Публикация на тему

Go

Go lang - как язык для обслуживания REST API. Go можно использовать вместе с PHP и Laravel

Автор
Михалькевич Александр Викторович

Публикация

Наименование Go

Автор А.В.Михалькевич

Специальность Go lang - как язык для обслуживания REST API. Go можно использовать вместе с PHP и Laravel,

Анотация

Anotation in English

Ключевые слова

Количество символов 76736

Содержание

Введение

Go — это язык программирования с открытым исходным кодом, первоначально разработанный как внутренний проект Google и ставший обще доступным еще в 2009 году.

1 Назначение языка Go

Go является языком программирования общего назначения, но в основном используется для написания системных инструментов, утилит командной строки, веб-сервисов и программного обеспечения, которое работает в сетях. С помощью Go также можно обучаться программированию, плюс он является хорошим кандидатом на первый язык программирования благодаря своей немногословности, четким идеям и принципам. Go может помочь в разработке следующих видов приложений:

профессиональные веб-сервисы;
сетевые инструменты и серверы, такие как Kubernetes и Istio;
серверные системы;
системные утилиты;
утилиты командной строки, такие как docker и hugo;
приложения, которые обмениваются данными в формате JSON;
приложения, обрабатывающие данные из реляционных баз данных SQL, баз данных NoSQL или систем баз данных;
компиляторы и интерпретаторы для разрабатываемых языков программирования;

Существует множество сценариев, когда вы можете выбрать Go, например:

создание сложных утилит командной строки со множеством команд, подкоманд и параметров командной строки;
создание приложений с высокой степенью параллелизма;
разработка серверов, работающих с API, и клиентов, которые взаимодействуют путем обмена данными во множестве форматов, включая JSON, XML и CSV;
разработка серверов и клиентов WebSocket;
разработка серверов и клиентов gRCP;
разработка надежных системных инструментов для UNIX и Windows;

изучение программирования.

2 Установка

Информация по установке - https://go.dev/

При установке происходит распаковка библиотек и файлов в указанную папку. Соответственно, для удаления Go - достаточно просто удалить папку.

Установка Go в Ubuntu

```
wget -c https://go.dev/dl/go1.22.0.linux-amd64.tar.gz
```

После загрузки архива необходимо открыть терминал `Ctrl+Alt+T`, перейти в папку с архивом:

```
cd ~/Downloads
```

и выполнить команду:

```
sudo tar -C /usr/local/ -xzf go1.22.0.linux-amd64.tar.gz
```

Чтобы система знала, где найти команду Go, можно добавить ее в PATH в качестве переменной окружающей среды.
Для этого откройте домашнюю папку, затем нажмите «Редактировать файл .profile» (или .bashrc). Когда файл откроется, добавьте следующие строки и сохраните его.

```
# set PATH so it includes /usr/local/go/bin if it exists
if [ -d "/usr/local/go/bin" ] ; then
    PATH="/usr/local/go/bin:$PATH"
fi
```

Теперь в терминале можно проверить:

```
go -v
```

3 Hello world и запуск программ

Создайте файл hello.go со следующим содержимым:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Любой исходный код на Go начинается с пакета объявлений. В нашем случае название пакета — main, что имеет особое значение в Go. Ключевое слово import Позволяет включить

функционал из существующего пакета. В нашем случае, если нам понадобится только часть функций пакета fmt, который входит в комплект поставки библиотеки Go. Пакеты, которые не являются ее частью, импортируются с использованием их полного интернет-пути. Следующий важный момент при создании исполняемого приложения — это функция main(). Перейдите к ее точке входа в приложение и приступит к выполнению приложения с кодом, обнаруженного в функции main() пакета main.

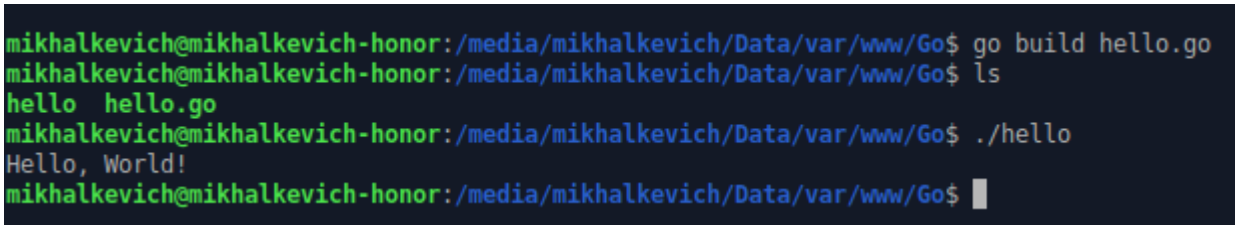
Теперь мы можем скомпилировать данный код в программу, а можем просто запустить.

Запуск программы:

```
go run hello.go
```

Компиляция и запуск:

```
go build hello.go
./hello
```



Go run vs Go build

Так какой же он, Go - компилируемый или скриптовой язык? На самом деле, Go - одновременно и компилируемый и скриптовой. Когда программа написана, мы можем скомпилировать результат с помощью `go build`, а в процессе разработки, часто удобно бывает просто запустить программу без компиляции, для этого имеется `go run`

4 Переменные локальные и глобальные

Любой исходный код на Go начинается с пакета объявлений. В нашем случае название пакета — main, что имеет особое значение в Go. Ключевое слово import Позволяет включить функционал из существующего пакета. В нашем В случае, если нам понадобится только часть функций пакета fmt, который входит в комплект поставки библиотеки Go. Пакеты, которые не являются ее частью, импортируются с использованием их полного интернет-пути. Следующий важный момент при создании исполняемого приложения — это функция main(). Перейдите к ее точке входа в приложение и приступит к выполнению приложения с кодом, обнаруженного в функции main() пакета main.

Глобальные переменные объявляются с помощью ключевого слова `var`. Существует также нотация `:=`, которую можно использовать вместо объявления var. Команда `:=` определяет новую переменную, делая вывод о данных из следующего за ней значения. Официальное название для `:=` звучит так: короткое при- сваивание
Пример

```
package main
import (
        "fmt"
        "math"
```

```
)

var Global int = 1234
var AnotherGlobal = -5678
func main() {
        var j int
        i := Global + AnotherGlobal
        fmt.Println("Initial j value:", j)
        j = Global
        // math.Abs() требует параметр float64
        // соответственно, мы приводим тип
        k := math.Abs(float64(AnotherGlobal))
        fmt.Printf("Global=%d, i=%d, j=%d k=%.2f.\n", Global, i, j, k)
}
```

5 Управляющие конструкции

К управляющим конструкциям относятся линейные алгоритмы, алгоритмы ветвления и циклы.

