GeekBrains

Go. Уровень 1

Операторы и управляющие конструкции. Базовые функции

На этом уроке

1. Познакомимся с управляющими конструкциями.
2. Попрактикуемся в написании Go-приложений, использующих управляющие конструкции.
3. Продолжим знакомство с «алгоритм».

Оглавление

Управляющие конструкции

Ещё одна концепция, которая понадобится нам в программировании, — управляющие конструкции (control structures). В этой части мы познакомимся с тремя из них: `if`, `for` и `switch`. Ещё одну конструкцию (`select`) оставим на второй модуль и уроки по конкурентности.

Если вы ещё не сталкивались с управляющими конструкциями в языках программирования, пора познакомиться с ветвлениями и циклами. Для описания алгоритмов с ветвлениями и циклами часто используют блок-схемы, с которыми вы, возможно, уже знакомы из школьного курса информатики.

Итак, конструкция `if` используется для реализации ветвлений. Она может использоваться самостоятельно или в комбинациях с `else` и `else if`:

```
// Простейший if
package main

import (
	"fmt"
)

func main() {
	a := 100
	if a%10 == 0 {
		fmt.Println("Число a кратно десяти")
```

```
	}
}
```

```
// Комбинация if - else
package main

import (
	"fmt"
)

func main() {
	a := 100
	if a%10 == 0 {
		fmt.Println("Число a кратно десяти")
	} else {
		fmt.Println("Число a не кратно десяти")
	}
}
```

```
// Комбинация if - else if
package main

import (
	"fmt"
)

func main() {
	a := 15
	if a%10 == 0 {
		fmt.Println("Число a кратно десяти")
	} else if a%5 == 0 {
		fmt.Println("Число a не кратно десяти, но кратно пяти")
	}
}
```

```
//  Комбинация if - else if - else
package main

import (
	"fmt"
)

func main() {
	a := 15
	if a%10 == 0 {
		fmt.Println("Число a кратно десяти")
	} else if a%5 == 0 {
		fmt.Println("Число a не кратно десяти, но кратно пяти")
	} else {
```

```
            fmt.Println("Число a не кратно ни десяти, ни пяти")
      }
}
```

Для реализации циклов в Go есть только одно ключевое слово — `for`. Возможно, из других языков программирования вы уже знакомы с циклами с предусловием (`while`) и постусловием (`do … while`). В Go мы, конечно, тоже можем написать циклы такого типа, но, как и для цикла со счётчиком, для циклов с пред- и постусловием мы будем использовать ключевое слово `for`.

Цикл со счётчиком — это цикл, который выполняется заданное количество раз. Для его реализации мы заводим переменную-счётчик, которая инкрементируется на каждой итерации цикла до тех пор, пока её значение не достигнет заданного условия. Например, напишем цикл, который пробегает по всем байтам строки и выводит значение каждого байта:

```
package main

import (
      "fmt"
)

func main() {
      s := "Hello, world!"

      for i := 0; i < len(s); i++ {
            fmt.Println(s[i])
      }
}
```

В этом примере `i` — счётчик цикла. Всё, что находится внутри фигурных скобок `for { }`, называется телом цикла.

Другой популярный вид циклов — **цикл с предусловием**, он выполняется до тех пор, пока истинно условие, заданное до начала цикла. Это условие проверяется перед первой итерацией, а затем каждый раз после выполнения очередной итерации цикла. Например:

```
package main

import (
      "fmt"
      "math/rand"
)

func main() {
      condition := true
      for condition {
```

```
            nr := rand.Int()
            fmt.Println(nr)
            if nr%4 == 0 {
                  condition = false
            }
      }
}
```

В этом примере мы генерируем случайные числа до тех пор, пока не будет сгенерировано число, кратное четырём.

Модифицируем этот же цикл так, чтобы он стал циклом с постусловием:

```
package main

import (
      "fmt"
      "math/rand"
)

func main() {
      for {
            nr := rand.Int()
            fmt.Println(nr)
            if nr%4 == 0 {
                  break
            }
      }
}
```

Этот пример выглядит даже прагматичнее. Мы избавились от части логики и просто выходим из цикла с помощью оператора прерывания `break`.

Другой полезный оператор при работе с циклами — `continue`. Он сразу же перенаправляет нас на следующую итерацию цикла. Рассмотрим пример:

```
package main

import (
      "fmt"
      "math/rand"
)

func main() {
      for {
            nr := rand.Int()
```

```
            if nr%2 == 0 {
                  fmt.Printf("%d кратно двум\n", nr)
                  continue
            }

            if nr%3 == 0 {
                  fmt.Printf("%d кратно трем\n", nr)
                  break
            }
      }
}
```

В этом примере мы выходим из цикла, когда нам встречается число, которое кратно трём, но при этом не кратно двум.

Наконец, рассмотрим конструкцию `switch`. Она применяется в качестве альтернативного варианта для конструкций `if - if else`. В отличие от многих других языков программирования, в Go в случае `switch` выполнится только выбранный `case`, поэтому нам не нужно использовать `break` для остановки. Наоборот, чтобы пройти через все последующие case, нужно использовать оператор `fallthrough`. Также в Go то, что мы определяем в `case`, не обязательно должно быть константой и числом. Можно использовать самые разные типы данных и даже выражения.

