err := someFunction()
if err != nil {
    // обработка ошибки
}
```

1. **Кастомные типы ошибок:** с помощью пакета `errors` можно создавать кастомные типы ошибок. Это дает возможность добавить дополнительную информацию к ошибке или создать проверяемые типы ошибок.
2. **Добавление дополнительного контекста к ошибке:** начиная с Go 1.13, были добавлены функции `errors.Is`, `errors.As` и `fmt.Errorf` для обертывания

ошибок, что позволяет сохранить исходную ошибку и добавить дополнительный контекст.

```
func DoSomething() error {  
    if err := someOperation(); err != nil {  
        return fmt.Errorf("someOperation failed: %w", err)  
    }  
    return nil  
}
```

1. **Нет finally** : так как в Go нет исключений, нет и блока `finally` . Очистка ресурсов или другие завершающие действия обычно выполняются с использованием `defer` .
2. **panic и recover** : хотя Go предпочитает явную обработку ошибок, существуют механизмы `panic` и `recover` для обработки исключительных ситуаций. Однако их рекомендуется использовать осторожно и в основном для обработки действительно неожиданных ошибок, таких как выход за границы массива.

Возврат ошибки в виде интерфейса `error` и выкидывание `panic` . Паника – это не тоже самое, что и классические исключения в других языках, поскольку паника гарантированно завершает выполнение текущей функции.

Как можно обработать панику? С помощью `recover` . Обратите внимание, что обработчик паники должен быть объявлен в той же горутине, где возникает паника.

Как можно определить место возникновения ошибки? В случае небольших микросервисов и приложений достаточно подробного описания ошибки, чтобы понять место её возникновения. В остальных случаях можно добавлять стек вызова функций в описание ошибки.

Реализовать функцию последовательности Фибоначчи

Классика! Стоит напомнить, что последовательность Фибоначчи — это ряд чисел, где каждое последующее является суммой двух предыдущих. Вот первые десять чисел: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Начнём с реализации через рекурсию:

```
package main

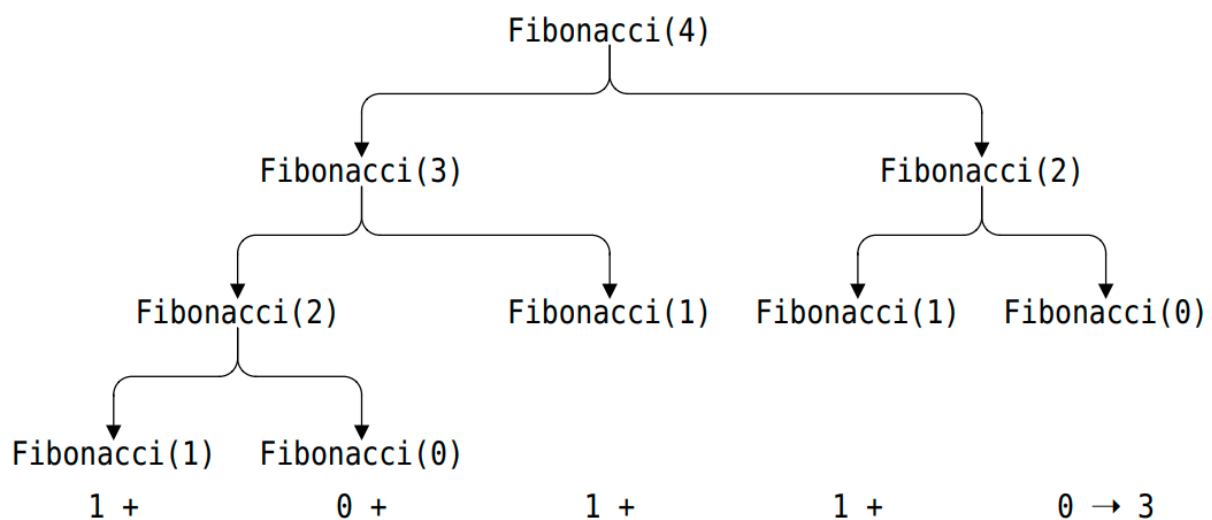
import "fmt"

func fibonacci(n uint) uint {
    if n < 2 {
        return n
    }

    return fibonacci(n-1) + fibonacci(n-2)
}

func main() {
    fmt.Println(fibonacci(10))
}
```

Рекурсивная реализация работает так:



Это работает хорошо, но появляется проблемка, когда параметр `n` имеет большое значение. Это происходит из-за того, что функция определяется рекурсивно: количество раз, когда функция должна вызывать саму себя, растет экспоненциально

по мере увеличения `n`. Например, попробуйте выполнить `fibonacci(100)` и программа будет считать медленно. Для преодоления этой проблемы мы можем улучшить наш код так, чтобы функция брала уже вычисленные ранее значения из кэша.

```
package main

import "fmt"

var (
    fibonacciCache = make(map[uint]uint)
)

func fibonacci(n uint) uint {
    if n < 2 {
        return n
    }

    if result, ok := fibonacciCache[n]; ok {
        return result
    }

    result := fibonacci(n-1) + fibonacci(n-2)

    fibonacciCache[n] = result

    return result
}

func main() {
    fmt.Println(fibonacci(1_000))
}
```

Теперь функция способна "переварить" большие аргументы. А вот решение без рекурсии:

```
package main
```

```
import "fmt"

func fibonacci(n uint) uint {
    if n < 2 {
        return n
    }

    var a, b uint

    b = 1

    for n--; n > 0; n-- {
        a += b
        a, b = b, a
    }

    return b
}

func main() {
    fmt.Println(fibonacci(100))
}
```

Мы улучшили производительность, но все еще есть предел тому, насколько высоко в последовательности Фибоначчи мы можем подняться. Проблема вызвана не тем, что нам не хватает вычислительной мощности или памяти. Это скорее потому, что числа Фибоначчи очень быстро становятся очень большими: даже если бы мы использовали `uint64`, мы бы вскоре переполнили тип данных. Тогда кажется очевидным, что нам нужно использовать другой тип возвращаемого значения в нашей функции Фибоначчи, который может содержать сколь угодно большие целые числа.

```
package main

import (
    "fmt"
    "math/big"
)
```

```
func fibonacci(n uint) *big.Int {
    if n < 2 {
        return big.NewInt(int64(n))
    }

    a, b := big.NewInt(0), big.NewInt(1)

    for n--; n > 0; n-- {
        a.Add(a, b)
        a, b = b, a
    }

    return b
}

func main() {
    fmt.Println(fibonacci(5_000))
}
```

Выше мы использовали пакет `math/big` из стандартной библиотеки Go, так что мы можем создавать чрезвычайно большие целые числа.

Функция `a.Add(a, b)` выполняет сложение, используя свои два аргумента, а затем сохраняет результат в `a`.

Теперь стало возможным получать огромные результаты, которые не смог бы вместить примитивный тип данных, например, где `n` равно 5000.

Что такое контекст (context) в Go и для чего он применяется?

`context` в Go — это специальный пакет, предназначенный для передачи параметров между API и управления жизненным циклом горутин.

Основное его назначение — передача метаданных, установка временных рамок выполнения и отслеживание отмены долгосрочных операций.

Основные моменты:

- `context` введен в Go 1.7 и с тех пор является предпочтительным механизмом для управления временем выполнения и отменами.
- Интерфейс `context.Context` является основным типом, который вы передаете между функциями.
- Основные методы: `WithCancel`, `WithDeadline`, `WithTimeout` и `WithValue`.
- `WithCancel` — возвращает копию переданного контекста и `cancelFunc`. Вызов `cancelFunc` отменяет этот контекст.
- `WithDeadline` & `WithTimeout` — позволяют задать временные рамки контексту.
- `WithValue` — позволяет передать произвольные пары ключ/значение в `context`.
- Отмена родительского `context` автоматически отменяет все дочерние.
- `context` используется для уведомления о том, что пора завершать работу, — это особенно удобно через канал `ctx.Done()`.
- Возвращаемая функция `cancel` позволяет рано завершить `context`.
- Не храните в `context` чувствительные данные: контекст может быть выведен и сохранен в логах, что может раскрыть чувствительные данные.
- При отмене `context` можно узнать причину через `ctx.Err()`, где возможные значения — `context.Canceled` или `context.DeadlineExceeded`.

Ну и простой пример:

```
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel() // освобождаем ресурсы

go someOperation(ctx)

if ctx.Err() == context.Canceled {
    fmt.Println("Operation was canceled")
}
```

Важно:

- `context.Background()` и `context.TODO()` — одно и то же. Разница лишь в том, что `context.TODO()` выставляется в местах, где пока нет понимания, что необходимо использовать `context.Background()` и возможно его надо заменить на дочерний `context`.
- Когда `context` отменяется (через `cancel`, `timeout` или `deadline`), `ctx.Done()` возвращает закрытый канал. Это удобный механизм для оповещения горутин о том, что пора завершать работу.

Как в Go реализованы конструкции циклов?

Цикл `for` используется для выполнения выражений определенное число раз. Цикл имеет следующий формат:

