Part 11: Arrays and Slices

20 APRIL 2017

Welcome to the part 11 of Golang tutorial series. In this tutorial we deal with Arrays and Slices in Go.

Arrays

An array is a collection of elements that belong to the same type. For example the collection of integers 5, 8, 9, 79, 76 form an array. Mixing values of different types, for example an array that contains both strings and integers is not allowed in Go.

Declaration

An array belongs to type `n[T]`. `n` denotes the number of elements in an array and `T` represents the type of each element. The number of elements `n` is also a part of the type(We will discuss this in more detail shortly.)

There are different ways to declare arrays. Lets look at them one by one.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7
8   func main() {
9       var a [3]int //int array with length 3
10      fmt.Println(a)
11  }
```

var a [3]int declares a integer array of length 3. **All elements in an array are automatically assigned the zero value of the array type**. In this case `a` is an integer array and hence all elements of `a` are assigned to `0`, the zero value of int. Running the above program will **output** `[0 0 0]`.

The index of an array starts from `0` and ends at `length - 1`. Lets assign some values to the above array.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7
8   func main() {
9       var a [3]int //int array with length 3
10      a[0] = 12 // array index starts at 0
11      a[1] = 78
12      a[2] = 50
13      fmt.Println(a)
14  }
```

a[0] assigns value to the first element of the array. The program will **output** `[12 78 50]`

Lets create the same array using the **short hand declaration**.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       a := [3]int{12, 78, 50} // short hand declarat
9       fmt.Println(a)
10  }
```

The program above program will print the same **output** `[12 78 50]`

It is not necessary that all elements in an array have to be assigned a value during short hand declaration.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       a := [3]int{12}
9       fmt.Println(a)
10  }
```

In the above program in line no. 8 `a := [3]int{12}` declares an array of length 3 but is provided with only one value `12`. The remaining 2 elements are assigned `0` automatically. The program will **output** `[12 0 0]`

You can even ignore the length of the array in the declaration and replace it with `...` and let the compiler find the length for you. This is done in the following program.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       a := [...]int{12, 78, 50} // ... makes the comp
9       fmt.Println(a)
10  }
```

The size of the array is a part of the type. Hence `[5]int` and `[25]int` are distinct types. Because of this, arrays cannot be resized. Don't worry about this restriction since `slices` exist to overcome this.

```
1   package main
2
3   func main() {
4       a := [3]int{5, 78, 8}
5       var b [5]int
6       b = a //not possible since [3]int and [5]int ar
7   }
```

The above program in line no. 6, we are trying to assign a variable of type `[3]int` to a variable of type `[5]int` which is not allowed and hence the compiler will throw error *main.go:6: cannot use a (type [3]int) as type [5]int in assignment.*

Arrays are value types

Arrays in Go are value types and not reference types. This means that when they are assigned to a new variable, a copy of the original array is assigned to the new variable. If changes are made to the new variable, it will not be reflected in the original array.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       a := [...]string{"USA", "China", "India", "Gerr
7       b := a // a copy of a is assigned to b
8       b[0] = "Singapore"
9       fmt.Println("a is ", a)
10      fmt.Println("b is ", b)
11  }
```

In the above program in line no. 7, a copy of `a` is assigned to `b`. In line no. 8, the first element of `b` is changed to `Singapore`. This will not reflect in the original array `a`. The program will **output**,

```
a is [USA China India Germany France]
b is [Singapore China India Germany France]
```

Similarly when arrays are passed to functions as parameters, they are passed by value and the original array in unchanged.

```
1   package main
2
3   import "fmt"
4
5   func changeLocal(num [5]int) {
6       num[0] = 55
7       fmt.Println("inside function ", num)
8
9   }
10  func main() {
11      num := [...]int{5, 6, 7, 8, 8}
12      fmt.Println("before passing to function ", num
13      changeLocal(num) //num is passed by value
14      fmt.Println("after passing to function ", num)
15  }
```

