BaseFunctionsGo_RU

Вступление

В этом гайде вы сможете получить основные зания по Go. Причиной написания гайда была сложность и запутаность которую я заметил в другий гайдах и руководствах. Безусловно я не эксперт в Go, но при этом имею какой-то опыт и думаю, что могу помочь этими знаниями другим :)

Полезные ссылки

Так как этот гайд достаточно субъективный я не уверен что захвачу все темы и не уверен что у нас не появяться вопросы после прочтения, по-этому в этом блоке прикреплю хорошие ссылки который можно использовать в таких случаях.

В первую очередь вы можете написать мне на почты и я попробую вам помочь:

boyarkin.gleb@gmail.com

source.boar@gmail.com

Так же почты продублированы у меня в профиле.

Кроме меня вы можете воспользоваться:

https://metanit.com/go/tutorial

Эти источники гарантирую что являються проверенными, сам по ним учился, но как по мне многие вещи там расписаны слишком сложными словами.

Коротко про Go

Go — очень молодой статический язык программирования. Создан он был для того что б объеденить в себе простоту и скорость, что в принципе он сделал. Отраслями использования чаще всего есть малые приложения, микросервисы и иногда Web.

Синтаксис

Синтаксис Go в большинстве своём явяеться C-подобным, но со своими мелкими отличиями, с которыми мы будем сталкиваться по ходу.

Если проходиться по основным правилам синтаксиса то:

- символа для окончания строки нету
- `{}` – сиволы для открытия и закрытия блоков кода
- `camelCase` – стиль именования переменный
- `PascalCase` – стиль именования функций, методов и структур
- `//` – символ для комментария одной строки
- `/* */` – символы для комментирования блока кода

Структура программы

Структура программы Go состоит из 2 обязательных вещей:

1. Объявление имени пакета

```
package /*имя пакета*/
```

Если ваша программа не является библиотекой то пакет объявляется как `main`

```
package main
```

2. Функция `main` – функция входа в программу

```
func main() {

}
```

Работа с консолью

Для работы с консолью используется модуль `fmt`

`fmt.Print(/*текст сообщения*/)` – вывод текста в консоль

`fmt.Println(/*текст сообщения*/)` – вывод текста в консоль с добавлением переноса на новую строку

`fmt.Printf(/*маска*/, /*переменная 1*/, /*переменная 2*/, ... /*переменная n*/)` – форматированый вывод текста в консоль

`fmt.Scan(/*ссылка на переменную куда записать результат*/)` – ввод текста с консоли

Переменные

Как я думаю вам известно, или станет известо сейчас, по факту весь язык программирования держиться на переменных и операциях над ними.

В Go переменная — это указатель на область или ячейку памяти в которой хранится какое то зничени. Факт про память пока можно просто положить в голову, к нему мы прийдем потом.

Так же Go является статическим строго и явно типизированым языком, по-этому при объявлении переменных нужно так же передавать тип или сразу передавать значение тогда тип будет определён автоматически.

Для объявления переменных используются ключевое слово `var` .

Пример объявления:

```
var a1 string = "Hello"
var a2 = 1
```

Так же в Go присутсвуют константы. Константы – обычные переменные, за исключением чего не могут быть изменены в будующем после объявления. Указание типа константы является не обязательным.

Для объявления констант используются ключевое слово `const`.

Пример объявления:

```
const a1 string = "Hello"
const a2 = 1
```

Так же переменные можно объявлять кратко: без использования ключевого слова `var`. Вместо этого нужно просто использовать символ `:=`.

Пример объявления:

```
a1 := "Hello"
```

При именовании переменных, как указано было выше, используется стиль camelCase.

Что ж в него входит?

camelCase:

- первая буква – нижнего регистра
- каждая первая буква следуещего слова – верхнего регистра

Так же стоит уточнить что Go чувствителен к регистру, то есть в нем `a` и `A` это разные имена.

Типы данных

Тип данных — типы того, какого типа значение может хранить переменная.