```
for <Начальное значение>; <Условие>; <Приращение> {  
    <Инструкции>  
}
```

Параметры имеют следующие значения:

- `<Начальное значение>` — присваивает переменной-счетчику начальное значение;
- `<Условие>` — содержит логическое выражение. Пока логическое выражение возвращает значение `true`, выполняются инструкции внутри цикла;
- `<Приращение>` — задает изменение переменной-счетчика на каждой итерации.

Цикл выполняется до тех пор, пока `<Условие>` не вернет `false`. Если это не произойдет, то цикл будет бесконечным. Логическое выражение, указанное в параметре `<Условие>`, вычисляется на каждой итерации.

Все параметры цикла `for` и инструкции внутри цикла являются необязательными. Хотя параметры можно не указывать, точки с запятой обязательно должны быть. Если

все параметры не указаны, то цикл окажется бесконечным. Чтобы выйти из бесконечного цикла следует использовать оператор `break`. Пример:

```
var i int = 1      // <Начальное значение>
for ; ; {          // Бесконечный цикл
    if i <= 10 {    // <Условие>
        fmt.Println(i)
        i++        // <Приращение>
    } else {
        break      // Выходим из цикла
    }
}
```

А вот пример использования `for` как отдельно, так и с ключевым словом `range`:

```
package main

import "fmt"

func main() {
    // традиционный цикл for
    for i := 0; i < 10; i++ {
        fmt.Print(i*i, " ")
    }
    fmt.Println()

    // 0 1 4 9 16 25 36 49 64 81
}
```

В этом коде показан традиционный цикл `for`, который использует локальную переменную `i`. Код выведет на экран квадраты 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9.

Перепишем этот код выше на более идиоматический для Go вариант:

```
package main

import "fmt"
```

```
func main() {  
    i := 0  
    for ok := true; ok; ok = (i != 10) {  
        fmt.Print(i*i, " ")  
        i++  
    }  
    fmt.Println()  
}
```

Аналог `while`. Существует также сокращенный формат цикла `for`, который аналогичен циклу `while` из других языков программирования:

```
<Начальное значение>  
for <Условие> {  
    <Инструкции>  
    <Приращение>  
}
```

Выведем все числа от 1 до 100 :

```
var i int = 1      // <Начальное значение>  
for i <= 100 {     // <Условие>  
    fmt.Println(i) // <Инструкции>  
    i++            // <Приращение>  
}
```

Перебор элементов массива, словаря и строки.

Пример перебора элементов массива и слайса:

```
var arr1 = [3]int{10, 20, 30}  
for index, value := range arr1 {  
    fmt.Println(index, value)  
}  
var arr2 = []int{40, 50, 60}
```



```
for index, value := range arr2 {  
    fmt.Println(index, value)  
}  
  
// 0 10  
// 1 20  
// 2 30  
// 0 40  
// 1 50  
// 2 60
```

Пример перебора элементов словаря:

```
var dict = map[string]int {  
    "x": 10,  
    "y": 20,  
    "z": 30,  
}  
for key, value := range dict {  
    fmt.Println(key, value)  
}  
  
// x 10  
// y 20  
// z 30
```

Пример перебора символов строки:

```
str := "тест"  
for index, ch := range str {  
    fmt.Println(index, ch, string(ch))  
}  
  
// 0 1090 т  
// 2 1077 е  
// 4 1089 с  
// 6 1090 т
```

FizzBuzz

Классика. Задача сводится к тому, чтобы написать программу, которая будет выводить числа от 1 до 100, при этом она должна выводить "Fizz", если число кратно 3, "Buzz", если число кратно 5, и "FizzBuzz", если число кратно и 3, и 5 одновременно.

Пишется элементарно, просто цикл `for` с условиями `if`.

```
package main

import "fmt"

func main() {
    for i := 1; i <= 100; i++ {
        if i%3 == 0 && i%5 == 0 {
            fmt.Println("FizzBuzz")
        } else if i%3 == 0 {
            fmt.Println("Fizz")
        } else if i%5 == 0 {
            fmt.Println("Buzz")
        } else {
            fmt.Println(i)
        }
    }
}
```

Можно ли вернуть из функции несколько значений?

```
func <Название функции>([<Название параметра 1> <Тип>
    [, ..., <Название параметра N> <Тип>]])[ <Тип результата>] {
    <Тело функции>
}
```

```
[return[ <Возвращаемое значение>]]  
}
```

Да, из функции можно вернуть сразу несколько значений. В этом случае в параметре <Тип результата> типы возвращаемых значений перечисляются через запятую внутри круглых скобок. В операторе `return` возвращаемые значения указываются через запятую.

Ну и вот пример функции, которая возвращает несколько значений:

```
package main  
  
import "fmt"  
  
func main() {  
    arr := []int{2, 5, 6, 1, 3}  
    // Получение всех значений  
    min, max := MinMax(arr)  
    fmt.Println("min =", min)  
    fmt.Println("max =", max)  
    // Получение только первого значения  
    min, _ = MinMax(arr)  
    fmt.Println("min =", min)  
    // Получение только второго значения  
    _, max = MinMax(arr)  
    fmt.Println("max =", max)  
}  
  
func MinMax(arr []int) (int, int) {  
    min := arr[0]  
    max := arr[0]  
    for _, value := range arr {  
        if value < min {  
            min = value  
        }  
        if value > max {  
            max = value  
        }  
    }  
}
```

```
    }  
    return min, max  
}  
  
// min = 1  
// max = 6  
// min = 1  
// max = 6
```

Объясните разницу между конкурентностью и параллельностью в Go

Конкурентность — это, когда программа может работать с несколькими задачами одновременно в рамках одного процесса. Конкурентность обеспечивает выполнение нескольких задач посредством переключения контекста.

Конкурентные вычисления реализуются на одном ядре системы. Прimitives конкурентности в Go:

- горутины
- каналы
- мьютексы (объекты `Mutex` , `RWMutex`)
- оператор `select ... case`
- объекты `waitGroup` , `errGroup`

Горутины — это конкурентные легковесные потоки, а каналы позволяют им взаимодействовать в процессе выполнения.

Параллельность — это, когда программа может одновременно выполнять несколько задач на нескольких процессорах.

Другими словами, конкурентность — это свойство программы, которое позволяет нескольким задачам быть запущенными одновременно, но не обязательно

одновременно выполняться. Параллельность же относится к свойствам среды выполнения, когда две или более задач выполняются одновременно.

Это значит, что посредством параллельности можно получить конкурентное поведение, но на этом ее возможности не ограничиваются.

Реализуйте функции `min` и `max`

Реализуйте функции `Min(x, y int)` и `Max(x, y int)`, получающие два целых числа и возвращающих меньшее или большее значение соответственно.

Решение в целом очевидное

```
package main
import "fmt"
// Min возвращает меньшее из x или y.
func Min(x, y int) int {
    if x > y {
        return y
    }
    return x
}
// Max возвращает большее из x или y.
func Max(x, y int) int {
    if x < y {
        return y
    }
    return x
}
func main() {
    fmt.Println(Min(5,10))
    fmt.Println(Max(5,10))
}
```

Какие механизмы синхронизации доступны в Golang?

В Go примитивы синхронизации — это инструменты из пакета `sync` (и не только), которые помогают нам гарантировать, что множество горутин может безопасно взаимодействовать с общими данными или координировать свою работу.

- `sync.Mutex` : основной примитив блокировки для исключения одновременного доступа к данным. Мьютексы позволяют только одной горутине получить доступ к общему ресурсу в определенный момент времени.
- `sync.RWMutex` : разрешает множественное чтение или одну операцию записи в текущий момент времени.
- `sync.WaitGroup` : используется для ожидания завершения группы горутин перед продолжением выполнения основной программы.
- `sync.Once` : гарантирует, что функция будет вызвана только один раз, несмотря на количество вызовов.
- `sync.Cond` : предоставляет механизм для блокирования горутин, пока не будет выполнено некоторое условие. Не так давно Расс Кокс [отменил](#) предложение удалить данные тип в будущей версии Go.

Подобную роль играют:

- **Каналы.** Каналы в Go хоть и не являются примитивами синхронизации в традиционном понимании, они играют ключевую роль в управлении горутинами, позволяют обеспечить безопасный обмен данными между ними. Каналы обеспечивают синхронизацию и блокируют выполнение до тех пор, пока данные не будут переданы или приняты.
- **Атомарные операции:** Golang предоставляет атомарные операции для безопасного выполнения операций чтения и записи разделяемых данных.

Что такое атомарная операция и для чего предназначен пакет `atomic`?

Атомарная операция выполняется за один шаг относительно других потоков или, в контексте Go, других горутин. Это означает, что атомарную операцию нельзя прервать в середине ее работы.

Стандартная библиотека Go содержит пакет `atomic`, который в некоторых простых случаях может помочь избежать использования мьютекса. С помощью него мы получаем доступ к атомарным счетчикам из нескольких горутин, не имея проблем с синхронизацией и не беспокоясь о *race condition*.

Как показано в примере ниже, при использовании атомарной переменной во избежание *race condition* все операции чтения и записи атомарной переменной должны выполняться с помощью функций, предоставляемых пакетом `atomic`.