5 .1 Управление ходом выполнения программы

Go поддерживает структуры управления `if/else` и `switch`. Обе эти структуры можно найти в большинстве современных языков программирования, так что если вы уже программировали на другом языке, то должны быть знакомы с if и switch. Оператор if не использует круглые скобки для встраивания проверяемых условий, потому что в Go вообще не используются круглые скобки. Кроме того, if ожидаемо поддерживает операторы else и else if.

5 .2 Оператор if/else

Продемонстрируем использование if с помощью очень распространенного паттерна, который повсеместно применяется в Go. Он гласит, что если значение переменной error, возвращаемой из функции, равно nil, то с выполнением функции все в порядке. В противном случае где-то возникла ошибка, требующая особого внимания. Этот паттерн обычно реализуется следующим образом:

```
err := anyFunctionCall()
if err != nil {
 // сделать что-нибудь, если возникла ошибка
}
```

err — это переменная, которая содержит значение error, возвращаемое функцией, а != говорит о том, что значение переменной err не равно nil. Подобный код вы встретите в Go-программах множество раз.

5 .3 Оператор switch

Оператор `switch` имеет две разные формы. В первой он содержит вычисляемое выражение, тогда как во второй не имеет выражения для вычисления. В этом случае выражения вычисляются в каждом операторе case, что повышает гибкость switch. Основное преимущество switch заключается в том, что при правильном использовании он упрощает сложные и трудночитаемые блоки if-else.

```
// с выражением после switch
switch argument {
 case "0":
  fmt.Println("Zero!")
 case "1":
  fmt.Println("One!")
 case "2", "3", "4":
  fmt.Println("2 or 3 or 4")
  fallthrough
 default:
  fmt.Println("Value:", argument)
}
```

Здесь мы видим блок switch с четырьмя ветвлениями. Первые три требуют точного совпадения string, а последнее соответствует всему остальному. Порядок операторов case важен, поскольку выполняется только первое совпадение. Ключевое слово fallthrough сообщает Go, что после выполнения этой ветки необходимо перейти на следующую, которая в данном случае является веткой default

Следующий код показывает вторую форму switch, где условие вычисляется в каждой ветви case:

```
value, err := strconv.Atoi(argument)
        if err != nil {
                fmt.Println("Cannot convert to int:", argument)
                return
        }

        // Без выражения после switch
        switch {
        case value == 0:
                fmt.Println("Zero!")
        case value > 0:
                fmt.Println("Positive integer")
        case value < 0:
                fmt.Println("Negative integer")
        default:
                fmt.Println("This should not happen:", value)
        }
```

5 .4 Итерации с помощью циклов for и range

Go поддерживает циклы `for`, а также ключевое слово range для перебора всех элементов массивов, срезов и карт. Примером простоты Go служит тот факт, что для работы с циклами существует только одно ключевое слово for.

Вы также можете создавать циклы for с переменными и условиями. Цикл for можно завершить с помощью ключевого слова `break` или пропустить текущую итерацию, применив ключевое слово `continue`. При использовании с `range` циклы for позволяют просматривать все элементы среза или массива, не зная размер структуры данных.

```
// Традиционное использование цикла for
        for i := 0; i < 10; i++ {
                fmt.Print(i*i, " ")
        }
        fmt.Println()

// Использование цикла for как do-while
        i := 0
        for ok := true; ok; ok = (i != 10) {
                fmt.Print(i*i, " ")
                i++
        }
        fmt.Println()

        i = 0
        for {
                if i == 10 {
                        break
                }
                fmt.Print(i*i, " ")
                i++
        }
        fmt.Println()

// С помощью ключевого слова range цикл for превращается в foreach для
прохода по массиву или срезу. Это срез, но диапазон также работает с
массивами.
        aSlice := []int{-1, 2, 1, -1, 2, -2}
        for i, v := range aSlice {
                fmt.Println("index:", i, "value: ", v)
        }
```

6 Типы данных

Данные хранятся и используются в переменных, и все переменные Go должны иметь тип данных, который определяется явно или неявно.

Все данные, которые хранятся в памяти, по сути представляют просто набор битов. И

именно тип данных определяет, как будут интерпретироваться эти данные и какие операции с ними можно производить. Язык Go является статически типизированным языком, то есть все используемые в программе данные имеют определенный тип.

Go имеет ряд встроенных типов данных, а также позволяет определять свои типы. Рассмотрим базовые встроенные типы данных, которые мы можем использовать.

6 .1 Основные типы данных, однотипные

К основным типам данных относятся:

тип данных error;
числовые типы данных;
нечисловые типы данных;
Go-константы;
указатели;

6 .1 .1 Числовые типы данных

Go поддерживает целочисленные значения, значения с плавающей запятой и комплексные числа в различных вариациях в зависимости от занимаемого ими объема памяти. Такой подход экономит память и вычислительное время. Целочисленные типы могут быть как со знаком, так и без знака.

Целочисленные типы

Ряд типов представляют целые числа:

int8: представляет целое число от -128 до 127 и занимает в памяти 1 байт (8 бит)

int16: представляет целое число от -32768 до 32767 и занимает в памяти 2 байта (16 бит)

int32: представляет целое число от -2147483648 до 2147483647 и занимает 4 байта (32 бита)

int64: представляет целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт (64 бита)

uint8: представляет целое число от 0 до 255 и занимает 1 байт

uint16: представляет целое число от 0 до 65535 и занимает 2 байта

uint32: представляет целое число от 0 до 4294967295 и занимает 4 байта

uint64: представляет целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт

byte: синоним типа `uint8`, представляет целое число от 0 до 255 и занимает 1 байт

rune: синоним типа `int32`, представляет целое число от -2147483648 до 2147483647 и занимает 4 байта

int: представляет целое число со знаком, которое в зависимости о платформы может занимать либо 4 байта, либо 8 байт. То есть соответствовать либо int32, либо int64.

uint: представляет целое беззнаковое число только без знака, которое, аналогично типу int, в зависимости о платформы может занимать либо 4 байта, либо 8 байт. То есть соответствовать либо uint32, либо uint64.

Здесь несложно запомнить, что есть типы со знаком (то есть которые могут быть отрицательными) и есть безнаковые положительные типы, которые начинаются на префикс u (uint32). Ну и также есть byte - синоним для uint8 и rune - синоним для int32.