Встроенные типы данных в Go:

- `bool` – логический тип данных, хранит 2 вида значений: `true` или `false`
- `int8/int16/int32/int64` – числовые типы данных, которые могут хранить целые числа размерностью 8, 16, 32 и 64 бита соответственно
- `int` – числовой тип данных который в зависимости от платформы хранит целое число размерностью 32 или 64 бита, то есть эквивалентно `int32` или `int64`
- `uint8/uint16/uint32/uint64` – числовые типы данных, которые могут хранить целые положительные числа размерностью 8, 16, 32 и 64 бита соответственно
- `uint` – числовой тип данных который в зависимости от платформы хранит целое положительное число размерностью 32 или 64 бита, то есть эквивалентно `uint32` или `uint64`
- `byte` – синоним типа `uint8`
- `rune` – синоним типа `int32`
- `float32/float64` – числовые типы данных, которые могут хранить дробные числа размерностью 32 и 64 бита соответственно

- `complex64/complex128` – числовые типы данных, которые могут хранить комплексные числа с основами `float32` и `float64` соответственно
- `string` – строчный тип данных, хранит строку текста почти неограниченого размера
- `nil` – ключевое слово, которое означает что нулевое или "пустое" значение

Указатели

Указатели представляют собой объекты, значением которых служат адреса других объектов (например, переменных). Указатель определяется как обычная переменная, только перед типом данных ставится символ звездочки `*`

***Пример указателя на тип int*:**

```
var a *int
```

Для получения адресса указателя или любой переменной используется символ `&`

Операторы

Операторы присваивания

`=` – присваивание

`+=` – присваивание со сложением

`-=` – присваивание с вычитанием

`*=` – присваивание с умножением

`/=` – присваивание с делением

`%=` – присваивание с делением с остатком

`++` – инкремент, тоже самое что `+= 1`

`--` – декремент, тоже самое что `-= 1`

`<<=` – присваивание с левым сдвигом

`>>=` – присваивание с правым сдвигом

`&=` – присваивание с побитовым `И`

`|=` – присваивание с побитовым `ИЛИ`

`^=` – присваивание с побитовым исключающим `ИЛИ`

Логические операторы

`==` – равенство

`!=` – неравенство

`>` – больше

`>=` – больше или равно

`<` – меньше

`<=` – меньше или равно

`&&` – логическое `И`, то есть оба условия в конструкции `(условие 1)&&(условие 2)` доожны быть `true`

`||` – логическое `ИЛИ`, то есть хотя б одно условие в конструкции `(условие 1)||(условие 2)` доожно быть `true`

Арифметические операторы

`+` – плюс

`-` – минус

`*` – умножение

`/` – деление

`%` – деление с остатком

Побитовые операторы

`&` – побитовое `И`

`|` – побитовое `ИЛИ`

`^` – побитовое исключающее `ИЛИ`

`~` – побитовое `НЕ`

`<<` – сдвиг влево

`>>` – сдвиг вправо

Условные конструкции

Переменные это конечно хорошо, но надо ж как-то с ними работать и вот условные конструкции это один из способов прописывания логики программы.

if...else

Самая известная и часто используемая конструкция.

В `if` передаёться условие, если условие истинное то выполняеться то, что в блоке `if` в ином случае то, что в блоке `else`

Синтаксис выглядит примерно так:

```
if /*условие: или логическое выражение или переменная типа bool*/ {
    // действие если условие == true
} else {
```

```
    // действие если условие == false
}
```

Так же в эту конструкцию можно добавить `else if`

Пример применения:

```
if /*условие 1: или логическое выражение или переменная типа bool*/ {
    // действие если условие 1 == true
} else if /*условие 2: или логическое выражение или переменная типа bool*/ {
    // действие если условие 1 == false, но условие 2 == true
} else {
    // действие если условие 1 == false и условие 2 == false
}
```

Таких конструкций `else if` можно добавлять любое количество раз

Синтаксис должен быть именно таким, другие варианты расположения скобок является неправильным и не воспринимается компилятором

switch..case...default

Так же условная конструкция которая укорачивает код который возможно написать через `if...else if...else`

`switch...case...default` отличается тем, что мы в switch передаём какою либо переменную, а потом конструкция сравнивает эту переменную с каждым значением `case` и если никакого совпадения с `case` не нашлось выполняеться блок `default`

Синтаксис выглядит примерно так:

```
switch /*переменная которую сравниваем*/ {
    case /*значение 1*/:
        // действие если совпало значение 1
    case /*значение 2*/:
        // действие если совпало значение 2
    ...
    default:
        // действие если никакое значение не совпало
}
```

Циклы

for

Вид цикла с действием которое выполняеться до начала обхода, условием выполнения и действием которое выполняеться в конце каждого прохода

Синтаксис выглядит примерно так:

```
for /*действие в начале*/; /*условие выполнения*/; /*действие в конце каждого прохода*/ {
    // действие пока условие true
}
```

Если действия в начале и в конце не нужны то можно использовать конструкцию только с условием

```
for /*условие выполнения*/ {
    // действие пока условие true
}
```

Ключевые слова `break` и `continue`

`break` используеться для досрочного выхода из цикла, то есть завершение работы цикла вне зависимости от условия