```
package main
import (
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64 }

```

Это структура для хранения требуемой атомарной переменной `int64`.

```
func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}

```

Это вспомогательная функция, которая возвращает текущее значение атомарной переменной `int64`, используя `atomic.LoadInt64()`.

```
func main() {  
    X := 100  
    Y := 4  
    var waitGroup sync.WaitGroup  
    counter := atomCounter{  
    for i := 0; i < X; i++ {
```

Мы создаем множество горутин, которые изменяют общую переменную. Благодаря использованию пакета `atomic` для работы с общей переменной мы получаем простой способ избежать *race condition* при изменении ее значения.

```
        waitGroup.Add(1)  
        go func(no int) {  
            defer waitGroup.Done()  
            for i := 0; i < Y; i++ {  
                atomic.AddInt64(&counter.val, 1)  
            }  
        }
```

Функция `atomic.AddInt64()` безопасно изменяет значение поля `val` структуры `counter`.

```
    }(i)  
}  
  
    waitGroup.Wait()  
    fmt.Println(counter.Value())  
}
```

Как устроен мьютекс?

Mutex означает MUTual EXclusion (взаимное исключение), и обеспечивает безопасный доступ к общим ресурсам. Мьютексы — один из наиболее распространенных примитивов синхронизации.

Под капотом мьютекса используются функции из пакета `atomic` (`atomic.CompareAndSwapInt32` и `atomic.AddInt32`), так что можно считать мьютекс надстройкой над `atomic`. Мьютекс медленнее чем `atomic`, потому что он блокирует другие горуты на всё время действия блокировки. А в свою очередь `atomic` быстрее потому как использует атомарные инструкции процессора.

В момент, когда нужно обеспечить защиту доступа — вызываем метод `Lock()`, а по завершению операции изменения/чтения данных — метод `Unlock()`.

Стандартная библиотека Go предоставляет два типа мьютексов для синхронизации доступа к общим ресурсам:

`sync.Mutex` — стандартный мьютекс, который предоставляет эксклюзивную блокировку (exclusive lock). Только одна горутина может захватить мьютекс и получить доступ к общему ресурсу.

```
package main

import (
    "fmt"
    "sync"
)

var count int
var mu sync.Mutex

func increment() {
    mu.Lock()
    count++
    mu.Unlock()
}

func main() {
    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            increment()
        }()
    }
    wg.Wait()
    fmt.Println(count)
```

```
        wg.Done()
    }()
}

wg.Wait()
fmt.Println(count)
}
```

Здесь мы используем `sync.Mutex` для обеспечения безопасности при инкременте глобальной переменной `count` из множества горутин.

`sync.RWMutex` — концептуально то же самое, что и `Mutex`. Тем не менее, `RWMutex` дает вам немного больше контроля над памятью.

Он предоставляет доступ к критической секции произвольному количеству читателей и не более, чем одному писателю. При этом, если есть писатель, то читателей нет.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var cache = make(map[string]string)
var mu sync.RWMutex

func set(key string, value string) {
    mu.Lock()
    cache[key] = value
    mu.Unlock()
}

func get(key string) string {
    mu.RLock()
    defer mu.RUnlock()
    return cache[key]
}
```

```
}

func main() {
    set("name", "John")

    var wg sync.WaitGroup

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            fmt.Println(get("name"))
            wg.Done()
        }()
    }

    time.Sleep(1 * time.Second)
    set("name", "Doe")

    wg.Wait()
}
```

Здесь мы используем `sync.RWMutex` для обеспечения безопасного доступа к кэшу. Множество горутин может одновременно читать из кэша, но только одна горутина может писать в кэш в данный момент времени.

В чем отличие `sync.Mutex` от `sync.RWMutex` ?

Помимо `Lock()` и `Unlock()` (у `sync.Mutex`), у `sync.RWMutex` есть отдельные аналогичные методы **только для чтения** — `RLock()` и `RUnlock()`. Если участок в памяти нуждается только в чтении — он использует `RLock()`, который не заблокирует другие операции чтения, **но заблокирует операцию записи** и наоборот.

По большому счёту, `RWMutex` это комбинация из двух мьютексов.

Как работает управление памятью в Go?

Go использует сборщик мусора для автоматического управления памятью.

Разработчику не нужно явно выделять и освобождать память, как в языках типа C или C++. Однако нужно быть внимательным при работе с большими структурами данных, чтобы избежать утечек памяти.

Некоторые ключевые аспекты управления памятью в Go:

- Go применяет алгоритм сборки мусора с маркировкой и освобождением. Сборщик мусора отмечает активные объекты, после чего освобождает память от неактивных.
- В Go можно работать с указателями, но нет прямого управления выделением и освобождением памяти через них. Память выделяется при создании объектов и автоматически освобождается сборщиком мусора.
- Хотя Go управляет памятью автоматически, неправильное использование, например, из-за циклических ссылок, может вызвать утечки памяти. Поэтому важно контролировать использование ресурсов.
- Срезы в Go — это динамические массивы, обеспечивающие автоматическое управление памятью при изменении их размера.
- Go разделяет память на стек и кучу. Стек — для локальных переменных и контекста функций; каждый поток имеет свой стек. Куча — для долгоживущих объектов и данных, которые могут быть доступны из разных частей программы. Управление памятью в куче осуществляется сборщиком мусора.
- *Escape analysis* в Go определяет, следует ли объекту быть на стеке или в куче, опираясь на его использование в программе. Этот анализ помогает оптимизировать управление памятью, делая его более эффективным.

Как легче всего проверить срез на пустоту?

Создайте программу, проверяющую срез на пустоту. Найдите самое простое решение.

Решение довольно простое. Легче всего проверить срез на пустоту с помощью встроенной функции `len()`, которая возвращает длину среза. Если `len(slice) == 0`, значит срез пуст.

Например, можно сделать так:

```
package main
import "fmt"
func main() {
    r := [3]int{1, 2, 3}
    if len(r) == 0 {
        fmt.Println("Empty!")
    } else {
        fmt.Println("Not Empty!")
    }
}
```

Как можно создать веб-сервер с использованием Golang?

В Golang создание веб-сервера осуществляется с использованием пакета `net/http`.
А вот пример создания простого веб-сервера:

```
goCopy code
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, World!")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Тут у нас функция `handler` является обработчиком запросов и выводит "Hello, World!" в ответ на любой запрос. Функция `main` устанавливает обработчик и запускает веб-сервер на порту 8080.

Собственно, вот и весь процесс создания сервера на Go.

****Что нужно, чтобы две функции были одного типа?****

Если мы хотим, чтобы две функции в Go считались одного типа, они должны иметь *одинаковую сигнатуру функции*.

