GeekBrains

Go. Уровень 1

Сложные типы данных: массивы, слайсы и мапы

На этом уроке

Научимся работать с тремя самыми распространёнными типами данных в computer science: массивами, слайсами и мапами.

Оглавление

Введение

Для прикладного программирования хранение групп похожих объектов — стандартная задача. Резонно ввести для таких задач специальные структуры данных, называемые коллекциями. Их основное назначение — хранение объектов одного типа с различными особенностями доступа к ним.

В двадцатом веке на заре computer science было разработано множество коллекций и алгоритмов для работы с ними. Поэтому сейчас нет нужды реализовывать вручную эти алгоритмы и типы данных. Зачастую они есть в стандартной библиотеке языка или вовсе встроены в синтаксис языка.

Язык Go не исключение. В его синтаксис встроены три самые распространённые в прикладном программировании коллекции:

- массивы — array;
- слайсы — slice, также называют срезами;
- мапы — map, также называют словарями, ассоциативными массивами, хешами, картами, отображениями.

Массивы (array)

Массив — это структура данных, предназначенная для хранения множества данных одного типа. Особенность массива в Go — его размер не меняется на протяжении работы всей программы. Рассмотрим массивы на примерах.

Допустим, по логике нашей программы нам необходимо считать 5 чисел из стандартного потока ввода, выбрать из них наименьшее и напечатать. Мы бы могли сделать это с помощью обычного цикла. Например, вот так:

```
package main

import (
   "bufio" // содержит функции и типы данных для работы с потоками
ввода/вывода
   "fmt" // содержит функции для форматированного вывода
   "math" // содержит популярные математические функции и константы
   "os" // содержит функции и константы для взаимодействия с операционной
системой
   "strconv" // содержит функции для конвертации строк в различные типы
)

func main() {
   // bufio.NewScanner создает экземпляр сканера ввода.
   // Сканер позволяет считывать содержимое ввода по строкам.
   // Документация: https://golang.org/pkg/bufio/#Scanner
   // os.Stdin - стандартный поток ввода.
   // Вместо него можно использовать, например, файл.
   // Документация: https://golang.org/pkg/os/#pkg-variables
   scanner := bufio.NewScanner(os.Stdin)
   // math.MaxInt64 содержит максимальное значение типа int64
   // это число можно найти так: (2^63 - 1)
   var minNum int64 = math.MaxInt64

   for i := 0; i < 5; i++ {
       if scanner.Scan() { // проверяет, что следующая строка существует
           line := scanner.Text() // возвращает следующую строку
           // strconv.ParseInt конвертирует строку в число
           // 10 - система счисления. 64 - размер числа в битах.
           num, err := strconv.ParseInt(line, 10, 64)

           // ParseInt может вернуть ошибку, например, если ввели не число.
           // nil - пустое значение ошибки:
           // если ошибка равна nil, значит, ошибки нет
           if err != nil {
               panic(err) // печатает ошибку и экстренно завершает программу
           }

           if num < minNum {
               minNum = num
```

```
            }
        } else {
            panic("you must input 5 numbers")
        }
    }

    fmt.Println("minimum number is:", minNum)
}
```

Работает, всё хорошо.

```
> go run app.go
2734723
345
12
3888
44
minimum number is: 12
```

Теперь усложним задачу. Нужно считать 5 чисел и написать их в порядке, обратном порядку ввода. Для этого мы бы могли использовать 5 переменных, но что-то подсказывает, что для этого должен быть более простой способ с циклом, так как действия однотипные.

Более того, нам может понадобиться считать не 5, а 1000 чисел. Для этого нам нужно какое-то хранилище однотипных объектов с определённым размером. Для таких задач и были придуманы массивы.

Создание массива

Есть несколько способов создать массив.

1. Объявить массив без инициализации:

```
var arrWithoutInit [5]int  // {0, 0, 0, 0, 0}
```

Так как все типы в Go имеют значение по умолчанию, массив будет заполнен значениями по умолчанию его типа. В данном случае нулями.

