

# کتاب فارسی گولنگ به طور خلاصه

آشنایی با ساختار گولنگ در کمترین زمان ممکن



نویسنده



سید امیر ایرانی

Junior Full Stack Developer

Mobile, Desktop Developer

آدرس اینترنتی کتاب برای دانلود کدها

<https://github.com/Gommunity/gosuccinctly>

محفل گفتگو برنامه نویسان گولنگ

<https://goforask.com>

« این کتاب را به پدر و مادرم یعنی بزرگترین و پیچیده ترین واژه هستی که هنوز معنی روشنی و واضحی از این ۲ معجزه زندگی ارائه نشده و هنوز مشخص نیست چرا تمام حیات یک فرزند مانند یک بند ناف به تعداد نفس هایی که آن ها میکشند وابسته است ... تقدیم می کنم. »

## اجرا اولین برنامه در زبان برنامه نویسی گو

۴	
۵	ساختار گولنگ
۵	Values
۶	Variables
۷	Constants
۹	For
۱۱	If/Else
۱۲	Switch
۱۴	Arrays
۱۶	Slices
۲۲	Maps
۲۷	Range
۲۹	Functions
۳۱	Multiple Return Values
۳۲	Variadic Functions
۳۳	Closures
۳۵	Recursion
۳۶	Pointers
۳۸	Structs
۴۰	Methods
۴۲	Embedded types
۴۴	Interfaces
۴۶	Empty interface
۴۷	Type assertion
۴۹	منابع

## اجرا اولین برنامه در زبان برنامه نویسی گو

ابتدا فایلی با فرمت go مانند hello.go بسازید و کد زیر را در آن قرار دهید :

<http://play.golang.org/p/2C7wwJ6nxG>

کد شماره ۱

```
// Our first program will print the classic "hello world"
// message. Here's the full source code.
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

حالا برای اجرای کد دو حالت وجود داره اگه از دستور `go build` استفاده کنید ابتدا کد شما به فایل باینری تبدیل و خروجی آن ساخته می شود که در محیط ترمینال می توانید اجرا کنید و اگر از دستور `go run` استفاده کنید کد شما ابتدا در temporary سیستم ساخته و سپس اجرا می شود .

```
$ go run hello-world.go
hello world
Sometimes we'll want to build our programs into binaries. We can do this using go build.
$ go build hello-world.go
$ ls
hello-world    hello-world.go
We can then execute the built binary directly.
$ ./hello-world
hello world
```

نکات قابل توجه :

- همیشه برای آغاز یک برنامه باید نامی به عنوان نام پکیج در ابتدای فایل مانند `package main` قرار گیرد . شما می توانید نام های مختلفی را برای `namespace` خود انتخاب کنید اما فایل اصلی که دستور اجرا برنامه گو روی آن صادر می شود حتما باید نام پکیج آن `main` باشد .
- دقت داشته باشیم وقتی برنامه آغاز به کار می کند فقط کدهایی که درون تابع `main` قرار می دهیم اجرا می شود پس بهتر است کدهایمان را به صورت فانکشنال خارج آن بنویسیم و داخل تابع `main` صدا بزنیم .
- تابعی با نام `init` در گو وجود دارد که در صورت تعریف آن با اجرای برنامه ابتدا محتویات داخل آن اجرا شده و بعد تکمیل به سراغ تابع `mian` می رود همانند کد زیر :

<https://play.golang.org/p/vyoCoYrzTpo>

کد شماره ۲

```
package main

import "fmt"

func init() {
    fmt.Println("first")
}

func main() {
```

```
fmt.Println("second")
}
```

## Values در گولنگ

گولنگ همانند زبان های دیگر انواع مختلفی از مقادیر را شامل می شود برای مثال : رشته ها ، اعداد صحیح ، اعداد شناور (اعشاری) ، بولین ها و غیره .

<http://play.golang.org/p/fgGV0yuZdu>

کد شماره ۳

```
// Go has various value types including strings,
// integers, floats, booleans, etc. Here are a few
// basic examples.

package main

import "fmt"

func main() {

    // Strings, which can be added together with `+`.
    fmt.Println("go" + "lang")

    // Integers and floats.
    fmt.Println("1+1 =", 1+1)
    fmt.Println("7.0/3.0 =", 7.0/3.0)

    // Booleans, with boolean operators as you'd expect.
    fmt.Println(true && false)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

خروجی :

```
$ go run values.go
golang
1+1 = 2
7.0/3.0 = 2.3333333333333335
false
true
false
```

- رشته ها را با علامت + می توان کنار هم یا به اصطلاح جمع کرد .

## Variables در گولنگ

همان طور که در گولنگ نمی توان پکیجی را لود کرد و استفاده نکرد در اینجا نیز در صورتی که متغیری میسازید باید از آن در برنامه خودتان استفاده کنید به کد زیر دقت کنید و ببینید متغیرها باید به صراحت ساخته و در صورت نیاز مقدار دهی شوند .

<https://play.golang.org/p/1FnG0dJfxs8>

کد شماره ۴

```
// In Go, _variables_ are explicitly declared and used by
// the compiler to e.g. check type-correctness of function
// calls.

package main

import "fmt"

func main() {

    // `var` declares 1 or more variables.
    var a = "initial"
    fmt.Println(a)

    // You can declare multiple variables at once.
    var b, c int = 1, 2
    fmt.Println(b, c)

    // Go will infer the type of initialized variables.
    var d = true
    fmt.Println(d)

    // Variables declared without a corresponding
    // initialization are _zero-valued_. For example, the
    // zero value for an `int` is `0`.
    var e int
    fmt.Println(e)

    // The `:=` syntax is shorthand for declaring and
    // initializing a variable, e.g. for
    // `var f string = "short"` in this case.
    f := "short"
    fmt.Println(f)
}
```

خروجی :

```
$ go run variables.go
initial
1 2
true
0
short
```

- متغیر **a** با کلمه کلیدی **var** بدون تعیین نوع متغیر ساخته و مقدار دهی شده است . در صورت تعیین نکردن نوع متغیر این کار به صورت خودکار انجام می شود .
- در تعریف متغیر **f** مشاهده می کنیم متغیری ساخته و مقدار دهی شده بدون تعیین نوع متغیر در واقع نوع تعریف کوتاه متغیر به این صورت است که نیازی نیست کلمه کلیدی **var** نوشته شود و فقط کفایست به جای **=** از **:=** استفاده شود . برای تعریف این متغیر به شکل ساده و با تعیین نوع متغیر باید از کد `var f string = "short"` استفاده کرد .
- در صورت عدم مقداری دهی با توجه به نوع متغیر مقداری ثابت در نظر گرفته می شود برای مثال مقدار متغیر **e** برابر **0** خواهد بود ، پس در صورتی که متغیری بدون مقدار تعیین می کنید باید حتما نوع آن را نیز تعیین کنید .
- متغیرهای **b** و **c** نیز به سینتکس تعریف چند متغیر با یک بار استفاده از کلمه کلیدی **var** اشاره دارد و می توانید چندین متغیر در یک خط بسازید .  
الگوی دیگر :

```
var (
    b int = 1
    c int = 2
)
```

## Constants در گولنگ

ثابت ها در گولنگ از مقادیر کارکتر ، رشته ، بولین و اعداد پشتیبانی می کنند .