```
package main

type sigFunc func(a int, b float64) (bool, error)

func functionA(a int, b float64) (bool, error) {
    return true, nil
}

func functionB(a int, b float64) (bool, error) {
    return false, nil
}

func main() {
    var x sigFunc = functionA
    x = functionB
    print(x)
}
```

Это просто означает, что они должны иметь соответствующие параметры (количество, типы) и возвращаемые значения.

Реализовать сортировку слиянием, используя горутины и каналы

В качестве опорной точки можно взять эту последовательную реализацию:

```
package main
import "fmt"

func Merge(left, right [] int) [] int{
    merged := make([] int, 0, len(left) + len(right))

    for len(left) > 0 || len(right) > 0{
        if len(left) == 0 {
            return append(merged,right...)
        } else if len(right) == 0 {
            return append(merged,left...)
        } else if left[0] < right[0] {
            merged = append(merged, left[0])
            left = left[1:]
        } else{
            merged = append(merged, right [0])
            right = right[1:]
        }
    }
    return merged
}

func MergeSort(data [] int) [] int {
    if len(data) <= 1 {
        return data
    }

    mid := len(data)/2
    left := MergeSort(data[:mid])
    right := MergeSort(data[mid:])
```

```
    return Merge(left,right)
}

func main(){
    data := [] int{9,4,3,6,1,2,10,5,7,8}
    fmt.Printf("%v\n%v", data, MergeSort(data))
}
```

Ну и доработаем её, используя горутины и каналы:

```
package main
import "fmt"

func Merge(left, right [] int) [] int{
    merged := make([] int, 0, len(left) + len(right))

    for len(left) > 0 || len(right) > 0{
        if len(left) == 0 {
            return append(merged,right...)
        } else if len(right) == 0 {
            return append(merged,left...)
        } else if left[0] < right[0] {
            merged = append(merged, left[0])
            left = left[1:]
        } else{
            merged = append(merged, right [0])
            right = right[1:]
        }
    }
    return merged
}

func MergeSort(data [] int) [] int {
    if len(data) <= 1 {
        return data
    }

    done := make(chan bool)
```



```
mid := len(data)/2
var left [] int

go func(){
    left = MergeSort(data[:mid])
    done <- true
}()

right := MergeSort(data[mid:])
<-done
return Merge(left,right)
}

func main(){
    data := [] int{9,4,3,6,1,2,10,5,7,8}
    fmt.Printf("%v\n%v", data, MergeSort(data))
}
```

В начале при сортировке слиянием мы рекурсивно разделяем массив на `right` и `left` стороны и вызываем `MergeSort` для обеих сторон.

Теперь нужно сделать так, чтобы `Merge(left, right)` выполнялась после получения возвращаемых значений от обоих рекурсивных вызовов, то есть и `left`, и `right` должны обновляться до того, как `Merge(left, right)` сможет быть выполнена. Для этого на строке 26 мы вводим канал типа `bool` и отправляем в него `true` сразу после выполнения `left = MergeSort(data[:mid])`.

Операция `<-done` блокирует код до инструкции `Merge(left,right)`, чтобы она не продолжилась, пока горутина не завершится. После завершения горутины и получения `true` в канале `done` код переходит к инструкции `Merge(left, right)`.

Каков побочный эффект использования ``time.After`` в выражении ``select``?

Если вы не знакомы с `time.After`, это функция в пакете времени Go, которая возвращает набор каналов для отправки текущего времени после указанной продолжительности.

```
func After(d Duration) <-chan Time
```

Обычно он используется в операторах `select` для реализации тайм-аутов или задержек. Например, представьте, что вы ждете 3 секунды, прежде чем напечатать что-то на экране:

```
func main() {
    timeout := 3 * time.Second
    start := time.Now()
    done := make(chan bool)

    select {
    case <-done:
        fmt.Println("Operation completed.")
        return
    case <-time.After(timeout):
        fmt.Printf("Timeout after %vn", time.Since(start))
    }
}
```

Что ж, теперь поговорим о побочном эффекте.

Для краткосрочных `time.After` это может не иметь большого значения, но рассмотрим сценарий, в котором тайм-аут установлен на 1 час, а работа заканчивается до истечения времени ожидания. В этой ситуации таймер все еще задерживается в памяти:

```
func main() {
    done := make(chan bool)

    go func() {
        time.Sleep(500 * time.Millisecond)
        done <- true
    }
}
```

```
    }()

    for {
        select {
        case <-done:
            fmt.Println("Operation completed.")
            return
        case <-time.After(time.Hour):
            fmt.Println("Still waiting...")
        }
    }
}
```

Как следствие, горутина, созданная `time.After`, не завершится, пока не истечет полный час, даже если операция завершится раньше.

Расскажи про `recover`

Панику можно обработать внутри отложенной функции и восстановить нормальное выполнение программы. Для этого предназначена глобальная функция `recover()`. Формат функции:

```
recover() interface{}
```

Если возникла паника, то функция вернет объект ошибки, указанный в функции `panic()`. Если паника не возникла, то возвращается значение `nil`. Вызывать функцию `recover()` нужно внутри отложенной функции (функции, зарегистрированной с помощью инструкции `defer`). После вызова функции `recover()` считается, что паника обработана и можно продолжить выполнение программы.

Вот пример обработки деления на 0:

```
package main
```

```
import "fmt"

func main() {
    fmt.Println(division(10, 2))
    fmt.Println(division(10, 0))
    fmt.Println("Выполнение программы продолжается!")
}

func division(x, y int) (n int) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println(r)
            n = 0 // Возвращаем из функции division() ноль
        }
    }()
    fmt.Println("Инструкция до деления")
    n = x / y
    fmt.Println("Инструкция после деления")
    return
}

// Инструкция до деления
// Инструкция после деления
// 5
// Инструкция до деления
// runtime error: integer divide by zero
// 0
// Выполнение программы продолжается!
```

Реализовать пересечение двух слайсов

На вход подаются два неупорядоченных слайса любой длины. Надо написать функцию, которая возвращает их пересечение. Стандартная задача и ее довольно часто спрашивают на собеседованиях в качестве простой задачи для разогрева.

Можно решить сортировкой за более долгое время, но без выделения дополнительной памяти. А можно выделить дополнительную память и решить за линейное время $O(n)$. Надо посчитать количество появлений элементов первого массива (лучше брать тот, что покороче) — используем для этого словарь. Потом пройтись по второму массиву и вычитать из словаря те элементы, которые есть в нем. По ходу добавляем в результат те элементы, у которых частота появлений больше нуля.

И получаем что-то такое:

```
package main

import (
    "fmt"
)

// На вход подаются два неупорядоченных массива любой длины.
// Необходимо написать функцию, которая возвращает пересечение массивов
func intersection(a, b []int) []int {
    counter := make(map[int]int)
    var result []int

    for _, elem := range a {
        if _, ok := counter[elem]; !ok {
            counter[elem] = 1
        } else {
            counter[elem] += 1
        }
    }
    for _, elem := range b {
        if count, ok := counter[elem]; ok && count > 0 {
            counter[elem] -= 1
            result = append(result, elem)
        }
    }
    return result
}

func main() {
```

```
a := []int{23, 3, 1, 2}
b := []int{6, 2, 4, 23}

// [2, 23]
fmt.Printf("%v", intersection(a, b))
a = []int{1, 1, 1}
b = []int{1, 1, 1, 1}

// [1, 1, 1]
fmt.Printf("%v", intersection(a, b))
}
```

В чем разница между методами `Time.Sub()` и `Time.Add()` пакета `time`?

Основное различие между методами `Time.Add()` и `Time.Sub()` в пакете `time` заключается в их параметрах и возвращаемых значениях. `Time.Add()` принимает параметр `Duration` и возвращает значение `Time`, а `Time.Sub()` принимает параметр `Time` и возвращает `Duration`.

Методы `Time.Add()` и `Time.Sub()` служат разным целям и имеют разные сигнатуры для конкретных вариантов использования:

```
func main() {
    now := time.Now()

    newTime := now.Add(2 * time.Hour)
    fmt.Println("Time after 2 hours:", newTime)
    newTime = now.Add(2 * time.Hour)
    fmt.Println("Time before 2 hours:", newTime)

    duration := newTime.Sub(now)
}
```

```

    fmt.Println("Duration newTime to now:", duration)
}
Time after 2 hours: 2023-05-09 03:05:03.177199 +0700 +07 m=+7200.000587
Time before 2 hours: 2023-05-09 03:05:03.177199 +0700 +07 m=+7200.000587
Duration newTime to now: 2h0m0s

```

Как показано в этом примере, `Time.Add()` используется для добавления или вычитания продолжительности из значения времени, а `Time.Sub()` используется для вычисления продолжительности между двумя значениями времени.

Что такое теги структур?

Теги структур в Go — это метаданные, прикрепленные к полям структуры, которые могут быть использованы для предоставления дополнительной информации или инструкций внешним пакетам или библиотекам.

```
`<Ключ>:"<Значение>"`
```

Пример добавления тегов:

```

type Point struct {
    X int `json:"x"`
    Y int `json:"y"`
}

```

Получить значение тега позволяют методы из пакета `reflect` :

```

// import "reflect"
p := Point{10, 20}
t := reflect.TypeOf(p)
field, ok := t.FieldByName("X")
if ok {
    fmt.Println(field.Tag)           // json:"x"
}

```

```
fmt.Println(field.Tag.Get("json")) // x  
}
```

Теги структур могут быть использованы для различных целей, включая:

1. **Контроль сериализации и десериализации:** теги могут указывать, как поля должны быть сериализованы или десериализованы в форматы, такие как JSON или XML. Например, тег `json:"name,omitempty"` указывает, что поле `Name` должно быть сериализовано как `name` в JSON, и если поле пустое, его следует опустить.
2. **Валидация данных:** теги могут быть использованы для указания правил валидации для полей, например, минимальной или максимальной длины строки.
3. **Описания и документация:** теги могут содержать документацию или описания полей.
4. **Оркестровка баз данных:** теги могут быть использованы для маппинга полей структуры на столбцы в базе данных.
5. **Другие кастомные обработки:** теги могут быть использованы для произвольной обработки кастомными библиотеками или кодом.

Для доступа к тегам структуры и их разбора часто используется пакет `reflect`. Он предоставляет функции для работы с типами и значениями во время выполнения, что позволяет изучать и изменять значения, типы и теги структур во время выполнения.

Подробнее [тут](#).

Исправь код

В коде ниже есть ошибка, её предстоит исправить.