2. То же самое можно сделать в форме инициализации с `:=`

```
arrWithEmptyInit := [5]int{}  // {0, 0, 0, 0, 0}
```

3. Можно создать массив из заранее известных значений:

```
arrWithInit := [5]int{10, 20, 30, 40, 50}
```

При этом можно не указывать длину массива, Go вполне способен вычислить её сам.

```
arrInitWithoutLenght := [...]int{10, 20, 30, 40, 50}
```

Важно

Длина массива должна быть известна на этапе компиляции, так как длина — это часть типа массива, то есть `[5]int` и `[10]int` — это два разных типа. Поэтому нельзя, например, сделать так:

```
package main

import "fmt"

func main() {
   arrLen := 10 * 32
   arr := [arrLen]int{}
   fmt.Println(arr)
}
```

Получим ошибку компиляции:

```
> go run app.go
./app.go:7:9: non-constant array bound arrLen
```

Но можно так:

```
package main

import "fmt"

const arrLen = 10 * 32

func main() {
   arr := [arrLen]int{}
   fmt.Println(arr)
}
```

```
> go run app.go
[0 0 ... 0]
```

Доступ к элементам массива

Доступ к элементам массива бывает двух видов:

- получение значения — значение копируется, как при любом присваивании в Go;
- запись или изменение значения

Для этих способов используется одинаковый синтаксис:

```
package main

import "fmt"

func main() {
   arr := [...]int{1, 2, 3}

   el0 := arr[0] // чтение (индексация с 0)

   arr[1] = 5 // запись

   arr[2]++ // изменение

   fmt.Println(
       "el0:", el0,
       "arr:", arr,
   )
}
```

```
> go run app.go
el0: 1 arr: [1 5 4]
```

Интересно

Так как длина массива — часть типа массива, при использовании констант в качестве индексов Go проверяет выход за границы массива на этапе компиляции:

```
package main

import "fmt"

func main() {
   arr := [...]int{1, 2, 3}
   el := arr[100] // ошибка компиляции
   fmt.Println(el)
}
```

```
> go run app.go
./app.go:7:11: invalid array index 100 (out of bounds for 3-element array)
```

Если в качестве индекса использовать не константу, а переменную, возникнет паника во время исполнения:

```
package main

import "fmt"

func main() {
   arr := [...]int{1, 2, 3}
   index := 100
   el := arr[index] // паника
   fmt.Println(el)
}
```

```
> go run app.go
panic: runtime error: index out of range [100] with length 3

goroutine 1 [running]:
main.main()
      /Users/dedefer/Desktop/dev/golang/app.go:8 +0x1d
exit status 2
```

Итерация по массиву

Логично, что должен быть способ перебирать все элементы в коллекции. В Go их даже два:

```
package main

import "fmt"

func main() {
   arr := [...]int{1, 2, 3}
   // i или el можно подавить с помощью _
   // как любые другие переменные
   // i и el -- всего лишь имена, можно использовать произвольные
   for i, el := range arr {
       fmt.Println(i, el)
   }



   // можно не указывать el вообще
   // тогда итерация будет перебирать только индексы
   for i := range arr {
       fmt.Println(i, arr[i])
   }

}
```

```
> go run app.go
0 1
1 2
2 3
0 1
1 2
2 3
```

Способ с классическим `for`:

```
package main

import "fmt"

func main() {
   arr := [...]int{1, 2, 3}
   // длину массива можно узнать с помощью функции len
   for i := 0; i < len(arr); i++ {
       fmt.Println(i, arr[i])
   }
}
```

Результат абсолютно аналогичен предыдущему.

Мы рекомендуем всегда использовать способ с `range`, так как он лучше считывается.

Многомерные массивы

Так как массив может хранить абсолютно любой тип, он также может хранить другой массив:

```
package main

import "fmt"

func main() {
   arr := [3][3]int{}
   fmt.Println(arr)
}
```

```
> go run app.go
[[0 0 0] [0 0 0] [0 0 0]]
```

Доступ в этом случае происходит сначала к первому, а потом ко второму уровню вложенности:

```
package main

import "fmt"

func main() {
   arr := [3][3]int{}
   arr[1][2] = 123
   fmt.Println(arr)
}
```

```
> go run app.go
[[0 0 0] [0 0 123] [0 0 0]]
```