Стоит отметить типы int и uint. Они имеют наиболее эффективный размер для определенной платформы (32 или 64 бита). Это наиболее используемый тип для представления целых чисел в программе. Причем различные компиляторы могут предоставлять различный размер для этих типов даже для одной и той же платформы.

Примеры определения переменных, которые представляют целочисленные типы:

```
var a int8 = -1
var b uint8 = 2
var c byte = 3  // byte - синоним типа uint8
var d int16 = -4
var f uint16 = 5
var g int32 = -6
var h rune = -7     // rune - синоним типа int32
var j uint32 = 8
var k int64 = -9
var l uint64 = 10
var m int = 102
var n uint = 105
```

Числа с плавающей точкой

Для представления дробных чисел есть два типа:

float32: представляет число с плавающей точкой от 1.4*10-45 до 3.4*1038(для положительных). Занимает в памяти 4 байта (32 бита)

float64: представляет число с плавающей точкой от 4.9*10-324 до 1.8*10308 (для положительных) и занимает 8 байт.

Тип float32 обеспечивает шесть десятичных цифр точности, в то время как точность, обеспечиваемая типом float64, составляет около 15 цифр

Примеры использования типов float32 и float64:

```
var f float32 = 18
var g float32 = 4.5
var d float64 = 0.23
var pi float64 = 3.14
var e float64 = 2.7
```

В качестве разделителя между целой и дробной частью применяется точка.

Комплексные числа

Существуют отдельные типы для представления комплексных чисел:

complex64: комплексное число, где вещественная и мнимая части представляют числа float32

complex128: комплексное число, где вещественная и мнимая части представляют числа float64

Пример использования:

```
var f complex64 = 1+2i
var g complex128 = 4+3i
```

6 .1 .2 Тип bool

Логический тип или тип bool может иметь одно из двух значений: true (истина) или false (ложь).

```
var isAlive bool = true
var isEnabled bool = false
```

6 .1 .3 Строки

Строки представлены типом string. В Go строке соответствует строковый литерал - последовательность символов, заключенная в двойные кавычки:

```
var name string = "Том Сойер"
```

Кроме обычных символов строка может содержать специальные последовательности (управляющие последовательности), которые начинаются с обратного слеша \. Наиболее распространенные последовательности:

\n: переход на новую строку
\r: возврат каретки
\t: табуляция
\": двойная кавычка внутри строк
\\: обратный слеш

Стандартный Go-пакет strings позволяет манипулировать строками UTF-8 в Go и содержит множество эффективных функций.

```
package main

import (
        "fmt"
        s "strings"
```

```
        "unicode"
)

var f = fmt.Printf

func main() {
        upper := s.ToUpper("Hello there!")
        f("To Upper: %s\n", upper)
        f("To Lower: %s\n", s.ToLower("Hello THERE"))

        f("%s\n", s.Title("tHis wiLL be A title!"))

        f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAlis"))
        f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAli"))

        f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
        f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
        f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
        f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))

        f("Index: %v\n", s.Index("Mihalis", "ha"))
        f("Index: %v\n", s.Index("Mihalis", "Ha"))
        f("Count: %v\n", s.Count("Mihalis", "i"))
        f("Count: %v\n", s.Count("Mihalis", "I"))
        f("Repeat: %s\n", s.Repeat("ab", 5))

        f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
        f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n\t "))
        f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n", "\n\t "))

        f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
        f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
        f("Compare: %v\n", s.Compare("MIHALIS", "MIHalis"))

        t := s.Fields("This is a string!")
        f("Fields: %v\n", len(t))
        t = s.Fields("ThisIs a\tstring!")
        f("Fields: %v\n", len(t))

        f("%s\n", s.Split("abcd efg", ""))
        f("%s\n", s.Replace("abcd efg", "", "_", -1))
        f("%s\n", s.Replace("abcd efg", "", "_", 4))
        f("%s\n", s.Replace("abcd efg", "", "_", 2))

        lines := []string{"Line 1", "Line 2", "Line 3"}
        f("Join: %s\n", s.Join(lines, "+++"))

        f("SplitAfter: %s\n", s.SplitAfter("123++432++", "++"))

        trimFunction := func(c rune) bool {
                return !unicode.IsLetter(c)
```

```
        }
        f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .", trimFunction))
}
```

6 .1 .4 Значение по умолчанию

Если переменной не присвоено значение, то она имеет значение по умолчанию, которое определено для ее типа. Для числовых типов это число 0, для логического типа - false, для строк - ""(пустая строка).

6 .1 .5 Неявная типизация

При определении переменной мы можем опускать тип в том случае, если мы явно инициализируем переменную каким-нибудь значением:

```
var name = "Tom"
```

В этом случае компилятор на основании значения неявно выводит тип переменной. Если присваивается строка, то то соответственно переменная будет представлять тип string, если присваивается целое число, то переменная представляет тип int и т.д.

То же самое по сути происходит при кратком определении переменной, когда также явным образом не указывается тип данных:

```
name := "Tom"
```

При этом стоит учитывать, что если мы не указываем у переменной тип, то ей обязательно надо присвоить некоторое начальное значение. Объявление переменной одновременно без указания типа данных и начального значения будет ошибкой:

```
var name    // ! Ошибка
```

Надо либо указать тип данных (в этом случае переменная будет иметь значение по умолчанию):

```
var name string
```

Либо указать начальное значение, на основании которого выводится тип данных:

```
var name = "Tom"
```

либо

```
var name string = "Tom"
```

Неявная типизация нескольких переменных:

```
var (
        name = "Tom"
        age = 27
)
```

Или так:

```
var name, age = "Tom", 27
```

6 .1 .6 Руны

В настоящее время поддержка символов Unicode является распространенным требованием — Go разработан с учетом поддержки Unicode, что является основной причиной наличия типа данных rune. Руна — 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.