```
package main  
  
import (  
    "fmt"  
)  
  
type Person struct {
```



```
    FirstName string
    LastName  string
}

func (p Person) Married(husband Person) {
    p.LastName = husband.LastName
}

func main() {
    eva := Person{"Eva", "First"}
    adam := Person{"Adam", "Second"}
    eva.Married(adam)

    fmt.Println(eva)
}
```

В Go можно передавать параметры в функцию по ссылке и по значению. Если параметр передается по значению(как в нашем примере), то все параметры копируются в другие адреса памяти и работа внутри функции происходит с ними, поэтому ожидаемой смены фамилии не происходит. Если же параметр функции передается по ссылке, создается новая ссылка на существующую область памяти и, соответственно, при изменении меняется и то значение которое находится по ссылке.

Для ожидаемого поведения нужно изменить объявление функции; вместо этого:

```
func (p Person) Married(husband Person)
```

написать так:

```
func (p *Person) Married(husband Person)
```

Таким образом мы передадим функции параметр не по значению, а по ссылке, что и нужно для правильной работы.

Если в функции есть return, обязательно ли она вернет то, что указано в return?

Мы привыкли, что обычно, если код внутри функции добрался до `return`, то на выходе мы получим то, что стоит после `return`. Но в Go есть интересная особенность: если есть именованный выходной параметр (параметры), то функция вернет последнее его значение, несмотря на то, что написано в `return`.

```
package main

import "fmt"

// Основной метод
func main() {
    // функция возвращает два значения
    m, d := calculator(105, 7)
    fmt.Println("105 x 7 = ", m)
    fmt.Println("105 / 7 = ", d)
}

// функция с именованными аргументами
func calculator(a, b int) (mul int, div int) {
    // здесь простое присваивание т.к. инициализация произошла выше
    // функция вернет именно эти переменные
    mul = a * b
    div = a / b

    // переменные, которые попытаемся вернуть через return
    test := 22
    best := 34
    // здесь у вас есть ключевое слово return
    return test, best
}
```

Что такое `iota`?

`iota` используется для создания последовательности целочисленных констант. Оно автоматически увеличивается на 1 после каждого использования:

```
const (  
    c0 = iota // c0 == 0  
    c1 = iota // c1 == 1  
    c2 = iota // c2 == 2  
)
```

Также `iota` можно использовать для:

- создания битовых масок. В этом случае каждая последующая константа будет иметь значение, увеличенное на степень двойки от предыдущей.
- создания последовательности строковых констант. В этом случае каждая последующая константа будет иметь значение, равное ее имени.

Строго говоря, значением `iota` является индекс `ConstSpec`. Несмотря на то, что первым индексом является 0, значение первой константы можно задать отличным от 0, что в свою очередь повлияет на значения последующих констант.

Реализовать генератор случайных чисел

Для решения можно использовать небуферизированный канал. Будем асинхронно писать туда случайные числа и закроем его, когда закончим писать:

```
package main  
  
import (  
    "fmt"  
    "math/rand"  
    "time"
```

)

```
func randNumsGenerator(n int) <-chan int {  
    r := rand.New(rand.NewSource(time.Now().UnixNano()))  
  
    out := make(chan int)  
    go func() {  
        for i := 0; i < n; i++ {  
            out <- r.Intn(n)  
        }  
        close(out)  
    }()  
    return out  
}  
  
func main() {  
    for num := range randNumsGenerator(10) {  
        fmt.Println(num)  
    }  
}
```

Вот собственно и всё, решение может выглядеть так.

Что такое псевдоним типа (type alias) в Go?

Псевдоним типа — это функциональность, позволяющая создавать альтернативное имя для существующего типа данных. Это особенно полезно при рефакторинге кода, когда необходимо переименовать тип или сделать его более удобным для использования, не меняя основного определения типа.

Псевдонимы типов вводятся с использованием ключевого слова `type` :

```
type <Псевдоним> <Существующий тип>
```

Псевдонимы типов полностью идентичны их оригинальным типам, включая методы, связанные с типом:

```
type MyInt int
var x MyInt = 10
fmt.Println(x)                // 10
fmt.Printf("%Tn", x)          // main.MyInt
```

Как видно тип `MyInt` наследует все свойства типа `int`, хотя это новый тип данных.

Как отсортировать ****массив структур**** по алфавиту по полю ``Name``?

Как вариант, это можно сделать так: преобразуем массив в слайс и воспользуемся функцией `sort.SliceStable`:

```
package main

import (
    "fmt"
    "sort"
)

func main() {
    var arr = [...]struct{ Name string }{{Name: "b"}, {Name: "c"}, {Name:
    //                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ анонимная структура с нужным нам

    fmt.Println(arr) // [{b} {c} {a}]

    sort.SliceStable(arr[:], func(i, j int) bool { return arr[i].Name < arr[j].Name
    //                ^^^ вот тут вся фишка - из массива сделали слайс
```

```
fmt.Println(arr) // [{a} {b} {c}]  
}
```

Вся фишка в том, что при создании слайса из массива “под капотом” у слайса образуется исходный массив, и функции из пакета `sort` нам становятся доступны над ними. Т.е. изменяя порядок элементов в слайсе функцией `sort.SliceStable` мы будем менять их в нашем исходном массиве.

Что такое сериализация? Зачем она нужна?

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи. Обратной операцией является десериализация (т.е. восстановление объекта/структуры из последовательности байтов). Синонимом можно считать термин “маршалинг” (marshal — упорядочивать).

Из минусов сериализации можно выделить нарушение инкапсуляции, т.е. после сериализации “приватные” свойства структур могут быть доступны для изменения.

Типичными примерами сериализации в Go являются преобразование структур в json-объекты. Кроме json существуют различные кодеки типа `MessagePack`, `CBOR` и т.д.

Слить N каналов в один

Задача: даны `n` каналов типа `chan int`. Надо написать функцию, которая смерджит все данные из этих каналов в один и вернет его.

Для этого напишем функцию, которая будет асинхронно читать из исходных каналов, которые ей передадут в качестве аргументов, и писать в результирующий канал, который вернется из функции.

Создаем канал, куда будем сливать все данные. Он будет небуферизированный, потому что мы не знаем, сколько данных придет из каналов.

Дальше асинхронно прочитаем из исходных каналов и закроем результирующий канал для мерджа, когда все чтение закончится. Чтобы дождаться конца чтения, просто обернем этот цикл по каналам в `wait group`.

```
package main

import (
    "fmt"
    "sync"
)

func joinChannels(chs ...chan int) chan int {
    mergedCh := make(chan int)

    go func() {
        wg := &sync.WaitGroup{}

        wg.Add(len(chs))

        for _, ch := range chs {
            go func(ch chan int, wg *sync.WaitGroup) {
                defer wg.Done()
                for id := range ch {
                    mergedCh <- id
                }
            }(ch, wg)
        }

        wg.Wait()
        close(mergedCh)
    }()

    return mergedCh
}

func main() {
    a := make(chan int)
    b := make(chan int)
    c := make(chan int)
```

```
go func() {  
    for _, num := range []int{1, 2, 3} {  
        a <- num  
    }  
    close(a)  
}()  
  
go func() {  
    for _, num := range []int{20, 10, 30} {  
        b <- num  
    }  
    close(b)  
}()  
  
go func() {  
    for _, num := range []int{300, 200, 100} {  
        c <- num  
    }  
    close(c)  
}()  
  
for num := range joinChannels(a, b, c) {  
    fmt.Println(num)  
}  
}
```

Как устроен сетевой ввод-вывод в Go?

Сетевой ввод-вывод в Go организован через пакет `net` стандартной библиотеки, который предоставляет обширный API для работы с сетью. Он использует модель неблокирующего ввода-вывода с горутинами для обеспечения масштабируемости и эффективности.

Когда мы создаем сетевое соединение или слушаем порт, каждая операция ввода-вывода (например, чтение или запись данных) может выполняться в отдельной горутине, позволяя обрабатывать множество соединений параллельно без блокировки главного потока выполнения.

Go автоматически управляет множеством горутин, что упрощает написание масштабируемого асинхронного сетевого кода по сравнению с традиционными подходами, основанными на потоках.

Вот простой пример, из него должно быть всё понятно:

```
package main

import (
    "fmt"
    "io"
    "net"
    "os"
)

func main() {
    // Слушаем на порту 8080
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        fmt.Println("Ошибка при создании слушателя:", err)
        os.Exit(1)
    }
    defer listener.Close()
    fmt.Println("Сервер запущен и слушает на порту 8080")

    for {
        // Принимаем входящее подключение
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Ошибка при принятии подключения:", err)
            continue
        }

        // Обработка подключения в отдельной горутине
    }
}
```

```

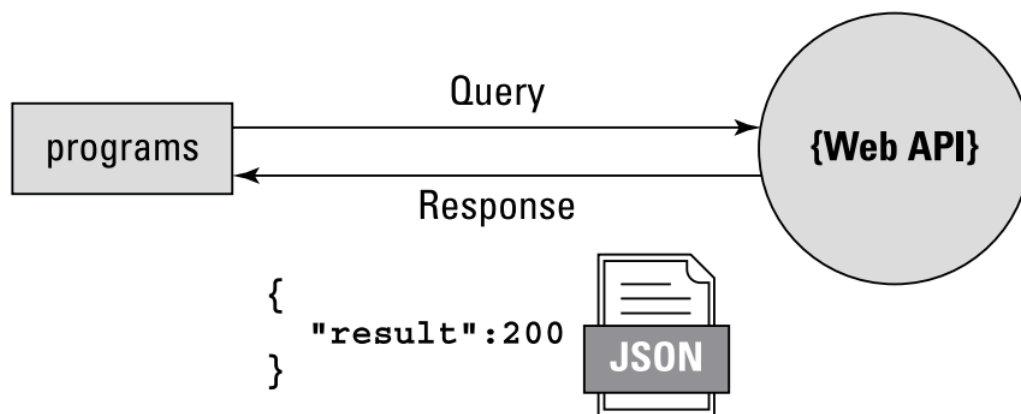
    go handleConnection(conn)
}

// handleConnection обрабатывает отдельное подключение
func handleConnection(conn net.Conn) {
    defer conn.Close()
    fmt.Println("Подключился клиент:", conn.RemoteAddr().String())

    // Отправляем сообщение клиенту
    _, err := io.WriteString(conn, "Привет от сервера!\n")
    if err != nil {
        fmt.Println("Ошибка при отправке сообщения:", err)
        return
    }

    fmt.Println("Сообщение отправлено клиенту:", conn.RemoteAddr().String())
}

```



Какие побитовые операторы знаешь?

Побитовые операторы предназначены для манипуляции отдельными битами. Язык Go поддерживает следующие побитовые операторы:

- **&** — двоичное И :

```
var x, y, z uint8 = 100, 75, 0
z = x & y
fmt.Printf("%bn", x) // 1100100
fmt.Printf("%bn", y) // 1001011
fmt.Printf("%bn", z) // 1000000
```

- `|` — двоичное ИЛИ :

```
var x, y, z uint8 = 100, 75, 0
z = x | y
fmt.Printf("%bn", x) // 1100100
fmt.Printf("%bn", y) // 1001011
fmt.Printf("%bn", z) // 1101111
```

- `^` — двоичное исключающее ИЛИ :

```
var x, y, z uint8 = 100, 250, 0
z = x ^ y
fmt.Printf("%bn", x) // 1100100
fmt.Printf("%bn", y) // 11111010
fmt.Printf("%bn", z) // 10011110
```

- `&^` — двоичное И НЕ :

```
var x, y, z uint8 = 100, 75, 0
z = x &^ y
fmt.Printf("%bn", x) // 1100100
fmt.Printf("%bn", y) // 1001011
fmt.Printf("%bn", z) // 100100
```

- `<<` — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
var x uint8 = 100
fmt.Printf("%bn", x) // 1100100
```

```
x = x << 1
fmt.Printf("%bn", x) // 11001000
x = x << 1
fmt.Printf("%bn", x) // 10010000
x = x << 2
fmt.Printf("%bn", x) // 1000000
```

- >> — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
var x uint8 = 100
fmt.Printf("%bn", x) // 1100100
x = x >> 1
fmt.Printf("%bn", x) // 110010
x = x >> 1
fmt.Printf("%bn", x) // 11001
x = x >> 2
fmt.Printf("%bn", x) // 110
```

Наиболее часто двоичное представление числа и побитовые операторы используется для хранения различных флагов (0 — флаг сброшен, 1 — флаг установлен). Вот примеры установки, снятия и проверки установки флага:

```
package main

import "fmt"

func main() {
    const (
        FLAG1 uint8 = 1 << iota
        FLAG2
        FLAG3
        FLAG4
        FLAG5
        FLAG6
        FLAG7
    )
```

FLAG8

```

)
var x uint8 = 0      // Все флаги сброшены
fmt.Printf("%bn", x) //      0
var y uint8 = 0xFF    // Все флаги установлены
fmt.Printf("%bn", y) // 11111111
// Устанавливаем флаги FLAG1 и FLAG7
x = x | FLAG1 | FLAG7
fmt.Printf("%bn", x) // 1000001
// Устанавливаем флаги FLAG4 и FLAG5
x = x | FLAG4 | FLAG5
fmt.Printf("%bn", x) // 1011001
// Снимаем флаги FLAG4 и FLAG5
x = x ^ FLAG4 ^ FLAG5
fmt.Printf("%bn", x) // 1000001
// Проверка установки флага FLAG1
if (x & FLAG1) != 0 {
    fmt.Println("FLAG1 установлен")
}
fmt.Printf("%bn", FLAG1) //      1
fmt.Printf("%bn", FLAG2) //     10
fmt.Printf("%bn", FLAG3) //    100
fmt.Printf("%bn", FLAG4) //   1000
fmt.Printf("%bn", FLAG5) //  10000
fmt.Printf("%bn", FLAG6) // 100000
fmt.Printf("%bn", FLAG7) // 1000000
fmt.Printf("%bn", FLAG8) // 10000000
}

```

Пример использования простой битовой маски:

```

type Bits uint8

const (
    F0 Bits = 1 << iota // 0b00_000_001 == 1
    F1                   // 0b00_000_010 == 2
    F2                   // 0b00_000_100 == 4
)

```

```
func Set(b, flag Bits) Bits    { return b | flag }
func Clear(b, flag Bits) Bits { return b &^ flag }
func Toggle(b, flag Bits) Bits { return b ^ flag }
func Has(b, flag Bits) bool    { return b&flag != 0 }

func main() {
    var b Bits

    b = Set(b, F0)
    b = Toggle(b, F2)

    for i, flag := range [...]Bits{F0, F1, F2} {
        println(i, Has(b, flag))
    }
    // 0 true
    // 1 false
    // 2 true
}
```

Как работает `init`?

В Go есть предопределенная функция `init()`. Она выделяет фрагмент кода, который должен выполняться перед всеми другими частями пакета. Этот код будет выполняться сразу после импорта пакета. Таким образом, хотя в Go нет конструкторов в классическом понимании, но `init()` предлагает возможность выполнять необходимую начальную настройку.

Пара важных особенностей:

1. **Автоматический вызов:** `init()` вызывается автоматически перед вызовом `main()` и не требует явного вызова.
2. **Использование:** `init()` можно использовать для инициализации глобальных переменных, проверки или установки конфигурации, установки соединений с базами данных и других целей.

3. **Несколько функций** `init()` : в одном пакете можно иметь несколько `init()` . Они будут вызваны в том порядке, в котором объявлены в файле.
4. В случае зависимостей между пакетами, функции `init()` из импортированных пакетов выполняются перед функцией `init()` из основного пакета.

Также функция `init()` используется для автоматической регистрации одного пакета в другом (например, так работает подавляющее большинство “драйверов” для различных СУБД, например [go-sql-driver/mysql/driver.go](https://github.com/go-sql-driver/mysql)).

Хотя использование `init()` и является довольно полезным, но часто оно затрудняет чтение/понимание кода, и (почти) всегда можно обойтись без неё, поэтому необходимость её использования — всегда очень большой вопрос.

Сделать конвейер чисел

Задача: даны 2 канала. В первый пишутся числа. Нужно, чтобы числа читались из первого по мере поступления, что-то с ними происходило (допустим, возводились в квадрат) и результат записывался во второй канал.

Решается довольно прямолинейно — запускаем две горутины. В одной пишем в первый канал. Во второй читаем из первого канала и пишем во второй. Главное — не забыть закрыть каналы, чтобы ничего нигде не заблокировалось.