Копирование массивов

Массив — абсолютно такой же тип, как и остальные. По этой причине для его копирования достаточно просто присвоить его значение в другую переменную.

```
package main

import "fmt"

func main() {
   arr0 := [3]int{}
   arr1 := arr0  // копируем массив
   arr0[1] = 123 // изменяем исходный массив
   fmt.Println("arr0:", arr0)
   fmt.Println("arr1:", arr1)
}
```

```
> go run app.go
arr0: [0 123 0]
arr1: [0 0 0]
```

Заключение

С этими знаниями мы можем решить задачу, которая была в начале урока. Напомним, как она звучала: нужно считать 5 чисел и написать их в порядке, обратном порядку ввода.

```
package main

import (
   "bufio"
   "fmt"
   "os"
   "strconv"
)

func main() {
   inputNums := [5]int64{}
   scanner := bufio.NewScanner(os.Stdin)

   // записываем введённые числа в массив в обратном порядке
   for i := len(inputNums) - 1; i >= 0; i-- {
       if scanner.Scan() {
           num, err := strconv.ParseInt(scanner.Text(), 10, 64)
           if err != nil {
               panic(err)
           }
           inputNums[i] = num
       } else {
           panic("you must input 5 numbers")
       }
   }
   for _, el := range inputNums {
       fmt.Println(el)
   }
}
```

```
> go run app.go
5
4
3
2
1
1
2
3
4
5
```

Слайсы (срезы) и функции для работы с ними (len, cap, append)

Массивы редко используются в прикладных программах. Обычно мы не знаем заранее, какой длины массив нам понадобится. Для этих случаев существуют слайсы, их также называют срезами или динамическими массивами. Они очень похожи на массивы, но их размер можно изменять во время работы программы.

Возьмём предыдущую задачу и модифицируем её. Пусть вместо пяти чисел нам нужно считать не определённое заранее количество чисел и вывести их в порядке, обратном порядку ввода. То есть модифицированная задача звучит так: «Нужно считать все числа, которые ввёл пользователь, а после вывести их в обратном порядке».

Создание слайса

Есть несколько способов создать слайс.

1. Создать слайс без инициализации:

```
sliceWithEmptyInit := []int{} // пустой слайс нулевой длины
```

2. Создать слайс из заранее известных значений:

```
sliceWithInit := []int{1, 2, 3, 4, 5}
```

Есть и другие способы, о которых мы поговорим чуть ниже.

Важно! *Значение слайса по умолчанию — `nil`. Некоторые функции не обрабатывают nil-слайсы (например, `json.Unmarshal`), поэтому способ ниже лучше не использовать.*

```
var slice []int // fmt.Println отобразит [], но на самом деле это []int(nil)
```

Итерация и доступ к элементам слайса

Для доступа и итерации по слайсам используется абсолютно тот же синтаксис, что и для массивов.

Нарезка массива на слайсы

Слайс не просто так получил своё название. Слайсы и массивы можно «нарезать» на слайсы.

```
arr := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
// берём «срез» массива с нулевого элемента до последнего
```

```
// обратите внимание, что последний элемент имеет индекс (len(arr) - 1)
// то есть слайс включает начальную границу и ИСКЛЮЧАЕТ конечную
sliceFromArray := arr[0:len(arr)] // []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

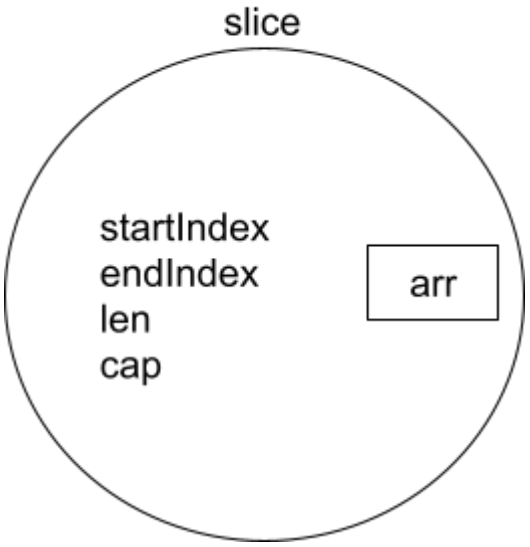
Можно указывать произвольные граничные значения индексов в рамках размера слайса.