<https://play.golang.org/p/T5sj0eINnp>

کد شماره ۵

```
// Go supports _constants_ of character, string, boolean,
// and numeric values.

package main

import "fmt"
import "math"

// `const` declares a constant value.
const s string = "constant"

func main() {
    fmt.Println(s)

    // A `const` statement can appear anywhere a `var`
    // statement can.
    const n = 500000000

    // Constant expressions perform arithmetic with
    // arbitrary precision.
    const d = 3e20 / n
    fmt.Println(d)

    // A numeric constant has no type until it's given
    // one, such as by an explicit cast.
    fmt.Println(int64(d))
}
```

```

// A number can be given a type by using it in a
// context that requires one, such as a variable
// assignment or function call. For example, here
// `math.Sin` expects a `float64`.
fmt.Println(math.Sin(n))
}

```

خروجی :

```

$ go run constant.go
constant
6e+11
600000000000
-0.28470407323754404

```

نکات :

- ثابت‌ها هر جایی که می‌توان متغیر ساخت قابل ساخت هستند .
- تفاوت کلی ثابت‌ها با متغیر در این است که مقدار ثابت‌ها قابل تغییر نیست .
- آگه شما هم **EcmaScript** یا **Typescript** کار می‌کنید توجه داشته باشید در گولنگ نمی‌توان داخل یک ثابت تابعی قرار داد و فقط دیتا تایپ‌های مشخصی که ذکر شد قابل استفاده هستند .



تنها متد ایجاد حلقه در گولنگ استفاده از کلمه کلیدی **FOR** است .

[https://play.golang.org/p/KNLLSX4lo\\_](https://play.golang.org/p/KNLLSX4lo_)

کد شماره ۶

```
// `for` is Go's only looping construct. Here are
// three basic types of `for` loops.

package main

import "fmt"

func main() {

    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    // until you `break` out of the loop or `return` from
    // the enclosing function.
    for {
        fmt.Println("loop")
        break
    }

    // You can also `continue` to the next iteration of
    // the loop.
    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

```
$ go run for.go
1
2
3
7
8
9
loop
1
3
5
```

## نکات :

- ساده ترین متد استفاده از حلقه مثال اول است .
- در مثال دوم متغیر `j` داخل حلقه ساخته و مقداردهی شده و شرط و تغییرات روی آن تعریف شده و در اصطلاح به آن `classic` در حلقه `initial/condition/after` در حلقه گویند .
- مثال سوم حلقه ای بی نهایت است که می توان با کلمه کلیدی `break` یا `return` آن را متوقف کرد .
- در مثال آخر برای نمایش هر `iteration` حلقه با استفاده از شرط ، از کلمه کلیدی `continue` استفاده می شود .
- کلمات کلیدی `break` و `continue` در اکثر زبان های برنامه نویسی مشترک است .

تعریف شرط `If/Else` یکی از ساده ترین سینتکس ها در گولنگ می باشد .

<https://play.golang.org/p/g-aqMz0lvf>

کد شماره ۷

```
// Branching with `if` and `else` in Go is
// straight-forward.

package main

import "fmt"

func main() {

    // Here's a basic example.
    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    // You can have an `if` statement without an else.
    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    // A statement can precede conditionals; any variables
    // declared in this statement are available in all
    // branches.
    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}

// Note that you don't need parentheses around conditions
// in Go, but that the braces are required.
```

خروجی :

```
$ go run if-else.go
7 is odd
8 is divisible by 4
9 has 1 digit
```

- مثال اول به شکل ساده یک شرط و خلاف آن را سنجیده ایم .
- در مثال دوم می بینیم که می توان شرطی را بدون `else` به کار برد .
- در مثال سوم نحوه استفاده از چند شرط با استفاده از `else if` نشان داده شده و متغیری که در شرط تعریف می شود در داخل همه فضاهای شرطی قابل دسترسی است .
- در گولنگ نیاز نیست اطراف شرط از پرانتز استفاده کنید اما محتوای داخل آن باید داخل یک فضای `{ }` قرار بگیرد .

## Switch در گولنگ

برای همه ما پیش میاد که نیاز داریم شرطهای زیادی را بسنجیم و باید چندین بار عبارت `else if` را تایپ کنیم اما اکثر زبان های برنامه نویسی از ویژگی با عنوان `Switch` بهره می برند که سنجش شرط های زیاد را ساده تر می کند .

<https://play.golang.org/p/TJ4Az0KuLfl>

کد شماره ۸

```
// _Switch statements_ express conditionals across many
// branches.

package main

import "fmt"
import "time"

func main() {

    // Here's a basic `switch`.
    i := 2
    fmt.Println("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    // You can use commas to separate multiple expressions
    // in the same `case` statement. We use the optional
    // `default` case in this example as well.
    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    // `switch` without an expression is an alternate way
    // to express if/else logic. Here we also show how the
    // `case` expressions can be non-constants.
    t := time.Now()
```

```

switch {
case t.Hour() < 12:
    fmt.Println("It's before noon")
default:
    fmt.Println("It's after noon")
}

// A type `switch` compares types instead of values. You
// can use this to discover the type of an interface
// value. In this example, the variable `t` will have the
// type corresponding to its clause.
whatAmI := func(i interface{}) {
    switch t := i.(type) {
    case bool:
        fmt.Println("I'm a bool")
    case int:
        fmt.Println("I'm an int")
    default:
        fmt.Printf("Don't know type %T\n", t)
    }
}
whatAmI(true)
whatAmI(1)
whatAmI("hey")
}

```

خروجی :

```

$ go run switch.go
Write 2 as two
It's a weekday
It's after noon
I'm a bool
I'm an int
Don't know type string

```

نکات :

- در اولین مثال مشاهده می‌کنیم سینتکس تعریف سویچ را به ساده‌ترین شکل ممکن.
- در مثال دوم مشاهده می‌کنیم که می‌توان به کمک کاما ، چند مقدار برای یک گزینه یا `case` انتخاب کنیم و با استفاده از کلمه کلیدی `default` می‌توانیم در صورتی که مقدار ورودی با هیچ گزینه‌ای مطابقت نداشت آن را نمایش دهیم .
- در مثال سوم مشاهده می‌کنیم که می‌توان به سویچ مقداری نداد تا نقش `If/Else` را ایفا کند و داخل `case` شرطی را قرار داد که وابستگی به ورودی ندارد .
- در مثال آخر کلید واژه `Interface` رو می‌بینیم که در قسمت های بعد کامل توضیح داده می‌شود و در این مثال ما نوع مقادیر را می‌سنجیم و یکی از سینتکس های تعریف یک تابع را مشاهده می‌کنیم که در ادامه آموزش ها به نحوه تعریف توابع خواهیم پرداخت .

در گولنگ `Data Type` دیگری با عنوان `Collection` وجود دارد که همان آرایه محسوب می شود .

<https://play.golang.org/p/l-A8eBnwio>

کد شماره ۹

```
// In Go, an _array_ is a numbered sequence of elements of a
// specific length.

package main

import "fmt"

func main() {

    // Here we create an array `a` that will hold exactly
    // 5 `int`s. The type of elements and length are both
    // part of the array's type. By default an array is
    // zero-valued, which for `int`s means `0`s.
    var a [5]int
    fmt.Println("emp:", a)

    // We can set a value at an index using the
    // `array[index] = value` syntax, and get a value with
    // `array[index]`.
    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    // The builtin `len` returns the length of an array.
    fmt.Println("len:", len(a))

    // Use this syntax to declare and initialize an array
    // in one line.
    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    // Array types are one-dimensional, but you can
    // compose types to build multi-dimensional data
    // structures.
    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

```
$ go run arrays.go
emp: [0 0 0 0 0]
set: [0 0 0 0 100]
get: 100
len: 5
dcl: [1 2 3 4 5]
2d: [[0 1 2] [1 2 3]]
```