Пример использования:

```
for i := 0; i < 10; i++ {
    if i%3 == 0 { // если число делится на 3 без остатка
        break
    }
    fmt.Println(i)
}
```

То есть цикл выполнялся пока не наткнулся на первое число которое делится на 3

`continue` используеться для досрочного прекращения прохода цикла и перехода на следующий проход, то есть используется когда вам надо что б все после его вызова не выполнялось в это проходе, а выполнялось сразу сделующий проход

```
for i := 0; i < 10; i++ {
    if i%3 == 0 { // если число делится на 3 без остатка
        continue
    }
    fmt.Println(i)
}
```

То есть для чисел которые деляться на 3 не выполнялся `fmt.Println(i)` , а сразу выполнялся следующий прогон цикла

Практические задания на данных знаниях

Для лучшего закрепления предлогаю выполнить несколько простых задач, которые проверят, насколько вы поняли этот уровень материала, все знания которые могут быть нужны для выполнения задач есть выше.

Все решения задач будут в папке "Задачи" и будут пронумерованы, но не рекомендую вам ими пользваться пока у вас самих не получится

1. Напишите цикл который выведет в консоль все числа от 1 до 100 включительно которые деляться на 5 или на 7 (`n%/*число*/ == 0` – условие делимости `n` на какое либо число)
2. Напишите цикл который выведет в консоль все чётные числа от 1 до 100 включительно
3. Напишите цикл который выведет в консоль все числа от 1 до 100 включительно которые содержат 1

Фукнции

Функции — ещё один важный инструмент из программирования. Если простыми словами это объединение какого либо участка кода для его дальнейшего вызова. В Go функции это объект и с ними можно работать как с объектом, то есть записывать в переменные, передавать как аргумент, писать методы и так далее.

Каждая функция в любом языке программирования имеет:

- Имя
- Список аргументов (параметров) – переменные, значения которых задаёться при вызове функции. Объявляються они в `()` через запятую. Так же при вызове их значения указываются при вызове так же в `()`. В блоке кода нашей функции они отыгрывают роль обычных переменных с блочной областью видимости, только не нуждаються в объявлении, так как объявляються при вызове функции
- Блок кода который будет выполняться при вызове. Записываеться в `{}`

Синтаксис выглядит примерно так:

```
func /*имя функции*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/) /*тип возвращаемого значения(если нужен)*/{
    // блок испольняемого кода
}
```

`func` – ключевое слово которое означает объявление функции

Для вызова функции достаточно написать её имя и `()` в которых можно перечислить аргументы если они были объявлены

Пример функции:

```
func f(a1 int, a2 int, a3 int){
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
}

f(1, 2, 3)
```

Фукнции и процедуры

Если отходить от темы Go, где синтаксической разницы между этими понятиями по факту нету, но это очень важный вопрос.

В чем же разница?

Функция — блок кода, который результате своего выполнения возвращет какое либо занчение.

Процедура — функция, которая не возвращает значения

За возвращение значения в Go отвечает ключевое слово `return`

Давайте расмотрим разницу на примерах:

```
func fnc(a1 int, a2 int, a3 int, a4 int) int { // функция
    fmt.Println(a1)
    fmt.Println(a2)
    fmt.Println(a3)
    return a4
}

func proc(b1 int, b2 int, b3 int, b4 int) { // процедура
    fmt.Println(b1)
    fmt.Println(b2)
    fmt.Println(b3)
}

a := fnc(1, 2, 3, 4) // == 4(a4)
b := proc(1, 2, 3, 4) // == ошибка
```

Что же случилось?

При объявлении **функции** мы использовали `return` по-этому вызов функции можно будет присвоить переменной и он будет иметь значение которое мы укзали в `return`

Упорядочиные типы данных

Переменные — это конечно хорошо, но иногда приходиться хранить много каких либо данных объеденив их в какую то одну структуру. Для таких вещей придуманы массивы и словари.

Массивы

Массивы — упорядоченый набор значений, которые можно получить обращаясь по индексу.

Массив в Go это участок памяти с фиксированой длиной.

Так же рассмотрим способ его объявления:

```
var a1[5]int = [5]int{1, 2, 3, 4, 5}
var a2 = [5]int{1, 2, 3, 4, 5}
var a3 := [5]int{1, 2, 3, 4, 5}
```

Для лучшего понимания воспользуемся картинкой выше. Как видно на ней массив — это грубо говоря набор "ячеек" со значениями к которым параллельно прикриплены индексы по которым можно получить значение которому они соответствуют.