```
sliceHalfArray := arr[0:len(arr)/2] // []int{1, 2, 3, 4, 5}
sliceFrom3To6 := arr[3:6] // []int{4, 5, 6}
```

Если начальная граница —индекс 0, то можно «опустить» значение. Аналогично, если конечная граница — len.

```
sliceFirstHalfArrayOmitBound := arr[:len(arr)/2] // arr[0:len(arr)/2]
sliceSecondHalfArrayOmitBound := arr[len(arr)/2:] // arr[len(arr)/2:len(arr)]
sliceOmitBounds := arr[:] // arr[0:len(arr)]
```

Важно! *Слайс не копирует массив, а является представлением массива (array view) и «оборачивает» массив. Рассмотрим на изображении.*



Если изменяется значение в исходном массиве, оно изменяется и в слайсе.

Подробнее мы поговорим об этом на уроке об указателях.

```
package main

import "fmt"

func main() {
	arr := [...]int{1, 2, 3}

	sliceFromArr := arr[:]
	fmt.Println(sliceFromArr)

	arr[0] = 100
	fmt.Println(sliceFromArr)
}
```

```
> go run app.go
[1 2 3]
[100 2 3]
```

Длина и ёмкость слайса

У слайса, как и у массива, есть длина. Узнать её можно с помощью функции `len`.

```
package main

import "fmt"

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

```
> go run app.go
3
```

В отличие от массива, у слайса есть ёмкость. Ёмкость — это длина массива, который «обёрнут» в слайс. Её можно узнать с помощью функции `cap`.

```
package main

import "fmt"

func main() {
	arr := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	sliceFromArr := arr[:3]

	slice := []int{1, 2, 3}

	fmt.Println("arr len:", len(arr))
	println()
	fmt.Println("sliceFromArr len:", len(sliceFromArr))
	fmt.Println("sliceFromArr cap:", cap(sliceFromArr))
	println()
	fmt.Println("slice len:", len(slice))
	fmt.Println("slice cap:", cap(slice))
}
```

```
> go run app.go
arr len: 10
```

```
sliceFromArr len: 3
sliceFromArr cap: 10

slice len: 3
slice cap: 3
```

Становится ясно, что **при создании слайса через инициализацию на самом деле создаётся массив**, а после его оборачивает слайс.

Функции для работы со слайсами

Теперь, когда мы знаем о ёмкости (capacity) слайса, мы можем выяснить, почему слайс называют *динамическим* массивом. Для слайса есть специальная функция `append`, с помощью которой можно добавить значение в конец слайса.

```go
package main

import "fmt"

func main() {
	slice := []int{}               // пустой слайс
	slice = append(slice, 1, 2, 3) // добавляем в конец значения 1, 2, 3
	slice = append(slice, 4)       // добавляем в конец значение 4
	slice = append(slice, 5, 6)    // принимает любое количество аргументов

	fmt.Println(slice)
}
```

```
> go run app.go
[1 2 3 4 5 6]
```

Так причём здесь capacity? Будем печатать длину и capacity после каждого расширения слайса:

```go
package main

import "fmt"

func main() {
	slice := []int{}
	fmt.Println("len:", len(slice), "cap:", cap(slice), "slice:", slice)

	slice = append(slice, 1, 2, 3)
	fmt.Println("len:", len(slice), "cap:", cap(slice), "slice:", slice)

	slice = append(slice, 4)
	fmt.Println("len:", len(slice), "cap:", cap(slice), "slice:", slice)
```

```
	slice = append(slice, 5, 6)
	fmt.Println("len:", len(slice), "cap:", cap(slice), "slice:", slice)
}
```

```
> go run app.go
len: 0 cap: 0 slice: []
len: 3 cap: 4 slice: [1 2 3]
len: 4 cap: 4 slice: [1 2 3 4]
len: 6 cap: 8 slice: [1 2 3 4 5 6]
```

Можем увидеть, что с добавлением значений ёмкость увеличивается не на количество добавляемых значений. Зачем? Всё просто: Go резервирует место под «расширение» слайса.