## نکات :

- هر آرایه ای دو خصوصیت **Data Type** و **Length** را دارا می باشد . ساده ترین متد تعریف یک آرایه در گولنگ مثال اول است که داخل متغیر **a** آرایه ای شامل ۵ خانه که همان صفت **Length** است و با **Data Type** عدد صحیح **int** تعریف شده است .
- اگر آرایه رو ساختید ولی مقدار دهی نکردید مقادیر ثابتی در خانه ها میشیند برای مثال اگر نوع داده ها **int** باشد در هر خانه 0 میشیند .
- بر خلاف زبان هایی چون جاوا اسکریپت نوع داده های داخل یک آرایه باید یکسان باشد .
- برای مقدار دادن به یک خانه خاص در آرایه می توان از متد **array[index] = value** استفاده کرد .
- در گولنگ تابعی با عنوان **len** وجود دارد که طول آرایه یا همان **Length** را نمایش می دهد .
- برای مقدار دادن به یک آرایه هنگام تعریف آن می توان از پرانتز استفاده کرد برای مثال نحوه تعریف آرایه در متغیر **b** را مشاهده نمایید .
- تا اینجا یاد گرفتیم چطور آرایه یک بعدی بسازیم اما برای تعریف آرایه چند بعدی می توان از مثال آخر استفاده کرد که آرایه ای ۲ بعدی تعریف شده است .
- می توان به جای تعداد خانه ها از ... به صورت **[...]string{"Penn", "Teller"}** استفاده نمود و در این حالت موقع کامپایل تعداد خانه ها شمرده می شود و جایگزین .
- نکته بسیار مهمی که در آرایه ها وجود دارد این است که اگر آرایه رو به عنوان یک پارامتر درون یک تابع ارسال کنیم اون مقدار داخل تابع یک کپی از آرایه محسوب می شود در صورتی که در **Slice** ها چنین نیست و برای رفع این محدودیت می توان از پوینتر ها برای رفرنس دادن موقعیت آرایه بر روی **memory** استفاده کرد .

<https://play.golang.org/p/Lu01QP5A0ri>

کد شماره ۱۰

```
package main

import (
    "fmt"
)

func main() {
    myArray := [...]string{"Apples", "Oranges", "Bananas"}
    fmt.Printf("Initial array values: %v\n", myArray)
    myFunction(myArray)
    fmt.Printf("Final array values: %v\n", myArray)
}

func myFunction(arr [3]string) {
    // Change Oranges to Strawberries
    arr[1] = "Strawberries"
    fmt.Printf("Array values in myFunction(): %v\n", arr)
}

// Output
```

```
// Initial array values: [Apples Oranges Bananas]
// Array values in myFunction(): [Apples Strawberries Bananas]
// Final array values: [Apples Oranges Bananas]
```

- با چاپ یک آرایه مقداری همانند [v1 v2 v3 ...] نمایش داده می شود .
- نکته مهمی که در آرایه ها وجود دارد اینکه شما نمی توانید تعداد خانه های آن را بیشتر از مقدار تعیین شده قرار بدید و چندان قابل انعطاف نیست .

## Slices در گولنگ

Slice ها یک نوع داده کلیدی در گولنگ محسوب می شوند که یک رابط قوی برای مدیریت آرایه ها فراهم می کنند . در واقع اگر می خواهید آرایه ها امکانات بیشتری و قابل انعطاف تر باشند باید از Slice ها استفاده کرد .

[https://play.golang.org/p/Z3\\_U32sn8RF](https://play.golang.org/p/Z3_U32sn8RF)

کد شماره ۱۱

```
// _Slices_ are a key data type in Go, giving a more
// powerful interface to sequences than arrays.

package main

import "fmt"

func main() {

    // Unlike arrays, slices are typed only by the
    // elements they contain (not the number of elements).
    // To create an empty slice with non-zero length, use
    // the builtin `make`. Here we make a slice of
    // `string`s of length `3` (initially zero-valued).
    s := make([]string, 3)
    fmt.Println("emp:", s)

    // We can set and get just like with arrays.
    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    // `len` returns the length of the slice as expected.
    fmt.Println("len:", len(s))

    // In addition to these basic operations, slices
    // support several more that make them richer than
    // arrays. One is the builtin `append`, which
    // returns a slice containing one or more new values.
    // Note that we need to accept a return value from
    // `append` as we may get a new slice value.
    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)
```



```

// Slices can also be `copy`'d. Here we create an
// empty slice `c` of the same length as `s` and copy
// into `c` from `s`.
c := make([]string, len(s))
copy(c, s)
fmt.Println("cpy:", c)

// Slices support a "slice" operator with the syntax
// `slice[low:high]`. For example, this gets a slice
// of the elements `s[2]`, `s[3]`, and `s[4]`.
l := s[2:5]
fmt.Println("s11:", l)

// This slices up to (but excluding) `s[5]`.
l = s[:5]
fmt.Println("s12:", l)

// And this slices up from (and including) `s[2]`.
l = s[2:]
fmt.Println("s13:", l)

// We can declare and initialize a variable for slice
// in a single line as well.
t := []string{"g", "h", "i"}
fmt.Println("dcl:", t)

// Slices can be composed into multi-dimensional data
// structures. The length of the inner slices can
// vary, unlike with multi-dimensional arrays.
twoD := make([][]int, 3)
for i := 0; i < 3; i++ {
    innerLen := i + 1
    twoD[i] = make([]int, innerLen)
    for j := 0; j < innerLen; j++ {
        twoD[i][j] = i + j
    }
}
fmt.Println("2d: ", twoD)
}

```

خروجی :

```

$ go run slices.go
emp: [ ]
set: [a b c]
get: c
len: 3
apd: [a b c d e f]
cpy: [a b c d e f]
s11: [c d e]
s12: [a b c d e]
s13: [c d e f]

```

```
dc1: [g h i]
2d: [[0] [1 2] [2 3 4]]
```

## نکات :

- برخلاف آرایه ها ، برش ها (Slices) با Length همیشه ثابتی تعریف نمی شوند و به مقداری که داخل آن ها ریخته می شود بستگی دارند.
- برای ساخت یک برش خالی و بدون مقدار می توان از تابع make کمک گرفت ، در مثال اول ما یک برش با سایز ۳ و بدون هیچ مقداری ساخته ایم .
- در برش ها نیز همانند آرایه ها می توان مقداری برای هر خانه انتخاب کرد و آن ها را فراخواند .
- تابع len نیز طول برش را همانند آرایه ها بر می گرداند .
- برش ها قابلیت های بیشتری نسبت به آرایه ها دارند برای مثال به کمک تابع append می توان یک یا چند مقدار را به برش اضافه نمود .
- برای کپی کردن یک برش می توان از دستور copy در تعریف برش جدید بر روی متغیر c یاد کرد که ابتدا برشی با سایز برش مد نظر ساخته می شود و سپس مقادیر داخل آن کپی می شوند .
- برش ها از سینتکس slice[low:high] برای دریافت قسمت مورد نظر شما از داخل برش استفاده می کنند . ( به متغیر 1 توجه کنید )
- در متغیر t نحوه تعریف یک برش با مقادیر در یک خط نمایش داده شده است .
- برش ها را می شود به ساختار دیتاهای چندبعدی پیاده سازی کرد اما طول مقادیر داخل برش ها می تواند متفاوت باشد و شبیه آرایه های چند بعدی نیست .
- برش ها برخلاف آرایه ها موقعی که به عنوان یک پارامتر به تابع ارسال می شود به صورت رفرنس عمل کرده و تغییرات روی آن باعث تغییر روی آرایه اصلی می شود .