In the above program in line no. 13, the array `num` is actually passed by value to the function `changeLocal` and hence will not change because of the function call. This program will *output*,

```
before passing to function  [5 6 7 8 8]
inside function  [55 6 7 8 8]
after passing to function  [5 6 7 8 8]
```

Length of an array

The length of the array is found by passing the array as parameter to the `len` function.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       a := [...]float64{67.7, 89.8, 21, 78}
7       fmt.Println("length of a is",len(a))
8
9   }
```

The **output** of the above program is `length of a is 4`

Iterating arrays using range

The `for` loop can be used to iterate over elements of an array.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       a := [...]float64{67.7, 89.8, 21, 78}
7       for i := 0; i < len(a); i++ { //looping from 0
8           fmt.Printf("%d th element of a is %.2f\n",
9       }
10  }
```

The above program uses a `for` loop to iterate over the elements of the array starting from index `0` to `length of the array - 1`. This program works and will print,

```
0 th element of a is 67.70
1 th element of a is 89.80
2 th element of a is 21.00
3 th element of a is 78.00
```

Get the free Golang tools cheat sheet

Go provides a better and concise way to iterate over an array by using the **range** form of the `for` loop. `range` returns both the index and the value at that index. Let's rewrite the above code using range. We will also find the sum of all elements of the array.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       a := [...]float64{67.7, 89.8, 21, 78}
7       sum := float64(0)
8       for i, v := range a {//range returns both the i
9           fmt.Printf("%d the element of a is %.2f\n",
10          sum += v
11      }
12      fmt.Println("\nsum of all elements of a",sum)
13  }
```

line no. 8 `for i, v := range a` of the above program is the range form of the for loop. It will return both the index and the value at that index. We print the values and also calculate the sum of all elements of the array `a`. The **output** of the program is,

```
0 the element of a is 67.70
1 the element of a is 89.80
2 the element of a is 21.00
3 the element of a is 78.00

sum of all elements of a 256.5
```

In case you want only the value and want to ignore the index, you can do this by replacing the index with the `_` blank identifier.

```
for _, v := range a { //ignores index
}
```

The above for loop ignores the index. Similarly the value can also be ignored.

Multidimensional arrays

The arrays we created so far are all single dimension. Its possible to create multidimensional arrays.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func printarray(a [3][2]string) {
8       for _, v1 := range a {
9           for _, v2 := range v1 {
10              fmt.Printf("%s ", v2)
11          }
12          fmt.Printf("\n")
13      }
14  }
15
16  func main() {
17      a := [3][2]string{
18          {"lion", "tiger"},
19          {"cat", "dog"},
20          {"pigeon", "peacock"}, //this comma is nece
21      }
22      printarray(a)
23      var b [3][2]string
24      b[0][0] = "apple"
25      b[0][1] = "samsung"
26      b[1][0] = "microsoft"
27      b[1][1] = "google"
28      b[2][0] = "AT&T"
29      b[2][1] = "T-Mobile"
30      fmt.Printf("\n")
31      printarray(b)
32  }
```

In the above program in line no. 17, a two dimensional string array `a` has been declared using short hand syntax. The comma at the end of line no. 20 is necessary. This is because of the fact that the lexer automatically inserts semicolons according to simple rules. Please read https://golang.org/doc/effective_go.html#semicolons if you are interested to know more as to why this is needed.

Another 2d array `b` is declared in line no. 23 and strings are added to it one by one for each index. This is another way of initialising a 2d array.

The `printarray` function in line no. 7 uses two for range loops to print the contents of 2d arrays. The **output** of the above program is

```
lion tiger
cat dog
pigeon peacock

apple samsung
microsoft google
AT&T T-Mobile
```

Thats it for arrays. Although arrays seem to be flexible enough, they come with the restriction that they are of fixed length. Its not possible to increase the length of an array. This is were **slices** come into picture. In fact in Go, slices are more common than conventional arrays.

Slices

A slice is a convenient, flexible and powerful wrapper on top of an array. Slices do not own any data on their own. They are the just references to existing arrays.