Дело в том, что, если бы на каждое расширение слайса выделялся новый массив, программы работали бы очень медленно: на каждое расширение приходилось бы выделить новую память под массив и скопировать туда все значения из текущего массива. Выделение памяти и копирование — очень долгие операции, поэтому резервировать «лишнюю» память под потенциальное расширение дешевле с точки зрения ресурсов компьютера.

Подробнее о выделении памяти мы поговорим на уроке об указателях.

Представим ещё одну ситуацию. Мы создали слайс и хотим в цикле его расширять:

```
package main

import "fmt"

func main() {
	count := 5
	slice := []int{}
	for i := 0; i < count; i++ {
		slice = append(slice, i)
		fmt.Println("cap:", cap(slice))
	}
	fmt.Println(slice)
}
```

```
> go run app.go
cap: 1
cap: 2
cap: 4
cap: 4
cap: 8
[0 1 2 3 4]
```

Заметно, что за 5 `append` произошло 4 настоящих расширения с копированием и выделением памяти. Хотелось бы этого избежать. Для этого существует функция `make`.

```
package main

import "fmt"

func main() {
	count := 5
	// 0 - начальная длина
	// count - начальная capacity
	slice := make([]int, 0, count)
	for i := 0; i < count; i++ {
		slice = append(slice, i)
		fmt.Println("cap:", cap(slice))
	}
	fmt.Println(slice)
}
```

```
> go run app.go
cap: 5
cap: 5
cap: 5
cap: 5
cap: 5
[0 1 2 3 4]
```

Если нам заранее известна нижняя или верхняя граница роста слайса, то стоит использовать её в качестве capacity.

Примечание: функцию `make` можно использовать без последнего аргумента для создания слайса заданного размера:

```
slice := make([]int, 5) // []int{0, 0, 0, 0, 0}
```

В таком случае ёмкость слайса также будет равна 5.

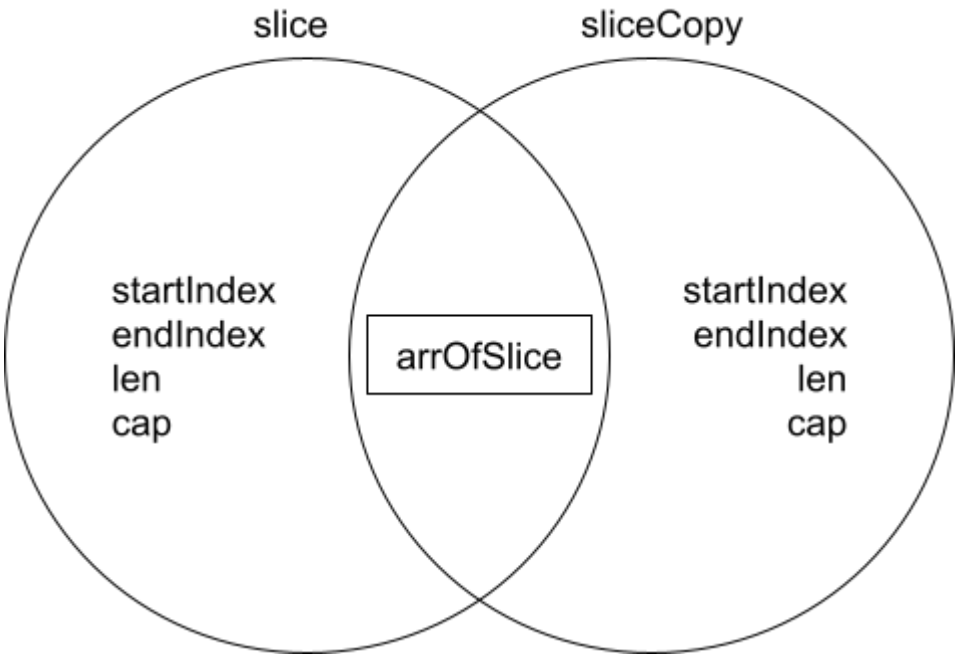
Копирование слайсов

Как мы говорили ранее, слайс — это представление массива. Подробнее о его устройстве мы поговорим через одно занятие. Сейчас важно усвоить, что при присваивании копируется только обёртка, но не обёрнутый массив.

```
package main

import "fmt"

func main() {
	slice := []int{1, 2, 3}
	sliceCopy := slice

	slice[0] = 100

	fmt.Println("slice:", slice)
	fmt.Println("sliceCopy:", sliceCopy)
}
```

```
> go run app.go
slice: [100 2 3]
sliceCopy: [100 2 3]
```