<https://play.golang.org/p/kHqSobJC2Yv>

کد شماره ۱۲

```
package main

import (
    "fmt"
)

func main() {
    mySlice := []string{"Apples", "Oranges", "Bananas"}
    fmt.Printf("Initial slice values: %v\n", mySlice)
    myFunction(mySlice)
    fmt.Printf("Final slice values: %v\n", mySlice)
}

func myFunction(fruits []string) {
    // Change Oranges to Strawberries
    fruits[1] = "Strawberries"
    fmt.Printf("Slice values in myFunction(): %v\n", fruits)
}

// Output :
// Initial slice values: [Apples Oranges Bananas]
// Slice values in myFunction(): [Apples Strawberries Bananas]
// Final slice values: [Apples Strawberries Bananas]
```

- با این که برش ها کاملتر از آرایه ها هستند اما به صورت مشابه در fmt.Println چاپ می شوند .

نتیجه : در واقع اگر آرایه را با مقدار تعریف کردیم ولی تعداد خانه ها رو مشخص نکردیم یا از ... استفاده نکردیم یعنی ما از Slice استفاده کردیم و نه Array و روش ساخت بدون مقدار به کمک تابع درونی make صورت می گیرد .

خوب آگه دقت کرده باشید موقع استفاده از `make` ما باید تعداد خونه ها یا ظرفیت `Capacity` را مشخص کنیم برای این کار اما این تابع درون ساخته شده متغیر سومی هم دارد که به معنا حداکثر ظرفیت است. در برش ها تابع درونی به نام `cap` وجود داره که ظرفیت برش را نشون میده! در مثال زیر برشی با ظرفیت اولیه ۴ و حداکثری ۸ ساخته می شود.

<https://play.golang.org/p/p3ewucUjFer>

کد شماره ۱۳

```
package main

import (
    "fmt"
)

func main() {
    mySlice := make([]int, 4, 8)
    fmt.Printf("Initial Length: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))
    fmt.Printf("Contents: %v\n", mySlice)
}

// Output :
// Initial Length: 4
// Capacity: 8
// Contents: [0 0 0 0]
```

خوب چون حداکثر ظرفیت ۸ بود اگر بخوایم خانه ۹ را پر کنیم با خطا مواجه میشیم:

<https://play.golang.org/p/BtiZKrqylmq>

کد شماره ۱۴

```
package main

import (
    "fmt"
)

func main() {

    mySlice := make([]int, 0, 8)
    mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))

    mySlice[8] = 19

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))
}
```

```

/*
Output :

Contents: [1 3 5 7 9 11 13 17]
Number of Items: 8
Capacity: 8
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
    /home/amir/Desktop/Code/go/Go-Succinctly/listing22.go:18 +0x464
exit status 2
*/

```

اما همان طور که بیان شد یکی دیگر راه های افزودن تابع درونی `append` معرفی شد یکی از کارهای جادویی این تابع اینکه اگر خانه ای بیشتر از ظرفیت بسازید خطا نمیده و خودکار ظرفیت رو دو برابر قرار می دهد .

<https://play.golang.org/p/yE-1Rhccbv>

کد شماره ۱۵

```

package main

import (
    "fmt"
)

func main() {

    mySlice := make([]int, 0, 8)
    mySlice = append(mySlice, 1, 3, 5, 7, 9, 11, 13, 17)

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))

    mySlice = append(mySlice, 19)

    fmt.Printf("Contents: %v\n", mySlice)
    fmt.Printf("Number of Items: %d\n", len(mySlice))
    fmt.Printf("Capacity: %d\n", cap(mySlice))

}

/*
Output :

Contents: [1 3 5 7 9 11 13 17]
Number of Items: 8
Capacity: 8
Contents: [1 3 5 7 9 11 13 17 19]
Number of Items: 9
Capacity: 16
*/

```

دیگر نگران ظرفیت برش ها نباشید !

**منابع بیشتر**

Go Slices: usage and internals <https://blog.golang.org/go-slices-usage-and-internals>

Map در واقع مجموعه ای از دیتا ها به صورت key-value هستند با ساختار `associative data type` همانند دیکشنری در پایتون .

<https://play.golang.org/p/U67R66Cab8r>

کد شماره ۶

```
// _Maps_ are Go's built-in [associative data type]
// (http://en.wikipedia.org/wiki/Associative_array)
// (sometimes called _hashes_ or _dicts_ in other languages).

package main

import "fmt"

func main() {

    // To create an empty map, use the builtin `make`:
    // `make(map[key-type]val-type)`.
    m := make(map[string]int)

    // Set key/value pairs using typical `name[key] = val`
    // syntax.
    m["k1"] = 7
    m["k2"] = 13

    // Printing a map with e.g. `fmt.Println` will show all of
    // its key/value pairs.
    fmt.Println("map:", m)

    // Get a value for a key with `name[key]`.
    v1 := m["k1"]
    fmt.Println("v1: ", v1)

    // The builtin `len` returns the number of key/value
    // pairs when called on a map.
    fmt.Println("len:", len(m))

    // The builtin `delete` removes key/value pairs from
    // a map.
    delete(m, "k2")
    fmt.Println("map:", m)

    // The optional second return value when getting a
    // value from a map indicates if the key was present
    // in the map. This can be used to disambiguate
    // between missing keys and keys with zero values
    // like `0` or `""`. Here we didn't need the value
    // itself, so we ignored it with the _blank identifier_
    // `_`.
    _, prs := m["k2"]
    fmt.Println("prs:", prs)

    // You can also declare and initialize a new map in
```

```
// the same line with this syntax.
n := map[string]int{"foo": 1, "bar": 2}
fmt.Println("map:", n)
}
```

خروجی :

```
$ go run maps.go
map: map[k1:7 k2:13]
v1: 7
len: 2
map: map[k1:7]
prs: false
map: map[foo:1 bar:2]
```

نکات :

- همان طور که در نگاه اول مشخص همیشه نحوه تعریف Map مشابه Slice هستند با این تفاوت که سینتکس آن به صورت `make(map[key-])` است.
- برای مقداردهی به یک کلید می توان از سینتکس `name[key] = val` و برای دریافت مقدار یک کلید می توان از `name[key]` استفاده نمود.
- برای چاپ می توان از `fmt.Println` استفاده کرد و خروجی همانند `map[k1:7 k2:13]` دریافت کرد همانند کد بالا.
- همانند `array` و `slice` تابع `len` برای `map` نیز قابل استفاده است تا بتوانید تعداد کلید ها رو بشمارید.
- تابع داخلی `delete` که شامل دو پارامتر هستند ( پارامتر اول دریافت `map` و پارامتر دوم کلید ) به ما این دسترسی را می دهد تا کلید خاصی را حذف کنیم.
- قبلا بیان شد که اگر نوع مقدار مثلا `int` قرار داده شد و مقداری مشخص نشد ( یا وجود نداشت ) برای ما `0` بر می گرده ! اما اگر ما واقعا `0` را خودمان به عنوان متغیر قرار داده باشیم و بخوایم بسنجیم ، از کجا میشه فهمید این `0` را ما گذاشتیم یا در صورت نبودن مقدار برگشت داده شده . بعضی جاها عبارتی به نام `comma OK` ذکر کردن که به این معناس وقتی با سینتکس `name[key]` شروع به دریافت مقدار می کنید در واقع دو خروجی وجود دارد و اون خروجی دومی به صورت `boolean` به ما میگه که ایم کلید مقدار دهی شده یا خیر . در مثال بالا نمونه ای ذکر شده است اما به جای `ok` از نام `prs` استفاده شده .
- در آخرین مثال کد بالا نحوه تعریف `map` و مقدار دهی در یک خط مشخص شده است .
- شما می توانید با کلمه کلیدی `range` روی آرایه ها ، برش ها و `map` ها به کمک `for` پروسه `iterate` را انجام بدید اما نکته حایز اهمیتی که وجود دارد اینکه بر اساس طراحی زبان Go این خروجی بر روی `map` نظم نداره و هر دفعه به صورت راندوم نمایش داده می شود .