Вы можете определить руну, используя одинарные кавычки: r := '€', а вывести целое значение составляющих ее байтов с помощью fmt.Println(r) — в этом случае целочисленное значение равно 8364. Для вывода в виде одного символа Unicode потребуется использование управляющей строки %c в fmt.Printf().

```
package main

import "fmt"

func main() {
        aString := "Hello World! €"
        fmt.Println("First character", string(aString[0]))

        // Runes
        // A rune
        r := '€'
        fmt.Println("As an int32 value:", r)
        // Convert Runes to text
        fmt.Printf("As a string: %s and as a character: %c\n", r, r)

        // Print an existing string as runes
        for _, v := range aString {
                fmt.Printf("%x ", v)
        }
        fmt.Println()

        // String to rune Array
        // myRune := []rune(aString)
        // fmt.Printf("myRune %U\n", myRune)

        // Rune array to string
        // runeArray := []rune{'1', '2', '3'}
        // s := string(runeArray)
        // fmt.Println(s)

        // Print an existing string as characters
```

```
        for _, v := range aString {
                fmt.Printf("%c", v)
        }
        fmt.Println()
}
```

6 .1 .7 Дата и время

Главное при работе с временем и датами в Go — тип данных time.Time, который представляет момент времени с точностью до наносекунды. Каждое значение time.Time связано с местоположением (часовым поясом).

Функция `time.Now().Unix()` возвращает популярное время эпохи UNIX, которое представляет собой количество секунд, прошедших с 00:00:00 UTC 1 января 1970 года. Если вы хотите преобразовать время UNIX в эквивалентное значение time.Time, то можете использовать функцию time.Unix().

Для синтаксического анализа используется функция `time.Parse()`, и ее полная сигнатура выглядит так: Parse(layout, value string) (Time, error), где layout — это формат, а value — входные данные для анализа. Возвращаемое значение time.Time представляет собой момент времени с точностью до наносекунды и содержит информацию как о дате, так и о времени.

В приведенном ниже коде показано, как работать с временем эпохи в Go, и представлен сам процесс анализа.

```
package main

import (
        "fmt"
        "os"
        "time"
)

func main() {
        start := time.Now()

        if len(os.Args) != 2 {
                fmt.Println("Usage: dates parse_string")
                return
        }
        dateString := os.Args[1]

        // Сравнение только даты
        d, err := time.Parse("02 January 2006", dateString)
        if err == nil {
                fmt.Println("Full:", d)
                fmt.Println("Time:", d.Day(), d.Month(), d.Year())
        }

        // Сравнение даты и времени
```

```
        d, err = time.Parse("02 January 2006 15:04", dateString)
        if err == nil {
                fmt.Println("Full:", d)
                fmt.Println("Date:", d.Day(), d.Month(), d.Year())
                fmt.Println("Time:", d.Hour(), d.Minute())
        }

        // Сравнение даты с числовым значением месяца и времени
        d, err = time.Parse("02-01-2006 15:04", dateString)
        if err == nil {
                fmt.Println("Full:", d)
                fmt.Println("Date:", d.Day(), d.Month(), d.Year())
                fmt.Println("Time:", d.Hour(), d.Minute())
        }

        // Сравнение только времени
        d, err = time.Parse("15:04", dateString)
        if err == nil {
                fmt.Println("Full:", d)
                fmt.Println("Time:", d.Hour(), d.Minute())
        }

        t := time.Now().Unix()
        fmt.Println("Epoch time:", t)
        // Convert Epoch time to time.Time
        d = time.Unix(t, 0)
        fmt.Println("Date:", d.Day(), d.Month(), d.Year())
        fmt.Printf("Time: %d:%d\n", d.Hour(), d.Minute())

        duration := time.Since(start)
        fmt.Println("Execution time:", duration)
}
```

Данная утилита принимает дату и время и преобразует их в разные часовые пояса. Пример:

```
loc, _ = time.LoadLocation("America/New_York")
fmt.Printf("New York Time: %s\n", now.In(loc))
```

6 .1 .8 Константы

Константы, как и переменные, хранят некоторые данные, но в отличие от переменных значения констант нельзя изменить, они устанавливаются один раз. Вычисление констант производится во время компиляции. Благодаря этому уменьшается количество работы, которую необходимо произвести во время выполнения, упрощается поиск ошибок, связанных с константами (так как некоторые из них можно обнаружить на момент компиляции).

Для определения констант применяется ключевое слово const:

```
const pi float64 = 3.1415
```

И в отличие от переменной мы не можем изменить значение константы. А если и попробуем это сделать, то при компиляции мы получим ошибку:

```
const pi float64 = 3.1415
pi = 2.7182             // ! Ошибка
```

В одном определении можно объявить разу несколько констант:

```
const (
    pi float64 = 3.1415
    e float64 = 2.7182
)
```

или

```
const pi, e = 3.1415, 2.7182
```

Если у константы не указан тип, то он выводится неявно на основании того значения, которым инициализируется константа:

```
const n = 5     //  тип int
```

В то же время необходимо обязательно инициализировать константу начальным значением при ее объявлении.

Если определяется последовательность констант, то инициализацию значением можно опустить для всех констант, кроме первой. В этом случае константа без значения полчит значение предыдущей константы:

```
const (
    a = 1
    b
    c
    d = 3
    f
)
fmt.Println(a, b, c, d, f)      // 1, 1, 1, 3, 3
```

Константы можно инициализировать только константными значениями, например, литералами типа чисел или строк, или значениями других констант. Но инициализировать константу значением переменной мы не можем.

Генератор констант `iota` предназначен для объявления последовательности связанных значений, которые используют увеличивающиеся числа, не прибегая к необходимости явно вводить каждое из них.

```
package main

import (
        "fmt"
)
```

```
func main() {

        const (
                Zero int = iota
                One
                Two
                Three
                Four
        )

        fmt.Println(One)
        fmt.Println(Two)
        fmt.Println(Three)
        fmt.Println(Four)
}
```

Сохраните данный исходник в файл constants.go и запустите:

```
go run constants.go
```

получим:

```
-3rd-main/ch02$ go run constants.go
1
2
3
4
```

Если какие-то значения нужно пропустить, то можем использовать символ нижнего подчёркивания:

```
        const (
                Zero int = iota
                _
                One
                _
                Two
                _
                Three
                _
                Four
        )
```

При запуске программы получим:

```
-3rd-main/ch02$ go run constants.go
2
4
6
8
```

6 .1 .9 Массивы и срезы

Когда надо сохранить несколько значений одного и того же типа данных в одной переменной и получать к ним доступ, используя индекс, то самый простой способ сделать это

в Go — использовать массивы или срезы.

Массивы являются наиболее широко используемыми структурами данных и поддерживаются практически во всех языках программирования благодаря своей простоте и скорости доступа. Go предоставляет альтернативу массивам, которая называется **срезом**.

При определении переменной массива вы должны задать ее размер. В противном случае необходимо поместить в объявление массива `[...]` и позволить компилятору Go определить длину для вас. Таким образом, вы можете создать массив с четырьмя элементами string либо как [4]string{"Zero", "One", "Two", "Three"}, либо как [...]string{"Zero", "One", "Two", "Three"}. Если ничего не заключить в квадратные скобки, то вместо массива будет создан срез.