Для получения значения или присваивания по индексу достаточно после имени массива в `[]` указать индекс

```
a := arr[0] // получение
arr[0] = 1 // присваивание
```

!!!Индесация в языках программирования начинается с 0!!!

Срезы

Срезы это последовательности элементов одного типа переменной длины. В отличие от массивов длина в срезах не фиксирована и динамически может меняться, то есть можно добавлять новые элементы или удалять уже существующие.

Так же рассмотрим способ его объявления:

```
var a1[]int = []int{1, 2, 3, 4, 5}
var a2 = []int{1, 2, 3, 4, 5}
var a3 := []int{1, 2, 3, 4, 5}
```

Добавление элементов

Для добавления элементов используется функция `append(/*имя массива*/, /*элемент добавления*/)`

Взятие среза

Для взятие среза используеться конструкция `/*имя массива*/[/*начальный индекс*/:/*конечный индекс*/]`

Длина среза

Для получения длины массива используеться функция `len(/*имя массива*/)`

Перебор массива/среза

for

```
for i := 0; i < /*длина массива/среза*/; i++ {
    // элемент массива/среза arr[i]
```

```
}
```

for...range

```
for i, a := range arr {
    // элемент массива/среза arr[i]
    // элемент массива/среза a
}
```

Отображения

Иногда возникают ситуации, когда надо что б индексами были не цифры, а что-то своё. Для такого были созданы отображения.

Отображения – это массив с самоуказаными индексами.

Синтаксис объявления:

```
mp := map[/*тип ключей*/]/*тип значений*/ {
    "/*название индекса 1*/": /*занчение 1*/,
    "/*название индекса 2*/": /*занчение 2*/,
    ...
    "/*название индекса N*/": /*занчение N*/}
```

Практические задания на данных знаниях

4. Придумайте как перевернуть строку в обратном порядке
5. Найдите способ строку `"abcdef"` превратить в `"cdefgh"`
6. Напишите функцию которая будет возвращать строку созданую по правилу `camelCase` (разделитель должен передаваться вторым аргументом)

   Пример:

   `hello world` -> `helloWorld`
7. Создайте функцию для заполнения массива отображений, которые содержат информацию про человека

   Структура:

   ```
   people := []map[string]string{} // основной массив
   /* структура словаря
   {
       "firstName": "",
       "secondName": "",
       "age": "",
       "country": "",
       "city": ""
   }
   */
   ```

8. *Придумайте как можно отсортировать массив

"*" – сложная задача

Структуры, методы и интерфейсы

Структуры

Структуры – типы данных, созданые пользователем для представления каких либо нестандартных объектов

Для объявления структур используются ключевые слова `type` и `struct`

Синтаксис выглядит примерно так:

```
type /*имя структуры*/ struct {
    /*поля структуры*/
}
```

Каждое поле имеет название и тип данных, как переменная

Инициализация структур может выглядить примерно так:

```
var s /*имя структуры*/ = /*имя структуры*/{/*значения, которые передаются полям*/}
```

Обращение к полям структуры выполняется через `.`

Методы

Методы – функции, связаные с определёнными типами(структурами)

Синтаксис выглядит примерно так:

```
func (/*имя переменной которая будет означать тип*/ /*имя типа*/) /*имя
метода*/(/*аргумент1*/, /*аргумент2*/.../*аргументN*/) /*тип возвращаемого
значения(если нужен)*/{
    // блок испольняемого кода
}
```

Вызов методов выполняется как получение полей структуры, через `.`, единственное что добавляются скобки с аргументами

Интерфейсы

Интерфейсы представляют абстракцию поведения других типов. Интерфейсы позволяют определять функции, которые не привязаны к конкретной реализации. То есть интерфейсы определяют некоторый функционал, но не реализуют его

Синтаксис выглядит примерно так:

```
type /*имя интерфейса*/ interface {
    /*методы интерфейся*/
```

```
}
```

В Go присутсвует Утиная типизация("Всё что крякает – всё утка") из-за чего интерфейсы не требуют инициализации. Все объекты структур с методами которые соответсвуют методам интерфейса автоматически являются объектами этого интерфейса

Практические задания на данных знаниях

9. Напишите структуру человека где будет хранится его имя, возраст, страна и город
10. Реализуйте методы для этой структуры с использованием интерфейса

Всё!

В принцепе это всё! Я дал вам всё что хотел. Надеюсь этот гайд помог вам.

Так же у меня есть гайд посложнее с более сложными темами, но он больше просто сборник решений, но кому интересно можете почитать: https://github.com/s0urcedev/AdditionalFunctions

Всем спасибо! Надеюсь ещё увидимся :)