This is by design: the Go team didn't want programmers to rely upon an ordering that was essentially unreliable, so they randomized the iteration order to make that impossible.

مثال :

<https://play.golang.org/p/jYhjajBm7pQ>

کد شماره ۱۷

```
package main

import "fmt"

func main() {
    actor := map[string]int{
        "Paltrow": 43,
        "Cruise": 53,
    }
}
```

```

        "Redford": 79,
        "Diaz": 43,
        "Kilmer": 56,
        "Pacino": 75,
        "Ryder": 44,
    }

    for i := 1; i < 4; i++ {
        fmt.Printf("\nRUN NUMBER %d\n", i)
        for key, value := range actor {
            fmt.Printf("%s : %d years old\n", key, value)
        }
    }
}

```

```
/*
```

```
Output :
```

```
RUN NUMBER 1
```

```
Redford : 79 years old
```

```
Diaz : 43 years old
```

```
Kilmer : 56 years old
```

```
Pacino : 75 years old
```

```
Ryder : 44 years old
```

```
Paltrow : 43 years old
```

```
Cruise : 53 years old
```

```
RUN NUMBER 2
```

```
Paltrow : 43 years old
```

```
Cruise : 53 years old
```

```
Redford : 79 years old
```

```
Diaz : 43 years old
```

```
Kilmer : 56 years old
```

```
Pacino : 75 years old
```

```
Ryder : 44 years old
```

```
RUN NUMBER 3
```

```
Cruise : 53 years old
```

```
Redford : 79 years old
```

```
Diaz : 43 years old
```

```
Kilmer : 56 years old
```

```
Pacino : 75 years old
```

```
Ryder : 44 years old
```

```
Paltrow : 43 years old
```

```
*/
```

راه حل : بهترین راه حل تغییر ساختار هستش به این صورت که ابتدا اطلاعات را به ساختار برش ها انتقال بدیم و سپس با پکیج `sort` آن ها رو مرتبط کنیم .



```
package main

import (
    "fmt"
    "sort"
)

func main() {
    actor := map[string]int{
        "Paltrow": 43,
        "Cruise": 53,
        "Redford": 79,
        "Diaz": 43,
        "Kilmer": 56,
        "Pacino": 75,
        "Ryder": 44,
    }

    // Store the keys in a slice
    var sortedActor []string
    for key := range actor {
        sortedActor = append(sortedActor, key)
    }
    // Sort the slice alphabetically
    sort.Strings(sortedActor)

    /* Retrieve the keys from the slice and use
       them to look up the map values */
    for _, name := range sortedActor {
        fmt.Printf("%s : %d years old\n", name, actor[name])
    }
}

/*
Output :

Cruise : 53 years old
Diaz : 43 years old
Kilmer : 56 years old
Pacino : 75 years old
Paltrow : 43 years old
Redford : 79 years old
Ryder : 44 years old
*/
```

## ترفند :

داخل `if` ما می توانیم متغیری تعریف کنیم و سپس با `;` اون را جدا کنیم و به ارزشیابی بپردازیم . صرفا برای آشنایی برای چنین سینتکس هایی بر روی `if` `statement` ها مطرح شد .

<https://play.golang.org/p/ueJmA28U5EF>

کد شماره ۱۹

```
package main

import "fmt"

func main() {

    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13

    delete(m, "k2")

    if _, ok := m["k2"]; ok {
        fmt.Println("Ok")
    } else {
        fmt.Println("NotOk")
    }
}
```

کلمه کلیدی `range` به ما اجازه `iterate` بر روی ساختار داده هایی که تاکنون آموختیم را می دهد .

[https://play.golang.org/p/SkL\\_AS-1Jd](https://play.golang.org/p/SkL_AS-1Jd)

کد شماره ۲۰

```
// _range_ iterates over elements in a variety of data
// structures. Let's see how to use `range` with some
// of the data structures we've already learned.

package main

import "fmt"

func main() {

    // Here we use `range` to sum the numbers in a slice.
    // Arrays work like this too.
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // `range` on arrays and slices provides both the
    // index and value for each entry. Above we didn't
    // need the index, so we ignored it with the
    // blank identifier `_`. Sometimes we actually want
    // the indexes though.
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // `range` on map iterates over key/value pairs.
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // `range` can also iterate over just the keys of a map.
    for k := range kvs {
        fmt.Println("key:", k)
    }

    // `range` on strings iterates over Unicode code
    // points. The first value is the starting byte index
    // of the `rune` and the second the `rune` itself.
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

```

    }
}

```

خروجی :

```

$ go run range.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
0 103
1 111

```

نکات :

- در مثال اول یعنی متغیر `nums` ما بر روی یک برش یا `slice` عمل `iterate` را انجام دادیم .
- در واقع کلمه کلیدی `range` دو خروجی دارد که در `array` و `slice` ها اولی `index` و دومی مقدار خانه است . در مثال اول ما متغیر `index` را با قرار دادن `blank identifier` یا همان `_` گفتیم که استخراج نکن چون اگر استخراج کنیم و استفاده نکنیم کامپایلر گو خطا می دهد .
- در متغیر `kvs` ما یک `map` با مقدار اولیه ساختیم و آن را `iterate` کردیم اما تفاوت خروجی `range` در `map` این است که اولین پارامتر کلید و دومین پارامتر مقدار است .
- برای `map` ها می توانیم فقط یک خروجی تحت عنوان کلید از `range` بگیریم .
- به کمک `range` همچنین می توانیم بر روی رشته ها `iterate` کنیم ، در اینجا پارامتر اول ایندکس و پارامتر دوم `Unicode code points` را بر می گردونه !

توابع یکی از قابلیت های مهم هر زبانی محسوب می شود و در گولنگ دارای سینتکس بسیار ساده ای هستند به مثال های زیر توجه کنید .

<https://play.golang.org/p/JpNfvtAjrc4>

کد شماره ۲۱

```
// _Functions_ are central in Go. We'll learn about
// functions with a few different examples.

package main

import "fmt"

// Here's a function that takes two `int`s and returns
// their sum as an `int`.
func plus(a int, b int) int {

    // Go requires explicit returns, i.e. it won't
    // automatically return the value of the last
    // expression.
    return a + b
}

// When you have multiple consecutive parameters of
// the same type, you may omit the type name for the
// like-typed parameters up to the final parameter that
// declares the type.
func plusPlus(a, b, c int) int {
    return a + b + c
}

func main() {

    // Call a function just as you'd expect, with
    // `name(args)`.
    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

خروجی :

```
$ go run functions.go
1+2 = 3
1+2+3 = 6
```

## نکات :

- برای تعریف یک تابع ساده به الگو پیاده سازی تابع `plus` توجه کنید ، همیشه باید دیتا تایپ پارامترهای ورودی مشخص شوند در این مثال ما دو پارامتر `a` و `b` را با تایپ `int` به تابع منتقل می کنیم .
- بعد از تعیین پارامتر های ورودی و قبل از قرار دادن کد داخل براکت برای تابع کلید واژه `int` قرار گرفته به این معنا که مقدار خروجی تابع باید دیتا تایپ `int` باشد .
- در صورتی که دیتا تایپ تمام پارامترهای ورودی یکسان باشد می توان همانند تابع `plusPlus` کلید واژه دیتا تایپ را در آخر همه پارامترها ذکر کرد .
- برای صدا زدن یک تابع کافیسست از سینتکس `name(args)` استفاده کرد .
- توابع در گولنگ ویژگی های زیادی دارند که در آموزش های بعدی معرفی خواهند شد .