Проиллюстрировать это можно таким образом:



По этой причине есть специальная функция для копирования слайсов `copy`:

```go
package main

import "fmt"

func main() {
	slice := []int{1, 2, 3}
	sliceCopy := make([]int, len(slice))
	copy(sliceCopy, slice) // copy(куда, откуда)

	slice[0] = 100

	fmt.Println("slice:", slice)
	fmt.Println("sliceCopy:", sliceCopy)
}
```

```
> go run app.go
slice: [100 2 3]
sliceCopy: [1 2 3]
```

Если размеры слайсов разные, то `copy` копирует минимум из `len(dst)` и `len(src)` элементов.

Важно: функция `copy` никак не изменяет `len` и `cap` слайсов, так что создание слайса нужного размера — ваша ответственность.

Важно: по причине того, что в Go содержимое слайса не копируется при присваивании, оно также не копируется при передаче слайса в функцию. Таким образом, нужно быть аккуратным, чтобы функция случайно не изменила значения в слайсе.

Заключение

Теперь со всеми этими знаниями решим модифицированную нами задачу. Напомним: нужно считывать числа, пока пользователь их вводит, а позже вывести их в обратной вводу последовательности.

```go
package main

import (
	"bufio"
	"fmt"
	"os"
	"strconv"
)

func main() {
	inputNums := []int64{}
	scanner := bufio.NewScanner(os.Stdin)
```

```
	for scanner.Scan() {
		num, err := strconv.ParseInt(scanner.Text(), 10, 64)
		if err != nil {
			panic(err)
		}
		inputNums = append(inputNums, num)
	}
	for i := len(inputNums) - 1; i >= 0; i-- {
		fmt.Println(inputNums[i])
	}
}
```

```
> go run app.go
1
2
3
3
2
1
```

Мапы (словари, хеши, отображения, карты, ассоциативный массив)

Мап — это структура данных, которая способна хранить значения по ключу. То есть с помощью неё можно составить соответствие **уникального** в мапе **ключа** какому-то **произвольному значению**.

Примечание: мапы реализуют структуру данных «хеш-таблица». Это одна из основных структур данных computer science, и имеет смысл обратиться к «классическим» учебникам по алгоритмам для дополнительного изучения работы мап в Go. Например, можно посмотреть «Искусство программирования» Дональда Кнута или «Алгоритмы: построение и анализ» Томаса Кормена. На собеседованиях очень любят спрашивать об устройстве хеш-таблиц, и мы рекомендуем подробно познакомиться с этой структурой данных и алгоритмами для работы с ней.

Рассмотрим похожую на предыдущие задачу. Пользователь всё так же вводит произвольное количество чисел, а нам нужно посчитать, сколько раз он ввёл каждое из чисел.

То есть при вводе:

```
1
2
1
1
3
```

Нужно вывести что-то вроде:

```
number: 1 count: 3
number: 2 count: 1
number: 3 cound: 1
```

Создание мапы

Тип мапы описывается так:

```
map[keyType]valueType
```

Создать мапу можно следующими способами:

1. Мапа через пустую инициализацию:

```
mapEmptyInit := map[string]int{} // пустая мапа
```

2. Мапа через список инициализации:

```
mapWithInit := map[string]int{
      "a": 1,
      "b": 2,
      "c": 3,
}
```

Для создания мапы можно использовать функцию `make`:

```
mapWithMake := make(map[string]int) // == map[string]int{}
```

Внутреннее устройство мапы похоже на слайс. Она оборачивает другую структуру — хеш-таблицу, к ней язык не предоставляет доступ, — которая тоже имеет capacity и умеет расширяться.