Вы не можете изменить размер массива после того, как он уже создан. Когда вы передаете массив функции, Go создает его копию и передает ее в функцию — поэтому любые изменения, которые вы вносите в массив внутри функции, теряются при возврате.

6 .1 .10 Срезы

Срезы в Go эффективнее массивов главным образом потому, что динамичны, а это значит, что при необходимости они могут увеличиваться или уменьшаться после создания. Кроме того, любые изменения, которые вы вносите в срез внутри функции, влияют и на исходный срез.

Вы можете создать срез с помощью `make()` или как массив, не указывая его размера или используя `[...]`. Если инициализировать срез не требуется, то лучше и быстрее использовать make(). Однако если вы хотите инициализировать его во время создания, то make() уже не подойдет. Как итог, вы можете создать срез с тремя элементами `float64` с помощью

```
aSlice := []float64{1.2, 3.2, -4.5}
```

Создать срез, вмещающий те же три элемента, с помощью make() можно так:

```
var users []string = make([]float64, 3)
aSlice[0] = 1.2
aSlice[1] = 3.2
aSlice[2] = -4.5
```

Как срезы, так и массивы могут иметь много измерений — создать срез с двумя измерениями с помощью make() так же просто:

```
make([][]int, 2)
```

Оператор возвращает срез двумя измерениями, где первое равно 2 (строки), а второе (столбцы) не определено и должно быть явно указано при добавлении в него данных.

Если необходимо определить и инициализировать срез с двумя измерениями одновременно, то можем сделать следующее:

```
twoD := [][]int{{1, 2, 3},{4, 5, 6}}
```

Длину массива или среза можно получить с помощью `len()`. Вы можете добавлять новые элементы к полному срезу с помощью функции `append()`.

```
package main

import "fmt"

func main() {
        // Create an empty slice
        aSlice := []float64{}
        // Both length and capacity are 0 because aSlice is empty
        fmt.Println(aSlice, len(aSlice), cap(aSlice))

        // Add elements to a slice
        aSlice = append(aSlice, 1234.56)
        aSlice = append(aSlice, -34.0)
        fmt.Println(aSlice, "with length", len(aSlice))

        // A slice with length 4
        t := make([]int, 4)
        t[0] = -1
        t[1] = -2
        t[2] = -3
        t[3] = -4
        // Now you will need to use append
        t = append(t, -5)
        fmt.Println(t)

        // A 2D slice
        // You can have as many dimensions as needed
        twoD := [][]int{{1, 2, 3}, {4, 5, 6}}

        // Visiting all elements of a 2D slice
        // with a double for loop
        for _, i := range twoD {
                for _, k := range i {
                        fmt.Print(k, " ")
                }
                fmt.Println()
        }

        make2D := make([][]int, 2)
        fmt.Println(make2D)
        make2D[0] = []int{1, 2, 3, 4}
        make2D[1] = []int{-1, -2, -3, -4}
        fmt.Println(make2D)
}
```

Ёмкость среза, cap()

У срезов также есть и дополнительное свойство — емкость (capacity),которое можно получить с помощью функции cap().

Вот небольшая Go-программа, в которой показаны свойства длины и емкости срезов.

```
package main

import "fmt"

func main() {
        // Only length is defined. Capacity = length
        a := make([]int, 4)
        fmt.Println("L:", len(a), "C:", cap(a))
        // Initialize slice. Capacity = length
        b := []int{0, 1, 2, 3, 4}
        fmt.Println("L:", len(b), "C:", cap(b))
        // Same length and capacity
        aSlice := make([]int, 4, 4)
        fmt.Println(aSlice)
        // Add an element
        aSlice = append(aSlice, 5)
        fmt.Println(aSlice)
        // The capacity is doubled
        fmt.Println("L:", len(aSlice), "C:", cap(aSlice))
        // Now add four elements
        aSlice = append(aSlice, []int{-1, -2, -3, -4}...)
        fmt.Println(aSlice)
        // The capacity is doubled
        fmt.Println("L:", len(aSlice), "C:", cap(aSlice))
}
```

Емкость показывает, насколько можно расширить срез, не выделяя большего объема памяти и не изменяя базового массива. Хотя после создания среза его емкость обрабатывается Go, разработчику позволяется задать емкость среза во время создания с помощью функции make() — после этого емкость среза удваивается каждый раз, когда длина среза становится больше его текущей емкости. Первый аргумент make() — это тип среза и его размеры, второй — его начальная длина, а третий необязательный) — емкость среза. Тип данных среза уже не может изменяться после создания, в отличие от двух других свойств.

Запись наподобие `make([]int, 3, 2)` генерирует сообщение об ошибке, поскольку в любой момент времени емкость среза (2) не может быть меньше его длины (3).

Установка правильной емкости среза, если она известна заранее, ускорит ваши программы, так как Go не придется выделять новый базовый массив и копировать все данные.

Выбор части среза

Go позволяет выбирать части среза при условии, что все нужные элементы расположены рядом друг с другом.

С помощью нотации `[1:]` можно пропустить первый элемент.

```
package main

import "fmt"

func main() {
        aSlice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
        fmt.Println(aSlice)
        l := len(aSlice)

        // Первые 5 элементов
        fmt.Println(aSlice[0:5])
        // Первые 5 элементов
        fmt.Println(aSlice[:5])

        // Последние 2 элемента
        fmt.Println(aSlice[l-2 : l])

        // Последние 2 элемента
        fmt.Println(aSlice[l-2:])

        // Первые 5 элементов
        t := aSlice[0:5:10]
        fmt.Println(len(t), cap(t))

        // Элементы по индексам 2,3,4
        t = aSlice[2:5:10]
        fmt.Println(len(t), cap(t))

        // Элементы по индексам 0,1,2,3,4
        t = aSlice[:5:6]
        fmt.Println(len(t), cap(t))
}
```

Функция copy()

Go содержит функцию copy(), предназначенную для копирования существующего массива в срез или существующего среза в другой срез.