## الگو دیگر تعریف تابع :

<https://play.golang.org/p/bOG8oBahGnm>

کد شماره ۲۲

```
package main

import "fmt"

func main() {

    plus := func(a int, b int) int {

        return a + b
    }

    res := plus(1, 2)
    fmt.Println("1+2 =", res)
}
```

خروجی :

```
$ go run functions.go
1+2 = 3
```

## نکات :

- به هیچ عنوان نمی شود داخل یک تابع ، تابع دیگری ساخت مگر با استفاده از این سینتکس .
- در استفاده از این سینتکس توجه داشته باشید برای `Recursion` نمی توانید از این قابلیت استفاده کنید ، همانند زبان های `es` و `ts` از نام دوم نیز نمی توان استفاده کرد .

## Multiple Return Values در گولنگ

زبان برنامه نویسی گو به ما اجازه می دهد تا هر تابع چند خروجی داشته باشد برای مثال ما می توانیم خروجی تابع و خطا را همزمان از تابع بگیریم و مثل بعضی زبان ها نیاز نیست یک خروجی وجود داشته باشد و ما بسنجیم آیا خروجی خطا هست یا نه !

<https://play.golang.org/p/chwFmr5dG1>

کد شماره ۲۳

```
// Go has built-in support for _multiple return values_.
// This feature is used often in idiomatic Go, for example
// to return both result and error values from a function.

package main

import "fmt"

// The `(int, int)` in this function signature shows that
// the function returns 2 `int`s.
func vals() (int, int) {
    return 3, 7
}

func main() {

    // Here we use the 2 different return values from the
    // call with _multiple assignment_.
    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    // If you only want a subset of the returned values,
    // use the blank identifier `_`.
    _, c := vals()
    fmt.Println(c)
}
```

خروجی :

```
$ go run multiple-return-values.go
3
7
7
```

نکات :

- در کد بالا تابعی با عنوان **vals** ساختیم اما این تابع دو خروجی دارد بنابراین نحوه تعریف نوع داده خروجی متفاوت است نسبت به یک تابع معمولی و با سینتکس (**data-type, data-type, ...**) تعریف می شود، همچنین خروجی آن را با **,** باید از هم جدا کرد.
- در مثال اول ما به کمک دو متغیر **a** و **b** دو خروجی تابع **vals** را دریافت کردیم.
- در مثال بعدی ما فقط خروجی دوم را با متغیر **c** استخراج کردیم و بجای خروجی اول از **\_ blank identifier** استفاده کردیم تا مقدار خروجی اول دریافت یا استخراج نشود.

## Variadic Functions در گولنگ

یکی از جذاب ترین قابلیت های گو وجود Variadic Function (توابع متنوع) هاست که به شما اجازه می دهد هر چقدر پارامتر نیاز دارید بدون تعریف وارد تابع کنید همانند Rest-parameter ها در es6. معمول ترین Variadic Function در گو تابع `fmt.Println` می باشد.

<https://play.golang.org/p/7f0JlVhToDD>

کد شماره ۲۴

```
// [_Variadic functions_](http://en.wikipedia.org/wiki/Variadic_function)
// can be called with any number of trailing arguments.
// For example, `fmt.Println` is a common variadic
// function.

package main

import "fmt"

// Here's a function that will take an arbitrary number
// of `int`s as arguments.
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {

    // Variadic functions can be called in the usual way
    // with individual arguments.
    sum(1, 2)
    sum(1, 2, 3)

    // If you already have multiple args in a slice,
    // apply them to a variadic function using
    // `func(slice...)` like this.
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

خروجی:

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```



- در مثال بالا تابع `sum` به کمک `...` که قبل نوع داده قرار گرفته به ما اجازه می دهد چندین پارامتر را به داخل تابع انتقال دهیم .
- برای صدا زدن این توابع می تواند به صورت `funcName(param1, param2, ...)` عمل کرد .
- اگر شما داده ای با ساختار `slice` دارید به راحتی به کمک `operator ...` می توانید با سینتکس `funcName(mySlice...)` آن را به تابع انتقال دهید .

## Closures در گولنگ

به دلیل اینکه `Closures` در همه زبان های برنامه نویسی وجود دارد قبل از شروع مطالعه بخشی از ویکی پدیا خالی از لطف نیست .

### Closure (computer programming)

The following program fragment defines a higher-order function (function returning a function) `add` with a parameter `x` and a nested function `addX`. The nested function `addX` has access to `x`, because `x` is in the lexical scope of `addX`. The function `add` returns a closure; this closure contains (1) a reference to the function `addX`, and (2) a copy of the environment around `addX` in which `x` has the value given in that specific invocation of `add`.

```
function add(x)
  function addX(y)
    return y + x
  return addX

variable add1 = add(1)
variable add5 = add(5)

assert add1(3) = 4
assert add5(3) = 8
```

[https://en.wikipedia.org/wiki/Closure\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

زبان گو از توابع ناشناس `anonymous functions` در فرم `closures` بهره می برد . توابع ناشناس این امکان را می دهند تا تابعی بدون نام به صورت `inline` بسازیم .

<https://play.golang.org/p/zb93qzV6iN3>

کد شماره ۲۵

```
// Go supports [_anonymous functions_](http://en.wikipedia.org/wiki/Anonymous_function),
// which can form <a href="http://en.wikipedia.org/wiki/Closure_(computer_science)">
<em>closures</em></a>.
// Anonymous functions are useful when you want to define
// a function inline without having to name it.

package main

import "fmt"

// This function `intSeq` returns another function, which
// we define anonymously in the body of `intSeq`. The
```

```
// returned function _closes over_ the variable `i` to
// form a closure.
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {

    // We call `intSeq`, assigning the result (a function)
    // to `nextInt`. This function value captures its
    // own `i` value, which will be updated each time
    // we call `nextInt`.
    nextInt := intSeq()

    // See the effect of the closure by calling `nextInt`
    // a few times.
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    // To confirm that the state is unique to that
    // particular function, create and test a new one.
    newInts := intSeq()
    fmt.Println(newInts())
}
```

خروجی :

```
$ go run closures.go
1
2
3
1
```

## نکات

- ما با تعریف یک تابع و نوع داده خروجیش آشنا شدیم اما اگر خروجی تابع یک تابع دیگر بود در این حالت باید از سینتکس `func() data-type` استفاده کنیم ، همانند تابع `intSeq` . در اینجا `data-type` خروجی ، خروجی تابع درونیست .
- در تابع `intSeq` خروجی ما یک تابع دیگر است و چون تابع درونی به متغیر `i` که در تابع بالایی خودش وجود داره مرتبط است میشه بیان داشت خروجی تابع `intSeq` یک فرم از `closure` است .
- داخل تابع `main` متغیری تحت عنوان `nextInt` تعریف کردیم و داخل آن تابع `intSeq` را قرار دادیم حالا هر چقدر ما `nextInt` را صدا یا `invoke` کنیم `state` ما یا همون متغیر `i` درون تابع `intSeq` آپدیت می شود .
- برای اینکه بفهمیم این `state` مخصوص تابع اصلیس دوباره آن را تحت عنوان `newInts` ساختیم و `invoke` کردیم .

همه برنامه نویسان به اصطلاحاتی چون **Recursion** در علوم کامپیوتر آشنا هستند و گولنگ این نیز از این قابلیت پشتیبانی می کند و در زیر مثال کلاسیک فاکتوریل را داریم .

<https://play.golang.org/p/RFn-rf42ap>

کد شماره ۲۶

```
// Go supports
// <a href="http://en.wikipedia.org/wiki/Recursion_(computer_science)"><em>recursive
functions</em></a>.
// Here's a classic factorial example.

package main

import "fmt"