Capacity мапы можно узнать только через рефлексию (рассмотрим позже в курсе), однако `make` позволяет задать предпочтительную capacity:

```
// рекомендуем зарезервировать память под 10 пар ключ-значение
mapWithMakeCap := make(map[string]int, 10)
```

Важно: чаще всего в качестве ключей мапы используют строки. В качестве ключа можно использовать значение произвольного типа, но в большинстве случаев рекомендуется использовать неизменяемые значения в связи с особенностями реализации мапы. Подробнее детали реализации можно узнать в прикреплённой литературе.

Важно: значение по умолчанию у мап, как и у слайсов, `nil`. По той же причине, что и для слайсов, мы не рекомендуем использовать объявление через `var` без инициализации.

Доступ к элементам мапы

Доступ к элементам мапы использует синтаксис, похожий на синтаксис доступа к элементам слайса и массива.

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{
		"a": 1,
		"b": 2,
		"c": 3,
	}
	fmt.Println(exampleMap["a"])
}
```

```
> go run app.go
1
```

Попытка доступа к несуществующему ключу возвращает значение по умолчанию типа значения:

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{
		"a": 1,
		"b": 2,
		"c": 3,
	}
	fmt.Println(exampleMap["abc"])
}
```

```
> go run app.go
0
```

Для проверки существования ключа в мапе можно использовать следующий синтаксис:

```
package main
```

```
import "fmt"

func main() {
	exampleMap := map[string]int{
		"a": 1,
		"b": 2,
		"c": 3,
	}

	// в первую переменную записывается значение
	// во вторую — true, если значение существует
	//              false, если не существует

	a, aExists := exampleMap["a"]
	abc, abcExists := exampleMap["abc"]

	fmt.Println("a:", a, "exists:", aExists)
	fmt.Println("abc:", abc, "exists:", abcExists)
}
```

```
> go run app.go
a: 1 exists: true
abc: 0 exists: false
```

Запись в мапу осуществляется через такой синтаксис:

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{}

	exampleMap["a"] = 1

	fmt.Println("exampleMap:", exampleMap)
}
```

```
> go run app.go
exampleMap: map[a:1]
```

Модификация через операторы `+=`, `++`, `-=`, `--` и подобные также доступна. Причём при использовании этих операторов на несуществующий ключ происходит его «автооживление»:

```
package main
```

```
import "fmt"

func main() {
	exampleMap := map[string]int{}
	fmt.Println("exampleMap:", exampleMap)

	exampleMap["a"]++
	fmt.Println("exampleMap:", exampleMap)

	exampleMap["a"] -= 10
	fmt.Println("exampleMap:", exampleMap)
}
```

```
> go run app.go
exampleMap: map[]
exampleMap: map[a:1]
exampleMap: map[a:-9]
```

Удалить значение из мапы можно с помощью функции `delete`:

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{
		"a": 1,
		"b": 2,
		"c": 3,
	}

	fmt.Println("exampleMap:", exampleMap)

	delete(exampleMap, "a")

	fmt.Println("exampleMap:", exampleMap)

}
```

```
> go run app.go
exampleMap: map[a:1 b:2 c:3]
exampleMap: map[b:2 c:3]
```

Примечание: мапа гарантирует доступ к элементу за константное время, то есть здесь мы говорим об оценке времени выполнения O(1). Значит, скорость поиска элемента в мапе по ключу не зависит от общего количества ключей в мапе.

Примечание: количество ключей (а значит, пар ключ-значение) в мапе можно узнать с помощью функции `len`.

Итерация по мапе

Для перебора значений в мапе используется тот же синтаксис с `range`, что и с массивами и слайсами:

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{}

	exampleMap["a"] = 1
	exampleMap["b"] = 2
	exampleMap["c"] = 3

	exampleMap["abc"] = 123

	for k, v := range exampleMap {
		fmt.Println("key:", k, "value:", v)
	}
}
```

```
> go run app.go
key: abc value: 123
key: a value: 1
key: b value: 2
key: c value: 3
```

Важно

Порядок ключей при переборе мапы через `range` **не детерминирован**, то есть **не зависит от порядка добавления ключей** в мапу. При разработке программ никогда не полагайтесь на то, в каком порядке ключи будут представлены при использовании `range`.

Копирование мап

Так как мапа устроена похоже на слайс, она не копируется при обычном присваивании. Однако функций вроде `copy` для мап нет. Приходится копировать мапы вручную:

```
package main

import "fmt"

func main() {
	exampleMap := map[string]int{
		"a": 1,
		"b": 2,
		"c": 3,
	}

	// создаем мапу, в которую будем копировать
	// указываем предпочитаемую capacity равную len(exampleMap)

	exampleMapCopy := make(map[string]int, len(exampleMap))

	// проходимся по exampleMap и копируем значения
	for k, v := range exampleMap {
		exampleMapCopy[k] = v
	}

	exampleMap["d"] = 4 // изменяем первую мапу
	delete(exampleMap, "a")

	fmt.Println("exampleMap:", exampleMap)
	fmt.Println("exampleMapCopy:", exampleMapCopy)
}
```

```
> go run app.go
exampleMap: map[b:2 c:3 d:4]
exampleMapCopy: map[a:1 b:2 c:3]
```

Заключение

Теперь решим задачу из начала раздела о мапах. Не листайте, мы напомним: пользователь всё так же вводит произвольное количество чисел, а нам нужно посчитать, сколько раз он ввёл каждое из чисел.

```
package main

import (
   "bufio"
   "fmt"
```

```
    "os"
    "strconv"
)

func main() {
    counter := map[int64]uint64{}
    scanner := bufio.NewScanner(os.Stdin)

    for scanner.Scan() {
        num, err := strconv.ParseInt(scanner.Text(), 10, 64)
        if err != nil {
            panic(err)
        }
        counter[num]++
    }

    for number, count := range counter {
        fmt.Println("number:", number, "count:", count)
    }
}
```

```
> go run app.go
1
2
3
5
3
2
1
number: 1 count: 2
number: 2 count: 2
number: 3 count: 2
number: 5 count: 1
```

Выводы

На этом уроке мы посмотрели, как работать в Go с тремя самыми распространёнными типами данных в computer science, которые лежат в основе всех современных приложений. Теперь закрепим.

Массивы

1. Позволяют хранить определённое количество значений одного типа.
2. Длина должна быть известна в процессе разработки.
3. Копируются при присваивании и передаче в функцию.
4. Значение по умолчанию — массив, заполненный значениями по умолчанию типа значения.

Слайсы

1. Позволяют хранить заранее неизвестное количество значений одного типа.
2. Длина может изменяться в процессе работы приложения.
3. Не копируются при присваивании и передаче в функцию, так как являются представлениями массивов.
4. Значение по умолчанию — nil.

Мапы

1. Позволяют хранить заранее неизвестное количество пар ключ-значение одного типа.
2. Ключ обязан быть уникальным в пределах мапы.
3. Ключ обычно представлен неизменяемым типом, чаще всего string.
4. Скорость доступа по ключу не зависит от размера мапы (O(1)).
5. Порядок ключей при переборе произвольный.
6. Не копируются при присваивании и передаче в функцию, так как являются представлениями хеш-таблиц.
7. Значение по умолчанию — nil.

Практическое задание

1. Познакомьтесь с алгоритмом сортировки вставками. Напишите приложение, которое принимает на вход набор целых чисел и отдаёт на выходе его же в отсортированном виде. Сохраните код, он понадобится нам в дальнейших уроках.
2. Проверьте себя:
    a. вам должны быть знакомы следующие ключевые слова Go: `map`, `range`;
    b. вам должны быть знакомы функции: `make`, `len`, `cap`, `append`, `copy`, `delete`

Дополнительные материалы

1. Arrays, slices (and strings): The mechanics of 'append' ([Перевод] Массивы, срезы (и строки): Механизм 'вставки').
2. Полезные манипуляции над слайсами: Slice Tricks.
3. Статья о внутреннем устройстве слайсов: Go Slices: usage and internals - The Go Blog.
4. Статья про мапы в блоге Go: https://blog.golang.org/maps
5. Effective Go, главы Data, Initialization.
6. Книга «Алгоритмы: построение и анализ» (здесь можно узнать про хеш-таблицы).
7. Книга «Язык программирования Go».
8. Пакет sort — sort.Slice.