```
package main

import (
    "fmt"
)

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go func() {
        for x := 0; x <= 10; x++ {
            naturals <- x
        }
    }
```

```
    close(naturals)
}()

go func() {
    for x := range naturals {
        squares <- x * x
    }
    close(squares)
}()

for x := range squares {
    fmt.Println(x)
}
}
```

Прерывание `for/switch`

Что произойдёт в следующем примере, если `f()` вернёт `true` ?

```
for {
    switch f() {
    case true:
        break
    case false:
        // некое действие
    }
}
```

Очевидно, будет вызван `break` . Вот только прерван будет `switch` , но не цикл `for` .

Чтобы исправить ситуацию, и прервать именно цикл `for` можно использовать именованный (labeled) цикл и вызывать `break` с этой меткой. Например, так:

```
loop:
    for {
```



```
switch f() {
case true:
    break loop
case false:
    // некое действие
}
}
```

Дженерики — это про что?

Дженерики, или **обобщения** — это средства языка, позволяющего работать с различными типами данных без изменения их описания.

В версии 1.18 появились дженерики (вообще-то они были и ранее, но мы не могли их использовать в своём коде — вспомним функцию `make(T type)`).

Дженерики позволяют объявлять (описывать) универсальные методы, т.е. в качестве параметров и возвращаемых значений указывать не один тип, а их наборы.

Появились новые ключевые слова:

- `any` — аналог `interface{}` , можно использовать в любом месте (`func do(v any) any` , `var v any` , `type foo interface { Do() any }`)
- `comparable` — интерфейс, который определяет типы, которые могут быть сравнены с помощью `==` и `!=` (переменные такого типа создать нельзя — `var j comparable` будет вызывать ошибку)

И появилась возможность определять интерфейсы, которые можно будет использовать в параметризованных функциях и типах (переменные такого типа создать нельзя — `var j Int` будет вызывать ошибку):

```
type Int interface {
    int | int32 | int64
}
```

Если добавить знак `~` перед типами то интерфейсу будут соответствовать и производные типы, например `myInt` из примера ниже:

```
type Int interface {  
    ~int | ~int32 | ~int64  
}  
  
type myInt int
```

Разработчики `golang` создали для нас уже [готовый набор интерфейсов \(пакет `constraints`\)](#), который очень удобно использовать.

Написать `WorkerPool` с заданной функцией

Нам нужно разбить процессы на несколько горутин — при этом не создавать новую горутины каждый раз, а просто переиспользовать уже имеющиеся. Для этого создадим канал с *джобами* и результирующий канал. Для каждого *воркера* создадим горутины, который будет ждать новую *джобу*, применять к ней заданную функцию и *пулять* ответ в результирующий канал (*сорри за мой французский*).

В целом, вот и всё:

```
package main  
  
import (  
    "fmt"  
)  
  
func worker(id int, f func(int) int, jobs <-chan int, results chan<- int) {  
    for j := range jobs {  
        results <- f(j)  
    }  
}
```

```
func main() {
    const numJobs = 5
    jobs := make(chan int, numJobs)
    results := make(chan int, numJobs)

    multiplier := func(x int) int {
        return x * 10
    }

    for w := 1; w <= 3; w++ {
        go worker(w, multiplier, jobs, results)
    }

    for j := 1; j <= numJobs; j++ {
        jobs <- j
    }
    close(jobs)

    for i := 1; i <= numJobs; i++ {
        fmt.Println(<-results)
    }
}
```

Что из себя представляет буферизованный и небуферизованный файловый ввод-вывод?

Буферизованный файловый ввод-вывод — это использование буфера для временного хранения данных перед чтением или записью. Таким образом, вместо того чтобы читать файл побайтово, мы читаем сразу множество данных. Мы помещаем данные в буфер и ожидаем, пока кто-нибудь их не прочитает желаемым образом.

Небуферизованный файловый ввод-вывод: буфер для временного хранения данных не используется перед их фактическим чтением или записью, что может повлиять на производительность.

Когда какой использовать? При работе с критически важными данными небуферизованный файловый ввод-вывод, как правило, является лучшим выбором, поскольку буферизованное чтение может привести к использованию устаревших данных, а небуферизованная запись — к потере данных в случае сбоя. Однако в большинстве случаев однозначного ответа на этот вопрос нет.

Что насчёт линтеров?

Линтер — это статический анализатор кода. При помощи линтера можно отлавливать ошибки.

Рассмотрим вот такой код:

```
package main

import "fmt"

func main() {
    i := 0
    if true {
        i := 1
        fmt.Println(i)
    }
    fmt.Println(i)
}
```

Используя линтер `vet`, встроенный в набор инструментов Go, а также `shadow`, мы можем обнаружить затенённые переменные.

Устанавливаем `shadow`:

```
go install
golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow
```

...связываем его с `vet` и запускаем:

```
go vet -vettool=$(which shadow)
```

...получаем такой вывод — линтер нашёл затенённую переменную, и мы можем это исправить.

```
./main.go:8:3:
  declaration of "i" shadows declaration at line 6
```

В общем, использование линтеров помогает сделать код более надежным и обнаружить потенциальные ошибки, поэтому нужно выбрать подходящий линтер и использовать его чаще.

Уже давно на все случаи жизни существует [golangci-lint](#), который является универсальным решением, объединяющим множество линтеров в “одном флаконе”. Удобен как для запуска локально, так и на CI.

Что из себя представляет пакет `semaphore` в Go?

Семафор — это конструкция, которая может ограничивать или контролировать доступ к общему ресурсу. В контексте Go, семафор может ограничить доступ горутин к общему ресурсу, но первоначально семафоры использовались для ограничения доступа к потокам.

Семафоры могут иметь веса, которые задают максимальное количество потоков или горутин, получающих доступ к ресурсу.

Процесс поддерживается с помощью методов `Acquire()` и `Release()`, определенных следующим образом:

```
func (s *Weighted) Acquire(ctx context.Context, n int64) error
func (s *Weighted) Release(n int64)
```

Второй параметр `Acquire()` определяет вес семафора.

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
    "golang.org/x/sync/semaphore"
)

var Workers = 4
```

Эта переменная определяет максимальное количество горутин, которые могут быть выполнены данной программой.

```
var sem = semaphore.NewWeighted(int64(Workers))
```

Здесь мы определяем семафор с весом, идентичным максимальному количеству горутин, которые могут выполняться одновременно. Это означает, что получать семафор одновременно могут не более чем `Workers` горутин.

```
func worker(n int) int {
    square := n * n
    time.Sleep(time.Second)
    return square
}
```

Функция `worker()` выполняется как часть горутин. Однако поскольку мы используем семафор, нет необходимости возвращать результаты в канал.

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need #jobs!")
        return
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Считываем количество заданий, которые хотим запустить.

```
// где хранить результаты
var results = make([]int, nJobs)
// требуется для Acquire()
ctx := context.TODO()

for i := range results {
    err = sem.Acquire(ctx, 1)
    if err != nil {
        fmt.Println("Cannot acquire semaphore:", err)
        break
    }
}
```

Получаем семафор столько раз, сколько заданий определено `nJobs`. Если `nJobs` больше, чем `Workers`, то вызов `Acquire()` будет заблокирован и дождется вызовов `Release()` для разблокировки.

```
go func(i int) {
    defer sem.Release(1)
    temp := worker(i)
    results[i] = temp
}(i)
}
```

Запускаем горутины, которые выполняют эту задачу, и записываем результаты в срез `results`. Поскольку каждая горутина записывает данные в свой элемент среза, никаких *race condition* нет.

```
err = sem.Acquire(ctx, int64(Workers))
if err != nil {
    fmt.Println(err)
}
```

Получаем все токены таким образом, чтобы вызов `sem.Acquire()` блокировался до тех пор, пока все рабочие процессы/горутины не завершат работу. Функционально это похоже на вызов `Wait()`.

```
for k, v := range results {
    fmt.Println(k, "->", v)
}
}
```

Как-то так используется семафор на практике.

Преимущества и недостатки ORM по сравнению с использованием встроенных возможностей для SQL?

Преимущества ORM:

1. Удобство и скорость разработки: ORM позволяет взаимодействовать с базой данных, используя объектно-ориентированный подход, что часто упрощает и ускоряет процесс разработки.
2. Безопасность: ORM может помочь избежать некоторых распространенных уязвимостей за счет использования встроенных механизмов защиты.

3. Независимость от базы данных: ORM обеспечивает абстракцию, которая позволяет легче переходить между различными СУБД, не изменяя большую часть кода приложения.
4. Упрощение рефакторинга и поддержки: поскольку логика доступа к данным централизована, вносить изменения и поддерживать приложение становится проще.