// This `fact` function calls itself until it reaches the
// base case of `fact(0)`.
func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))
}
```

خروجی :

```
$ go run recursion.go
5040
```

نکات :

- وقتی توابع از درون خودشان را صدا می زنن به اصطلاح مبحث **recursion** (بازگشتی) به وجود می آید .
- در مثال بالا تا زمانی که **n-1** مساوی صفر نشود این تابع خودش را صدا می زند .

In computing, a memory address is a reference to a specific memory location used at various levels by software and hardware. Memory addresses are fixed-length sequences of digits conventionally displayed and manipulated as unsigned integers. Such numerical semantic bases itself upon features of CPU (such as the instruction pointer and incremental address registers), as well upon use of the memory like an array endorsed by various programming languages.

[https://en.wikipedia.org/wiki/Memory\\_address](https://en.wikipedia.org/wiki/Memory_address)

اگر تا حالا با زبان های c و ++ کار نکردید احتمالا با پوینترها آشنا نیستید . پوینتر برای انتقال آدرس حافظه memory location استفاده می شود .

<https://play.golang.org/p/KdE4TBbUL2>

کد شماره ۲۷

```
// Go supports <em><a
href="http://en.wikipedia.org/wiki/Pointer_(computer_programming)">pointers</a></em>,
// allowing you to pass references to values and records
// within your program.

package main

import "fmt"

// We'll show how pointers work in contrast to values with
// 2 functions: `zeroval` and `zeroptr`. `zeroval` has an
// `int` parameter, so arguments will be passed to it by
// value. `zeroval` will get a copy of `ival` distinct
// from the one in the calling function.
func zeroval(ival int) {
    ival = 0
}

// `zeroptr` in contrast has an `*int` parameter, meaning
// that it takes an `int` pointer. The `*iptr` code in the
// function body then _dereferences_ the pointer from its
// memory address to the current value at that address.
// Assigning a value to a dereferenced pointer changes the
// value at the referenced address.
func zeroptr(iptr *int) {
    *iptr = 0
}

func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    // The `&i` syntax gives the memory address of `i`,
    // i.e. a pointer to `i`.
    zeroptr(&i)
```

```

fmt.Println("zeroptr:", i)

// Pointers can be printed too.
fmt.Println("pointer:", &i)
}

```

خروجی :

```

$ go run pointers.go
initial: 1
zeroval: 1
zeroptr: 0
pointer: 0x42131100

```

نکات :

- برای ایجاد پوینتر از کارکتر `&` استفاده می شود و در مثال بالا دو تابع داریم اولی `zeroval` که یک تابع معمولی است و تابع `zeroptr` که کنار نوع داده ورودی علامت `*` مشاهده می کنید به این معنا که ورودی به کمک پوینتر رفرنس داده شده است و در ادامه هنگام مقدار دهی نیز از `*` استفاده شده تا مقدار در آدرس حافظه ای که رفرنس داده شده ذخیره شود .
- در تابع `main` با مثال پوینتر پیاده سازی شده و قابل درک است و در آخرین خط مشاهده می کنیم که آدرس حافظه پوینتر قابل چاپ است .

یکی از ساختارهای کاربردی در گولنگ **Struct** هاست ، این ساختار اجازه می دهد تا مجموعه از داده ها را به صورت کالکشن داشته باشیم چیزی شبیه به **object** در زبان های دیگر .

<https://play.golang.org/p/OMCP5KFC10>

کد شماره ۲۸

```
// Go's _structs_ are typed collections of fields.
// They're useful for grouping data together to form
// records.

package main

import "fmt"

// This `person` struct type has `name` and `age` fields.
type person struct {
    name string
    age  int
}

func main() {

    // This syntax creates a new struct.
    fmt.Println(person{"Bob", 20})

    // You can name the fields when initializing a struct.
    fmt.Println(person{name: "Alice", age: 30})

    // Omitted fields will be zero-valued.
    fmt.Println(person{name: "Fred"})

    // An `&` prefix yields a pointer to the struct.
    fmt.Println(&person{name: "Ann", age: 40})

    // Access struct fields with a dot.
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name)

    // You can also use dots with struct pointers - the
    // pointers are automatically dereferenced.
    sp := &s
    fmt.Println(sp.age)

    // Structs are mutable.
    sp.age = 51
    fmt.Println(sp.age)
}
```

```
$ go run structs.go
{Bob 20}
{Alice 30}
{Fred 0}
&{Ann 40}
Sean
50
51
```

## نکات

- برای ساخت یک `struct` از کلمه کلیدی `type` به صورت `struct {type StructName}` استفاده می کنیم .
- برای ساخت یک `instance` از ساختار `struct` دو الگو بلند و کوتاه وجود دارد که الگو بلند مقدار دهی با کلید است و الگو کوتاه آن مقداردهی با توجه به کلید در خود ساختار است .

```
// Short way
fmt.Println(person{"Bob", 20})

// Long way
fmt.Println(person{name: "Alice", age: 30})
```

- اگر در ساخت یک `instance` برای کلید مقداری قرار داده نشود با توجه به نوع داده مقدار ثابتی می گیرد .
- برای قرار دادن پویتر کفایست از اپراتور `&` قبل شروع تعریف آن استفاده کنیم .
- برای خواندن و مقدار دهی می توانید از `yourStruct.paramOne = sth` به صورت استفاده کنید .
- `struct` ساختاری قابل تغییر است .
- اگر نیاز شد `instance` ی از این ساختار بسازیم و پویتر بهش وصل کنیم سینتکس کوتاهی برای آن وجود دارد :

```
var MyPerson *person
MyPerson = new(person)

// Short way
MyPerson := new(person)
```

**methods** ها توابعی هستند که بر روی ساختار **struct** ساخته می شوند برای مثال فکر کنید ساختار **struct** یک **object** در جاوا اسکریپت هستند و ما می خواهیم یک **prototype** که تابع هست به اون اضافه کنیم همین! این قابلیت بوجود اومد زیرا همیشه لازم میشه داده های داخل ساختار **struct** را با توابعی تغییر داد.

[https://play.golang.org/p/254m\\_9Yjwa](https://play.golang.org/p/254m_9Yjwa)

کد شماره ۲۹

```
// Go supports _methods_ defined on struct types.

package main

import "fmt"

type rect struct {
    width, height int
}

// This `area` method has a _receiver type_ of `*rect`.
func (r *rect) area() int {
    return r.width * r.height
}

// Methods can be defined for either pointer or value
// receiver types. Here's an example of a value receiver.
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func main() {
    r := rect{width: 10, height: 5}

    // Here we call the 2 methods defined for our struct.
    fmt.Println("area: ", r.area())
    fmt.Println("perim:", r.perim())

    // Go automatically handles conversion between values
    // and pointers for method calls. You may want to use
    // a pointer receiver type to avoid copying on method
    // calls or to allow the method to mutate the
    // receiving struct.
    rp := &r
    fmt.Println("area: ", rp.area())
    fmt.Println("perim:", rp.perim())
}
```



خروجی :

```
$ go run methods.go
area: 50
perim: 30
area: 50
perim: 30
```

نکات :

- در کد بالا دو `method` با نام های `area` و `perim` برای ساختار `rect` با سینتکس `func (AnyVariable StructName) MethodName() Data-Type` ساخته شده است .