Creating a slice

A slice with elements of type T is represented by `[]T`

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       a := [5]int{76, 77, 78, 79, 80}
9       var b []int = a[1:4] //creates a slice from a[
10      fmt.Println(b)
11  }
```

The syntax `a[start:end]` creates a slice from array `a` starting from index `start` to index `end - 1`. So in line no. 9 of the above program `a[1:4]` creates a slice representation of the array `a` starting from indexes 1 through 3. Hence the slice `b` has values `[77 78 79]`.

Lets look at another way to create a slice.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       c := []int{6, 7, 8} //creates and array and ret
9       fmt.Println(c)
10  }
```

In the above program in line no. 9, `c := []int{6, 7, 8}` creates an array with 3 integers and returns a slice reference which is stored in c.

modifying a slice

A slice does not own any data of its own. It is just a representation of the underlying array. Any modifications done to the slice will be reflected in the underlying array.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       darr := [...]int{57, 89, 90, 82, 100, 78, 67, 6
9       dslice := darr[2:5]
10      fmt.Println("array before",darr)
11      for i := range dslice {
12          dslice[i]++
13      }
14      fmt.Println("array after",darr)
15  }
```

In line number 9 of the above program, we create `dslice` from indexes 2, 3, 4 of the array. The for loop increments the value in these indexes by one. When we print the array after the for loop, we can see that the changes to the slice are reflected in the array. The output of the program is

```
array before [57 89 90 82 100 78 67 69 59]
array after [57 89 91 83 101 78 67 69 59]
```

When a number of slices share the same underlying array, the changes that each one makes will be reflected in the array.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       numa := [3]int{78, 79 ,80}
9       nums1 := numa[:] //creates a slice which conta
10      nums2 := numa[:]
11      fmt.Println("array before change 1",numa)
12      nums1[0] = 100
13      fmt.Println("array after modification to slice
14      nums2[1] = 101
15      fmt.Println("array after modification to slice
16  }
```

In line no. 9, in `numa[:]` the start and end values are missing. The default values for start and end are `0` and `len(numa)` respectively. Both slices `nums1` and `nums2` share the same array. The output of the program is

```
array before change 1 [78 79 80]
array after modification to slice nums1 [100 79 80]
array after modification to slice nums2 [100 101 80]
```

From the output its clear that when slices share the same array, the modifications which each one makes are reflected in the array.

length and capacity of a slice

The length of the slice is the number of elements in the slice. **The capacity of the slice is the number of elements in the underlying array starting from the index from which the slice is created.**

Lets write some code to understand this better.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       fruitarray := [...]string{"apple", "orange", "g
9       fruitslice := fruitarray[1:3]
10      fmt.Printf("length of slice %d capacity %d", le
11  }
```

In the above program, `fruitslice` is created from indexes 1 and 2 of the `fruitarray`. Hence the length of `fruitslice` is 2.

The length of the `fruitarray` is 7. `fruiteslice` is created from index `1` of `fruitarray`. Hence the capacity of `fruitslice` is the no of elements in `fruitarray` starting from index `1` i.e from `orange` and that value is `6`. Hence the capacity of fruitslice is 6. The program outputs **length of slice 2 capacity 6**.

A slice can be re-sliced upto its capacity. Anything beyond that will cause the program to throw a run time error.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       fruitarray := [...]string{"apple", "orange", "g
9       fruitslice := fruitarray[1:3]
10      fmt.Printf("length of slice %d capacity %d\n",
11      fruitslice = fruitslice[:cap(fruitslice)] //re-
12      fmt.Println("After re-slicing length is",len(fr
13  }
```

In line no. 11 of the above program, `fruitslice` is re-sliced to its capacity. The above program outputs,

```
length of slice 2 capacity 6
After re-slicing length is 6 and capacity is 6
```