Недостатки ORM:

1. Производительность: ORM может быть менее эффективным по сравнению с оптимизированными вручную SQL-запросами, особенно в сложных сценариях.
2. Сложность: ORM может добавлять дополнительный уровень сложности, который может быть излишним для простых приложений или простых запросов.
3. Ограничения: некоторые ORM могут ограничивать способность разработчика использовать все функции и возможности конкретной СУБД.
4. Кривая обучения: для эффективного использования ORM требуется время на изучение его особенностей и лучших практик.

Примеры ORM для Go: gorm, Beego ORM, SQLBoiler и другие.

Реализовать обход ссылок из файла

Задача: дан некоторый файл, в котором содержатся HTTP-ссылки на различные ресурсы. Нужно реализовать обход всех этих ссылок, и вывести в терминал OK в случае 200-го кода ответа, и Not OK в противном случае.

Что ж, так будет выглядеть наивный вариант (читаем файл в память, и итерируем слайс ссылок):

```
package main
```

```
import (  
    "bufio"  
    "context"
```

```
"net/http"
"os"
"strings"
"time"
)

func main() {
    if err := run(); err != nil {
        println(err.Error())

        os.Exit(1)
    }
}

func run() error {
    var ctx = context.Background()

    // открываем файл
    f, err := os.Open("links_list.txt")
    if err != nil {
        return err
    }
    defer func() { _ = f.Close() }()

    // читаем файл построчно
    var scan = bufio.NewScanner(f)
    for scan.Scan() {
        var url = strings.TrimSpace(scan.Text())

        if ok, fetchErr := fetchLink(ctx, http.MethodGet, url); fetchErr !=
            return fetchErr
        } else {
            if ok {
                println("OK", url)
            } else {
                println("Not OK", url)
            }
        }
    }
}
```

```
// проверяем сканер на наличие ошибок
if err = scan.Err(); err != nil {
    return err
}

return nil
}

// объявляем HTTP клиент для переиспользования
var httpClient = http.Client{Timeout: time.Second * 5}

func fetchLink(ctx context.Context, method, url string) (bool, error) {
    // создаём объект запроса
    var req, err = http.NewRequestWithContext(ctx, method, url, http.NoBody)
    if err != nil {
        return false, err
    }

    // выполняем его
    resp, err := httpClient.Do(req)
    if err != nil {
        return false, err
    }

    // валидируем статус код
    if resp.StatusCode == http.StatusOK {
        return true, nil
    }

    return false, nil
}
```

Файл со списком ссылок (links_list.txt):

```
https://www.yahoo.com/foobar
https://stackoverflow.com/foobar
https://blog.iddqd.uk/
```

```
https://google.com/404error
```

```
https://ya.ru/
```

```
https://github.com/foo/bar
```

```
https://stackoverflow.com/
```

Запускаем код (`go run .`), видим результат:

```
Not OK https://www.yahoo.com/foobar
```

```
Not OK https://stackoverflow.com/foobar
```

```
OK https://blog.iddqd.uk/
```

```
Not OK https://google.com/404error
```

```
OK https://ya.ru/
```

```
Not OK https://github.com/foo/bar
```

```
OK https://stackoverflow.com/
```

Поменять местами значения переменных без промежуточной

Во многих других языках над этой задачей придется подумать (ну кроме питончика), в Go же реализовать ее не супер сложно:

```
package main
import "fmt"

func main() {
    fmt.Println(swap())
}

func swap() []int {
    a, b := 15, 10
    b, a = a, b
    return []int{a, b}
}
```

Вот собственно и всё.

Сумма квадратов чисел

Задача: реализовать функцию `SumOfSquares`, получающую целое число `c` и возвращающую сумму всех квадратов между 1 и `c`. Потребуется использовать инструкции `select`, горютины и каналы. Например, ввод 5 приведет к возвращению 55, потому что $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$.

В качестве отправной точки можно взять этот код:

```
package main
import "fmt"

func SumOfSquares(c, quit chan int) {
    // ваш код
}

func main() {
    mychannel := make(chan int)
    quitchannel := make(chan int)
    sum := 0
    go func() {
        for i := 0; i < 6; i++ {
            sum += <-mychannel
        }
        fmt.Println(sum)
    }()
    SumOfSquares(mychannel, quitchannel)
}
```

Ну а конечное решение может выглядеть так:

```
package main
import "fmt"
```

```
func SumOfSquares(c, quit chan int) {  
    y := 1  
    for {  
        select {  
        case c <- (y*y):  
            y++  
        case <-quit:  
            return  
        }  
    }  
}
```

```
func main() {  
    mychannel := make(chan int)  
    quitchannel := make(chan int)  
    sum := 0  
  
    go func() {  
        for i := 1; i <= 5; i++ {  
            sum += <-mychannel  
        }  
        fmt.Println(sum)  
        quitchannel <- 0  
    }()  
  
    SumOfSquares(mychannel, quitchannel)  
}
```

Рассмотрим функцию `SumOfSquares`. Сначала на строке 4 мы объявляем переменную `y`, после чего переходим к циклу `For-Select`. В инструкциях `select` прописано два кейса. `case c <- (y*y)` служит для отправки квадрата `y` по каналу `c`, который принимается в горутине, созданной в основной рутине. `case <-quit` служит для получения сообщения из основной рутины, которое вернется из функции.

Как можно обработать JSON-данные в Golang?

Golang предоставляет встроенный пакет `encoding/json` для работы с JSON-данными. А вот пример чтения и записи JSON-данных:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    // Преобразование структуры в JSON
    person := Person{Name: "John Doe", Age: 30}
    jsonData, _ := json.Marshal(person)
    fmt.Println(string(jsonData))

    // Чтение JSON в структуру
    var decodedPerson Person
    json.Unmarshal(jsonData, &decodedPerson)
    fmt.Println(decodedPerson.Name, decodedPerson.Age)
}
```

В приведенном примере структура `Person` представляет объект с полями `Name` и `Age`. Функция `json.Marshal()` используется для преобразования структуры в JSON-строку, а `json.Unmarshal()` – для чтения JSON-строки и преобразования ее в структуру.

Пакет `encoding/json` предоставляет мощные и гибкие инструменты для работы с JSON-данными в Golang.

Как реализовать rate limiter на Go?

Rate limiter — это механизм для контроля частоты доступа к определенному ресурсу. В Go для его реализации можно использовать пакет `rate` из стандартной библиотеки.

Один из распространенных подходов к ограничению скорости — использование алгоритма `token bucket`, который позволяет добавлять фиксированное количество токенов в пакет с фиксированной скоростью. Когда токен извлекается из бакета, скорость добавления токенов временно уменьшается.

Пакет `rate` предоставляет функцию `NewLimiter()`, которую можно использовать для создания нового `token bucket rate limiter`. Например:

```
limiter := rate.NewLimiter(rate.Limit(100), 100)
```

Затем можно использовать метод `limiter.Allow()`, чтобы проверить, доступен ли токен перед выполнением задачи:

```
if limiter.Allow() {  
  
    // выполнение задачи  
  
} else {  
  
    // превышен лимит скорости  
  
}
```

В качестве альтернативы можно использовать метод `limiter.Wait()`, чтобы подождать, пока токен станет доступен:

```
limiter.Wait()  
  
// выполнение задачи
```

Также можно использовать метод `limiter.Reserve()`, чтобы зарезервировать токен заранее и выполнить задачу позже.

Заключение

Что ж, это были 100 вопросов/заданий, которые с большой вероятностью могут попасться на собесе на джуниора Go-разработчика. Если пролистать их хотя бы по диагонали перед собесом и освежить в голове большую часть вопросов, вероятность не завалить существенно повышается — проверено)

Будет интересно послушать свежесобеседовавшихся в комментах, объявляю рубрику «Самый странный вопрос на собесе». Если есть какие-то интересные/полезные дополнения к ответам — тоже пишите.

Ну и напоследок, неплохой канал с массой годноты по Go — [@Golang_google](#)

И вот — [целая подборка отличного контента](#): тут и описание продвинутых библиотек Go, и масса полезных в работе инструментов



👁 Просмотры: 2 051

+1

Похожие записи



Полный пошаговый гайд по созданию Telegram бота на Go с нуля в 2024 году.

20.07.2024

22 лучших репозитория на GitHub для новичков, изучающих Python

18.07.2024



Лучшие бесплатные курсы и книги по PHP для изучения языка с нуля в 2024 году

13.07.2024

Ответить

Ваш адрес email не будет опубликован. Обязательные поля помечены *

Оставьте свой комментарий

☐ Сохранить моё имя, Email и адрес сайта в этом браузере для последующих комментариев.

Отправить комментарий

Все права защищены © 2024 UPROGER | Программирование