В следующей программе показано использование функции copy(). Введите ее в текстовом редакторе и сохраните как copySlice.go:

```
package main

import "fmt"

func main() {
        a1 := []int{1}
        a2 := []int{-1, -2}
        a5 := []int{10, 11, 12, 13, 14}
        fmt.Println("a1", a1)
        fmt.Println("a2", a2)
        fmt.Println("a5", a5)

        // copy(destination, input)
        // len(a2) > len(a1)
        copy(a1, a2)
        fmt.Println("a1", a1)
        fmt.Println("a2", a2)
```

```
        // len(a5) > len(a1)
        copy(a1, a5)
        fmt.Println("a1", a1)
        fmt.Println("a5", a5)

        // len(a2) < len(a5) -> OK
        copy(a5, a2)
        fmt.Println("a2", a2)
        fmt.Println("a5", a5)
}
```

Сортировка

Пакет sort может сортировать срезы встроенных типов данных, при этом вам не придется писать дополнительный код. Кроме того, Go предоставляет функцию `sort.Reverse()` для сортировки в порядке, обратном порядку по умолчанию. Однако интересен тот факт, что sort позволяет писать собственные функции сортировки для пользовательских типов данных, реализуя интерфейс `sort.Interface`.

```
package main

import (
        "fmt"
        "sort"
)

func main() {
        sInts := []int{1, 0, 2, -3, 4, -20}
        sFloats := []float64{1.0, 0.2, 0.22, -3, 4.1, -0.1}
        sStrings := []string{"aa", "a", "A", "Aa", "aab", "AAa"}

        fmt.Println("sInts original:", sInts)
        sort.Ints(sInts)
        fmt.Println("sInts:", sInts)
        sort.Sort(sort.Reverse(sort.IntSlice(sInts)))
        fmt.Println("Reverse:", sInts)

        fmt.Println("sFloats original:", sFloats)
        sort.Float64s(sFloats)
        fmt.Println("sFloats:", sFloats)
        sort.Sort(sort.Reverse(sort.Float64Slice(sFloats)))
        fmt.Println("Reverse:", sFloats)

        fmt.Println("sStrings original:", sStrings)
        sort.Strings(sStrings)
        fmt.Println("sStrings:", sStrings)
        sort.Sort(sort.Reverse(sort.StringSlice(sStrings)))
        fmt.Println("Reverse:", sStrings)
}
```

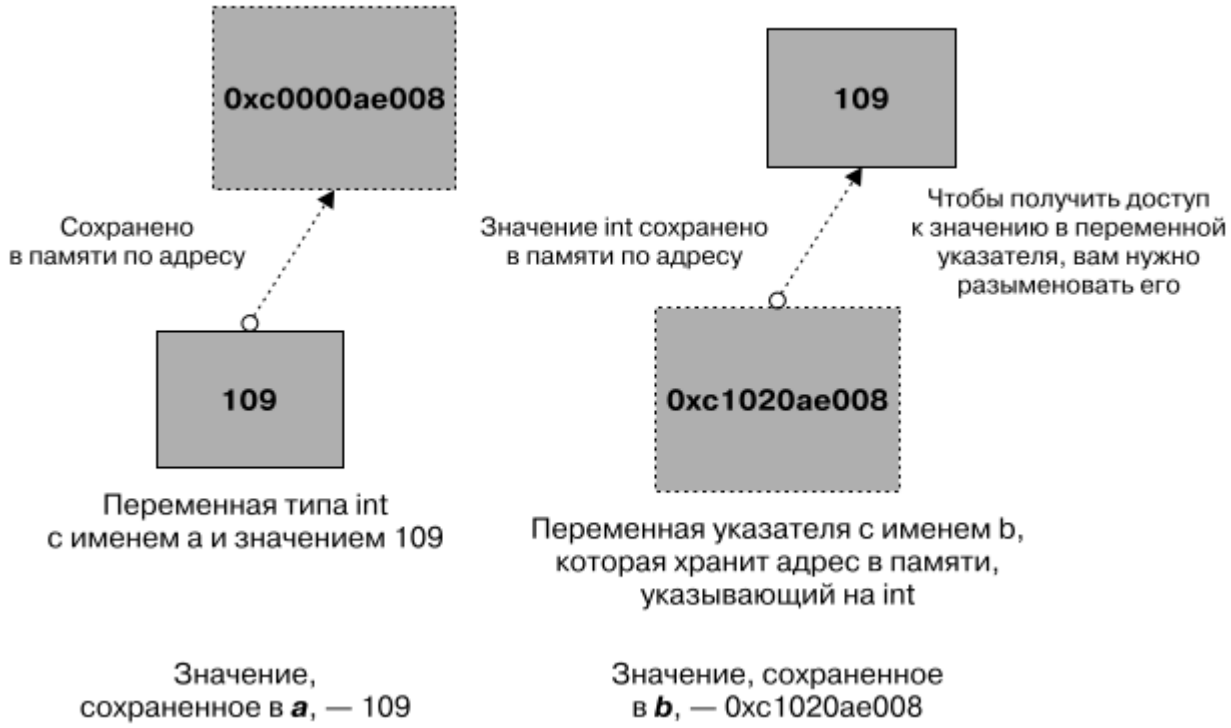
6 .1 .11 Указатели

Указатели представляют собой объекты, значением которых служат адреса других объектов (например, переменных).

Указатель определяется как обычная переменная, только перед типом данных ставится символ звездочки *. Например, определение указателя на объект типа int:

```
var p *int
```

Схематично разницу между указателем на int и переменной int можно представить так:



Если переменная-указатель указывает на существующую обычную переменную, то любые изменения, которые вы вносите в сохраненное значение с помощью переменной-указателя, изменят обычную переменную.

Указателю можно присвоить адрес переменной типа int. Для получения адреса применяется операция &, после которой указывается имя переменной (&x).

```
package main

import "fmt"

func main() {
    var x int = 4       // определяем переменную
    var p *int          // определяем указатель
    p = &x              // указатель получает адрес переменной
    fmt.Println(p)      // значение самого указателя - адрес переменной x
}
```

Здесь указатель p хранит адрес переменной x. Что важно, переменная x имеет тип int, и указатель p указывает именно на объект типа int. То есть должно быть соответствие по типу. И

если мы попробуем вывести адрес переменной на консоль, то увидим, что он представляет шестнадцатеричное значение:

```
0xc0420120a0
```

В каждом отдельном случае адрес может отличаться, но к примеру, в моем случае машинный адрес переменной x - 0xc0420120a0. То есть в памяти компьютера есть адрес 0xc0420120a0, по которому располагается переменная x.