Get the free Golang tools cheat sheet

creating a slice using make

func make([]T, len, cap) []T can be used to create a slice by passing the type, length and capacity. The capacity parameter is optional and defaults to the length. The make function creates an array and returns a slice reference to it.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       i := make([]int, 5, 5)
9       fmt.Println(i)
10  }
```

The values are zeroed by default when a slice is created using make. The above program will output `[0 0 0 0 0]`.

Appending to a slice

As we already know arrays are restricted to fixed length and their length cannot be increased. Slices are dynamic and new elements can be appended to the slice using `append` function. The definition of append function is `func append(s []T, x ...T) []T`.

x ...T in the function definition means that the function accepts variable number of arguments for the parameter x. These type of functions are called variadic functions.

One question might be bothering you though. If slices are backed by arrays and arrays themselves are of fixed length then how come a slice is of dynamic length. Well what happens under the hoods is, when new elements are appended to the slice, a new array is created. The elements of the existing array are copied to this new array and a new slice reference for this new array is returned. The capacity of the new slice is now twice that

[redacted]

changes made inside the function are visible outside the function too. Lets write a program to check this out.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func subtactOne(numbers []int) {
8       for i := range numbers {
9           numbers[i] -= 2
10      }
11
12  }
13  func main() {
14
15      nos := []int{8, 7, 6}
16      fmt.Println("slice before function call", nos)
17      subtactOne(nos)
18      fmt.Println("slice after function call", nos)
19
20  }
```

The function call in line number 17 of the above program decrements each element of the slice by 2. When the slice is printed after the function call, these changes are visible. If you can recall, this is different from an array where the changes made to an array inside a function are not visible outside the function. Output of the above program is,

```
array before function call [8 7 6]
array after function call [6 5 4]
```

Multidimensional slices

Similar to arrays, slices can have multiple dimensions.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7
8   func main() {
9        pls := [][]string {
10           {"C", "C++"},
11           {"JavaScript"},
12           {"Go", "Rust"},
13           }
14      for _, v1 := range pls {
15          for _, v2 := range v1 {
16              fmt.Printf("%s ", v2)
17          }
18          fmt.Printf("\n")
19      }
20  }
```

The output of the program is,

```
C C++
JavaScript
Go Rust
```

Memory Optimisation

Slices hold a reference to the underlying array. As long as the slice is in memory, the array cannot be garbage collected. This might be of concern when it comes to memory management. Lets assume that we have a very large array and we are interested in processing only a small part of it. Henceforth we create a slice from that array and start processing the slice. The important thing to be noted here is that the array will still be in memory since the slice references it.

One way to solve this problem is to use the copy function `func copy(dst, src []T) int` to make a copy of that slice. This way we can use the new slice and the original array can be garbage collected.

```
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func countries() []string {
8       countries := []string{"USA", "Singapore", "Gerr
9       neededCountries := countries[:len(countries)-2
10      countriesCpy := make([]string, len(neededCountr
11      copy(countriesCpy, neededCountries) //copies ne
12      return countriesCpy
13  }
14  func main() {
15      countriesNeeded := countries()
16      fmt.Println(countriesNeeded)
17  }
```

In line no. 9 of the above program, `neededCountries := countries[:len(countries)-2]` creates a slice of `countries` barring the last 2 elements. Line no. 11 of the above program copies `neededCountries` to `countriesCpy` and also returns it from the function in the next line. Now `countries` array can be garbage collected since `neededCountries` is no longer referenced.

I have compiled all the concepts we discussed so far into a single program. You can download it from github.

Thats it for arrays and slices. Thanks for reading. Please leave your valuable feedback and comments.

Get the free Golang tools cheat sheet

Next tutorial - Variadic Functions

Naveen Ramanathan

iOS developer at dietco.de and Golang enthusiast.

Share this post

About

For any queries/suggestions, please contact us at **naveen[at]golangbot[dot]com**

Follow Us

Newsletter

Join Our Newsletter

Signup for our newsletter and get the **Golang tools cheat sheet for free**.

Email*

Subscribe

Golang tutorial - Go programming tutorial from golangbot © 2017

Proudly published with Ghost