Перепишем пример, который мы использовали для `if - else`, на switch – case:

```
package main

import (
      "fmt"
)

func main() {
      a := 15
      switch a % 10 {
      case 0:
            fmt.Println("Число a кратно десяти")
      case 5:
            fmt.Println("Число a не кратно десяти, но кратно пяти")
      default:
            fmt.Println("Число a не кратно ни десяти, ни пяти")

      }
}
```

Простейший калькулятор

Чтобы было интереснее, напишем не просто абстрактное приложение, а самый настоящий калькулятор. Как и в прошлом уроке, начнём с простого варианта: пусть пользователь вводит два числа, а мы их будем складывать:

```
package main

import "fmt"

func main() {
  var a, b float32

  fmt.Print("Введите первое число: ")
  fmt.Scanln(&a)

  fmt.Print("Введите второе число: ")
  fmt.Scanln(&b)

  fmt.Printf("Сумма чисел: %f\n", a+b)
}
```

Теперь усложним приложение: предложим пользователю выбрать арифметическую операцию.

```
package main

import (
  "fmt"
  "os"
)

func main() {
  var a, b, res float32
  var op string

  fmt.Print("Введите первое число: ")
  fmt.Scanln(&a)

  fmt.Print("Введите второе число: ")
  fmt.Scanln(&b)

  fmt.Print("Введите арифметическую операцию (+, -, *, /): ")
  fmt.Scanln(&op)

  switch op {
  case "+":
     res = a + b
```

```
  case "-":
     res = a - b
  case "*":
     res = a * b
  case "/":
     res = a / b
  default:
     fmt.Println("Операция выбрана неверно")
     os.Exit(1)
  }

  fmt.Printf("Результат выполнения операции: %f\n", res)
}
```

Здесь мы добавили минимальную **валидацию** входных данных, то есть соответствие входных данных ограничениям задачи. В случае, если вместо указанной нами арифметической операции пользователь ввёл что-то неожиданное, мы выводим сообщение об ошибке и выходим из приложения. Подумайте, какие ещё данные в этой задаче нужно валидировать.

Структуры данных и алгоритмы

Продолжим знакомство с алгоритмами, начатое в прошлом уроке.

Например, рассмотрим задачу сортировки. Пусть у нас есть последовательность из нескольких чисел. Нам нужно переставить числа в последовательности так, чтобы все числа оказались отсортированы по возрастанию. Решение такой задачи в общем виде для любой последовательности и будет алгоритмом. Один из самых простых (но не самых эффективных) алгоритмов для решения этой задачи — сортировка пузырьком.

Вот так её можно реализовать на Go:

```
package main

import (
	"fmt"
)

func main() {
	fmt.Println(bubbleSort([]int{5, 3, 6, 8, 1, 2}))
}

func bubbleSort(arr []int) []int {
	swapped := true
	for swapped {
		swapped = false
		for i := 0; i < len(arr)-1; i++ {
```

```
                    if arr[i+1] < arr[i] {
                        arr[i+1], arr[i] = arr[i], arr[i+1]
                        swapped = true
                    }
                }
            }
            return arr
}
```

Обратите внимание на функцию `bubbleSort`, реализующую алгоритм сортировки:

1. На вход подаётся слайс `arr`.
2. На выходе ожидается отсортированный слайс.
3. В теле функции мы в цикле проходим по всем элементам слайса и меняем элементы местами, если они не отсортированы. Такое прохождение повторяем до тех пор, пока все элементы не будут отсортированы.

Все свойства алгоритма в реализации функции `bubbleSort` выполняются: есть входные и выходные данные, реализация функции преобразует входные данные в выходные так, чтобы решить задачу сортировки.

Эффективность и оценка алгоритмов

Алгоритмы, решающие одну и ту же задачу, могут различаться по эффективности. Как правило, при расчёте эффективности учитывают количество операций, которые нужно произвести для прохождения всего алгоритма.

В примере с сортировкой, который мы рассмотрели выше, количество операций всегда будет зависеть от количества элементов во входном слайсе. Например, рассмотрим слайс из N элементов. В лучшем случае, если нам передали слайс, который уже был отсортирован, нам всё равно придётся пройти его один раз и произвести N операций сравнения (внутренний цикл). В худшем случае и внешний цикл нам придётся выполнить N раз. Получается, мы N раз выполняем внешний цикл и на каждой его итерации N раз выполняем внутренний. Количество итераций будет N×N. Именно число N×N — оценка сложности алгоритма сортировки пузырьком.

Для всех задач, которые встретятся вам на этом уроке, попробуйте посчитать количество операций, требуемое для выполнения этих задач.

На следующих уроках мы продолжим разговор о сложности и оценке алгоритмов. Для более глубокого погружения в эту тему уже сейчас рекомендуем книгу «Алгоритмы: построение и анализ».

Практическое задание

1. Доработать калькулятор: больше операций и валидации данных.
2. *Задание для продвинутых (необязательное).* Написать приложение, которое ищет все простые числа от 0 до N включительно. Число N должно быть задано из стандартного потока ввода.
3. Проверьте себя. Вам должны быть знакомы следующие ключевые слова Go: `if`, `else`, `switch`, `case`, `default`, `for`, `break`, `continue`.

Дополнительные материалы

Для погружения в структуру приложения и управляющие конструкции

1. Go tour: пакеты, переменные и функции.
2. Go tour: управляющие конструкции.
3. Effective Go, главы Semicolons, Control structures, Initialization.