По адресу, который хранит указатель, мы получить значение переменной x. Для этого применяется операция * или операция разыменования. Результатом этой операции является значение переменной, на которую указывает указатель. Применим данную операцию и получим значение переменной x:

```
package main

import "fmt"

func main() {
    var x int = 4
    var p *int  = &x                // указатель получает адрес переменной
    fmt.Println("Address:", p)      // значение указателя - адрес переменной
x
    fmt.Println("Value:", *p)       // значение переменной x
}
```

Консольный вывод

```
Address: 0xc0420c058
Value: 4
```

И также используя указатель, мы можем менять значение по адресу, который хранится в указателе:

```
var x int = 4
var p *int = &x
*p = 25
fmt.Println(x)      // 25
```

Для определения указателей можно использовать также сокращенную форму:

```
f := 2.3
pf := &f
fmt.Println("Address:", pf)
fmt.Println("Value:", *pf)
```

Пустой указатель

Если указателю не присвоен адрес какого-либо объекта, то такой указатель по умолчанию имеет значение nil (по сути отстутствие значения). Если мы попробуем получить значение по такому пустому указателю, то мы столкнемся с ошибкой:

```
var pf *float64
fmt.Println("Value:", *pf)  // ! ошибка, указатель не указывает на какой-либо объект
```

Поэтому при работе с указателями иногда бывает целесообразано проверять на значение nil:

```
var pf *float64
if pf != nil{
    fmt.Println("Value:", *pf)
}
```

Функция new

Переменная представляет именованный объект в памяти. Язык Go также позволяет создавать безымянные объекты - они также размещаются в памяти, но не имеют имени как переменные. Для этого применяется функция `new(type)`. В эту функцию передается тип, объект которого надо создать. Функция возвращает указатель на созданный объект:

```
package main

import "fmt"

func main() {
    p := new(int)
    fmt.Println("Value:", *p)       // Value: 0 - значение по умолчанию
    *p = 8                          // изменяем значение
    fmt.Println("Value:", *p)       // Value: 8
}
```

В данном случае указатель p будет иметь тип *int, поскольку он указывает на объект типа int. Создаваемый объект имеет значение по умолчанию (для типа int это число 0).

Объект, созданный с помощью функции new, ничем не отличается от обычной переменной. Единственное что, чтобы обратиться к этому объекту - получить или изменить его адрес, необходимо использовать указатель.

Указатели как параметры функции

По умолчанию все параметры передаются в функцию по значению. Например:

```
package main
import "fmt"

func changeValue(x int){
    x = x * x
}
func main() {
    d := 5
    fmt.Println("d before:", d)     // 5
    changeValue(d)                  // изменяем значение
    fmt.Println("d after:", d)      // 5 - значение не изменилось
}
```

Функция changeValue изменяет значение параметра, возводя его в квадрат. Но после вызова этой функции мы видим, что значение переменной d, которая передается в changeValue, не изменилось. Ведь функция получает копию данной переменной и работает с ней независимо от оригинальной переменной d. Поэтому d никак не изменяется.

Однако что, если нам все таки надо менять значение передаваемой переменной? И в этом случае мы можем использовать указатели:

```
package main

import "fmt"

func changeValue(x *int){
    *x = (*x) * (*x)
}
func main() {
    d := 5
    fmt.Println("d before:", d)     // 5
    changeValue(&d)                 // изменяем значение
    fmt.Println("d after:", d)      // 25 - значение изменилось!
}
```

Теперь функция changeValue принимает в качестве параметра указатель на объект типа int. При вызове функции changeValue в нее передается адрес переменной d (changeValue(&d)). И после ее выполнения мы видим, что значение переменной d изменилось.

Указатель как результат функции

Функция может возвращать указатель:

```
package main
import "fmt"

func createPointer(x int) *int{
    p := new(int)
    *p = x
    return p
}

func main() {
    p1 := createPointer(7)
    fmt.Println("p1:", *p1)     // p1: 7
    p2 := createPointer(10)
    fmt.Println("p2:", *p2)     // p2: 10
    p3 := createPointer(28)
    fmt.Println("p3:", *p3)     // p3: 28
}
```

В данном случае функция createPointer возвращает указатель на объект int.

6 .2 Составные типы данных

В Go имеется поддержка **карт** и **структур** — составных типов данных. Если массив или срез не в силах справиться с задачей, то вам, скорее всего, придется использовать карты. Если и карта не может вам помочь, то следует рассмотреть необходимость создания и использования структуры.

6 .2 .1 Карты

Как массивы, так и срезы позволяют использовать в качестве индексов только целые положительные числа.

Карты (или хеш-таблицы) — эффективные структуры данных, поскольку позволяют использовать индексы различных типов данных в качестве ключей. Практическое эмпирическое правило заключается в том, что вы должны использовать карту, когда вам нужны индексы, не являющиеся целыми положительными числами, или когда целочисленные индексы разделены большими интервалами.

Возможность создавать индексы любого типа позволяет искать элементы и получать к ним доступ на основе заданного ключа или, в более сложных ситуациях, комбинации ключей.

Если необходимо создать карту, используя литерал карты, то можно сделать так:

```
m := map[string]int {
"key1": -1
"key2": 123
}
```

Порядок элементов в карте рэндомизирован. Это значит, что найти элемент в карте мы можем либо по ключу либо по значению, но не по номеру элемента.

Получить длину карты, которая представляет собой количество ключей в карте, можно используя функцию `len()`, работающую также с массивами и срезами. Вы можете удалить пару «ключ — значение» из карты, используя функцию `delete()`, которая принимает два аргумента: имя карты и название ключа, именно в таком порядке.

Объединяясь с ключевым словом `range`, цикл `for` реализует функциональность циклов foreach из других языков программирования и позволяет выполнять итерации по всем элементам карты, не зная ее размера или ключей.

```
package main

import "fmt"

func main() {
        // range works with maps as well
        aMap := make(map[string]string)
        aMap["123"] = "456"
        aMap["key"] = "A value"
        for key, v := range aMap {
```

```
                fmt.Println("key:", key, "value:", v)
        }

        for _, v := range aMap {
                fmt.Print(" # ", v)
        }
        fmt.Println()
}
```

6 .2 .2 Структуры

Структуры в Go одновременно и эффективны, и очень популярны. Они используются для организации и группировки различных типов данных под одним именем. Структуры — более универсальный тип данных в Go и даже могут быть связаны с функциями, которые называются методами.