```
// Go automatically handles conversion between values
// and pointers for method calls. You may want to use
// a pointer receiver type to avoid copying on method
// calls or to allow the method to mutate the
// receiving struct.
```

## Embedded types در گولنگ

حالا که با ساختار `struct` و `method` های آن آشنا شدید باید بدانیم هر `type` از داده می تواند `type` داده دیگری را بسازد اما اون `type` ناشناخته خواهد بود ، به این معنا که نمی توانید اسمی به آن بدهید بجاش شما می توانید اونو با نام `type` آخر صدا کنید .  
برای مثال در زیر `type` داده ای برای کد تخفیف پیاده کردیم :

```
type Discount struct {
    percent    float32
    promotionId string
}
```

حالا در زیر `type` ی برای یک تخفیف ویژه + تخفیف اولیه میسازیم :

```
type ManagersSpecial struct {
    Discount // The embedded type
    extraoff float32
}
```

برای استفاده از نوع داده بالا می توانیم به شکل زیر عمل کنیم .

```
januarySale := Discount{15.00, "January"}
managerSpecial := ManagersSpecials{januarySale, 10.00}
```

همچنین برای دسترسی به نوع داده `embed` شده همیشه از روش زیر استفاده کرد :

```
managerSpecial.Discount.percent // 15.00
managerSpecial.Discount.promotionId // "January"
```

اگر ما `method` ی به `Discount` متصل کنیم برای صدا زدن آن باید از سینتکس زیر استفاده کرد :

```
managerSpecial.Discount.someMethod(someParameter)
```

### مثال عملی

<https://play.golang.org/p/ufYS79Dx4BO>

کد شماره ۳۰

```
package main

import (
    "fmt"
)

type Discount struct {
    percent    float32
```

```
        promotionId string
    }

    type ManagersSpecial struct {
        Discount
        extraoff float32
    }

    func main() {

        normalPrice := float32(99.99)

        januarySale := Discount{15.00, "January"}
        managerSpecial := ManagersSpecial{januarySale, 10.00}

        discountedPrice := januarySale.Calculate(normalPrice)
        managerDiscount := managerSpecial.Calculate(normalPrice)

        fmt.Printf("Original price: $%4.2f\n", normalPrice)
        fmt.Printf("Discount percentage: %2.2f\n",
            januarySale.percent)
        fmt.Printf("Discounted price: $%4.2f\n", discountedPrice)
        fmt.Printf("Manager's special: $%4.2f\n", managerDiscount)
    }

    func (d Discount) Calculate(originalPrice float32) float32 {
        return originalPrice - (originalPrice / 100 * d.percent)
    }

    func (ms ManagersSpecial) Calculate(originalPrice float32) float32 {
        return ms.Discount.Calculate(originalPrice) - ms.extraoff
    }
}
```

`interface` همان طوری که از اسمش پیداست کارش اینکه مطمئن بشه یکسری دیتا ها وجود داشته باشه و نحوه تعریف آن مشابه `struct` است. در مثال پایین روی دو ساختار `rect` و `circle` ما دو متد `area` و `perim` ساختیم و داخل تابع `measure` ما نوع داده ورودی برای پارامتر `g` را اینترفیس `geometry` قرار دادیم و با این کار به تابع می‌گیم اگر ساختار `g` از این اینترفیس پیروی نکرد یعنی دو تابع `area` و `perim` نداشت اون موقع خطا بده!

<https://play.golang.org/p/313UebA3rD>

کد شماره ۳۱

```
// _Interfaces_ are named collections of method
// signatures.

package main

import "fmt"
import "math"

// Here's a basic interface for geometric shapes.
type geometry interface {
    area() float64
    perim() float64
}

// For our example we'll implement this interface on
// `rect` and `circle` types.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

// To implement an interface in Go, we just need to
// implement all the methods in the interface. Here we
// implement `geometry` on `rect`s.
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// The implementation for `circle`s.
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}

// If a variable has an interface type, then we can call
// methods that are in the named interface. Here's a
// generic `measure` function taking advantage of this
// to work on any `geometry`.
```

```
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The `circle` and `rect` struct types both
    // implement the `geometry` interface so we can use
    // instances of
    // these structs as arguments to `measure`.
    measure(r)
    measure(c)
}
```

خروجی :

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

## Empty interface در گولنگ

کد زیر را می‌گن `empty interface` :

```
interface {}
```

برخی زبان‌های دیگر مثل .Net و Java بهش `marker interface` می‌گن . در نگاه اول احمقانه به نظر می‌آید ولی خیلی کاربردیست . همان طور که می‌دونیم کامپایلر گولنگ به خاطر اینکه زبان `static-type` هست خیلی حساس عمل می‌کند و اگر نوع داده مشخص نشده باشد ما با خطا مواجه می‌شویم اما گاهی هنگام اجرا برنامه ما نمی‌دونیم چه نوع داده‌ای دریافت می‌کنیم پس اینترفیس خالی به ما کمک می‌کند این مشکل رو حل کنیم .

مثال :

<https://play.golang.org/p/7uw26q10mbY>

کد شماره ۳۲

```
package main

import (
    "fmt"
)

func main() {
    displayType(42)
    displayType(3.14)
    displayType("ここでは文字列です")
}

func displayType(i interface{}) {
    switch theType := i.(type) {
    case int:
        fmt.Printf("%d is an %$$$$$$$$integer\n", theType)
    case float64:
        fmt.Printf("%f is a 64-bit float\n", theType)
    case string:
        fmt.Printf("%s is a string\n", theType)
    default:
        fmt.Printf("I don't know what %v is\n", theType)
    }
}
```

## Type assertion در گولنگ

اگر از `empty interface` برای دسترسی به مقداری که نوع آن نامعلوم هست استفاده کردیم حتما برای تعیین نوع داده باید از این قابلیت استفاده کنیم . در واقع شما نمی توانید داده ای که نوع آن معلوم نیست را به راحتی استفاده کنید و باید به جاش نوع داده رو برای آن با این روش مشخص کنید .

```
var anything interface{} = "something"
aString := anything.(string)
```

[https://play.golang.org/p/njiry2LJ\\_qk](https://play.golang.org/p/njiry2LJ_qk)

کد شماره ۳۳

```
package main

import (
    "fmt"
)

func main() {
    var anything interface{} = "something"
    aString := anything.(string)
    fmt.Println(aString)
}

// Output :
// something
```

اما اگر نوع داده را اشتباه وارد کنیم با `panic` رو به رو میشویم !

<https://play.golang.org/p/9KT-MkAf3UJ>

کد شماره ۳۴

```
package main

import (
    "fmt"
)

func main() {
    var anything interface{} = "something"
    aInt := anything.(int)
    fmt.Println(aInt)
}

/*
Output :

panic: interface conversion: interface is string, not int
goroutine 1 [running]:
panic(0xda640, 0xc820010180)
    /usr/local/go/src/runtime/panic.go:464 +0x3e6
main.main()
    ../src/hello/main.go:9 +0xa8
exit status 2
*/
```

خوب اما حالا برای مدیریت بهتر خطایی که ممکنه بوجود بیاد باید توجه داشته باشیم این تابع دو خروجی دارد ، خروجی اول داده تبدیل شده است و خروجی دوم متغیری که با نوع `bool` به ما نشان می دهد که آیا نوع داده درست انتخاب شده است یا نه .

[https://play.golang.org/p/lcR\\_JPZvsSM](https://play.golang.org/p/lcR_JPZvsSM)

کد شماره ۳۵

```
package main

import (
    "fmt"
)

func main() {
    var anything interface{} = "something"
    aInt, ok := anything.(int)
    if !ok {
        fmt.Println("Cannot turn input into an integer")
    } else {
        fmt.Println(aInt)
    }
}

// Output :
// Cannot turn input into an integer
```



- Go by Example <https://gobyexample.com>
- Go Succinctly [https://www.syncfusion.com/ebooks/go\\_succinctly](https://www.syncfusion.com/ebooks/go_succinctly)
- Go in Action <https://www.manning.com/books/go-in-action>

نگارش ۱.۰.۰

خرداد ۱۳۹۷