Определяя новую структуру, вы группируете набор значений в единый тип данных, который позволяет вам передавать и получать этот набор как единый объект. Структура имеет поля, и у каждого поля есть собственный тип данных, который даже может быть другой структурой или срезом структур. Кроме того, поскольку структура является новым типом данных, она определяется с помощью ключевого слова type, за которым следует название структуры и ключевое слово struct. Следующий код определяет новую структуру Entry:

```
type Entry struct {
  Name string
  Surname string
  Year int
}
```

Кроме того, вы можете создавать новые экземпляры структуры, используя ключевое слово new(), например:

```
pS := new(Entry)
```

Таким образом создаётся объект, который обладает следующими свойствами. Рассмотрим работу со структурами на примере:

```
package main

import "fmt"

// Данный код объявляет структуру Entry
type Entry struct {
        Name    string
        Surname string
        Year    int
}
```

```
// в этой функции происходит вывод данных структуры
func zeroS() Entry {
        return Entry{}
}
```

Если переменной не задано начальное значение, то компилятор Go автоматически инициализирует ее нулевым значением ее типа данных. Для структур это означает, что структурная переменная без начального значения инициализируется нулевыми значениями каждого из типов данных ее полей. Следовательно, функция zeroS() возвращает инициализированную нулями структуру Entry.

Добавим функцию добавления элементов в структуру Entry:

```
// функция добавления данных в структуру Entry:
func initS(N, S string, Y int) Entry {
        if Y < 2000 {
                return Entry{Name: N, Surname: S, Year: 2000}
        }
        return Entry{Name: N, Surname: S, Year: Y}
}
```

Благодаря такому понятию как "срезы структур" мы можем получить доступ к группе элементов структур по заданным параметрам. Пример:

```
package main

import (
        "fmt"
        "strconv"
)

type record struct {
        Field1 int
        Field2 string
}

func main() {
        S := []record{}
        for i := 0; i < 10; i++ {
                text := "text" + strconv.Itoa(i)
                temp := record{Field1: i, Field2: text}
                S = append(S, temp)
        }
        // Accessing the fields of the first element
        fmt.Println("Index 0:", S[0].Field1, S[0].Field2)
        fmt.Println("Number of structures:", len(S))
        sum := 0
        for _, k := range S {
                sum += k.Field1
        }
        fmt.Println("Sum:", sum)
```

```
}
```

6 .3 Пакет errors

Для представления условий ошибки и сообщений Go содержит специальный тип данных error. На практике это означает, что этот язык обрабатывает ошибки как значения. Чтобы успешно программировать на Go, вы должны иметь представление об ошибках, которые могут возникнуть по мере использования функций и методов, а также соответствующим образом их обрабатывать.

Go следует такому соглашению о значениях error: если значение переменной error равно nil, то ошибки не было. Если значение error отлично от nil, это означает, что преобразование было неудачным и значение в string не является допустимым значением int.

Если требуется вернуть пользовательскую ошибку, то можно использовать errors.New() из пакета errors.

```
package main

import (
        "errors"
        "fmt"
)

// Custom error message with errors.New()
func check(a, b int) error {
        if a == 0 && b == 0 {
                return errors.New("this is a custom error message")
        }
        return nil
}


func main() {
        err := check(0, 10)
        if err == nil {
                fmt.Println("check() ended normally!")
        } else {
                fmt.Println(err)
        }

}
```

7 Рефлексия

Рефлексия позволяет работать с типами данных, которые не существуют на момент написания кода, но могут существовать в будущем, когда мы используем существующий пакет с пользовательскими типами данных.

Кроме того, рефлексия может пригодиться, когда необходимо работать с типами данных, которые не реализуют общий интерфейс и, следовательно, имеют необычное или неизвестное поведение. Это не означает плохое или ошибочное поведение, а просто необычное, такое как определяемая пользователем структура.

Наиболее полезными частями пакета `reflect` являются два типа данных: reflect.Value и reflect.Type. В частности, reflect.Value используется для хранения значений любого типа, тогда как reflect.Type служит для представления Go-типов. Существуют две функции: reflect.TypeOf() и reflect.valueOf(), которые возвращают reflect.Type и reflect.Value соответственно. Обратите внимание, что reflect.TypeOf() возвращает фактический тип переменной, и если мы исследуем структуру, то она вернет имя структуры. Поскольку структуры играют ключевую роль в Go, пакет reflect содержит метод reflect.NumField(), предназначенный для перечисления количества полей в структуре, а также метод Field(), позволяющий получать значение reflect.Value определенного поля структуры.

Пакет reflect также определяет тип данных reflect.Kind, который используется для представления определенного типа данных переменной: int, struct и т. д. В документации к пакету reflect перечислены все возможные значения типа данных reflect.Kind. Функция Kind() возвращает вид переменной. Наконец, методы Int() и String() возвращают целое и строковое значения reflect.Value соответственно.

Следующая утилита показывает, как использовать рефлексию для обнаружения внутренней структуры и полей переменной Go-структуры. Введите ее и сохраните как reflection.go:

```
package main

import (
        "fmt"
        "reflect"
)

type Secret struct {
        Username string
        Password string
}

type Record struct {
        Field1 string
        Field2 float64
        Field3 Secret
}

func main() {
        A := Record{"String value", -12.123, Secret{"Mihalis", "Tsoukalos"}}

        r := reflect.ValueOf(A) // Здесь возвращается значение reflect.Value
переменной A.
        fmt.Println("String value:", r.String())
```

```
        iType := r.Type() // Используя Type(), мы получаем тип данных
переменной – в данном случае переменной A.
        fmt.Printf("i Type: %s\n", iType)
        fmt.Printf("The %d fields of %s are\n", r.NumField(), iType)

    // Цикл for выше позволяет посетить все поля структуры и изучить их
характеристики.
        for i := 0; i < r.NumField(); i++ {
        // Оператор fmt.Printf() возвращает имя, тип данных и значение полей.
                fmt.Printf("\t%s ", iType.Field(i).Name)
                fmt.Printf("\twith type: %s ", r.Field(i).Type())
                fmt.Printf("\tand value _%v_\n", r.Field(i).Interface())

                // Проверяем, есть ли в значении другие структуры
                k := reflect.TypeOf(r.Field(i).Interface()).Kind()
                // Чтобы проверить тип данных переменной с помощью строки,
нам нужно сначала преобразовать тип данных в string.
                if k.String() == "struct" {
                        fmt.Println(r.Field(i).Type())
                }

                // Тоже  что и выше, но с использованием внутреннего значение
Struct
                if k == reflect.Struct {
                        fmt.Println(r.Field(i).Type())
                }
        }
}
```

Заключение

Список использованных источников

Приложения