Capacity and length of a slice in Go

October 5, 2021

introduction slice array

Share:

In Go, the length of a slice tells you how many elements it contains. It can be obtained using the `len()` function. The capacity is the size of the slice's underlying array and can be obtained with the `cap()` function.

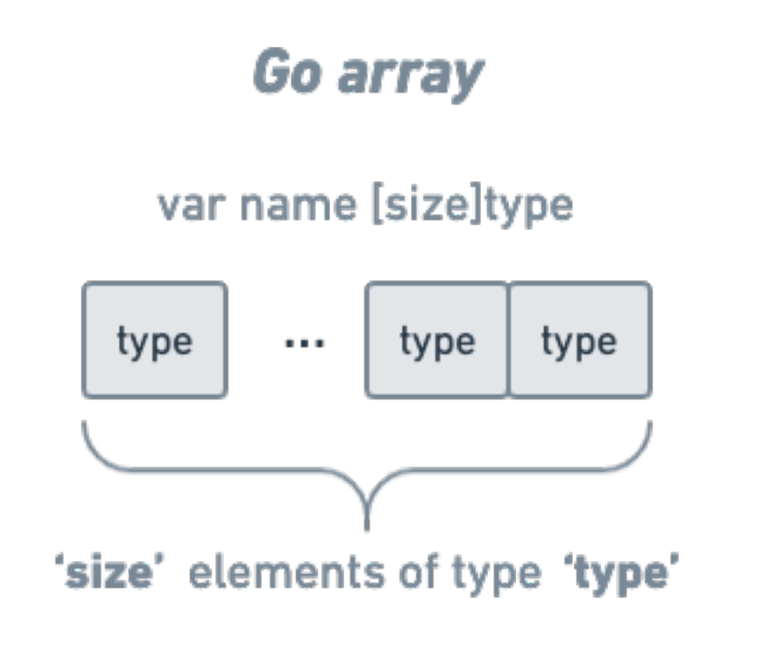
Difference between arrays and slices

To better understand the difference between the capacity and length of a slice, first, you should know the differences between arrays and slices.

Arrays

An array is an indexed collection of a certain `size` with values of the same `type`, declared as:

`var name [size]type`



Initializing an array

```
var a [4]int                    // array with zero values
var b [4]int = [4]int{0, 1, 2}    // partially initialized array
var c [4]int = [4]int{1, 2, 3, 4} // array initialization
d := [...]int{5, 6, 7, 0}         // ... - means that array size equals the numb

fmt.Printf("a: length: %d, capacity: %d, data: %v\n", len(a), cap(a), a)
fmt.Printf("b: length: %d, capacity: %d, data: %v\n", len(b), cap(b), b)
fmt.Printf("c: length: %d, capacity: %d, data: %v\n", len(c), cap(c), c)
fmt.Printf("d: length: %d, capacity: %d, data: %v\n", len(d), cap(d), d)
```

Output:

```
a: length: 4, capacity: 4, data: [0 0 0 0]
b: length: 4, capacity: 4, data: [0 1 2 0]
c: length: 4, capacity: 4, data: [1 2 3 4]
d: length: 4, capacity: 4, data: [5 6 7 0]
```

Properties of arrays

- Arrays have a fixed size and cannot be resized. Slices can be resized.
- The type of the array includes its size. The `[4]int` array type is distinct from `[5]int`, and they cannot be compared.
- Initializing an array with `var name [size]type` creates a collection of `size` elements of type `type` and each of them is the zero value for the given `type`.
- Arrays are **passed by value**. It means that when you assign one array to another, you will make a new copy of its contents:

```
var a [4]int = [4]int{1, 2, 3, 4}
b := a
a[1] = 999
fmt.Println(a)
fmt.Println(b)
```

Output:

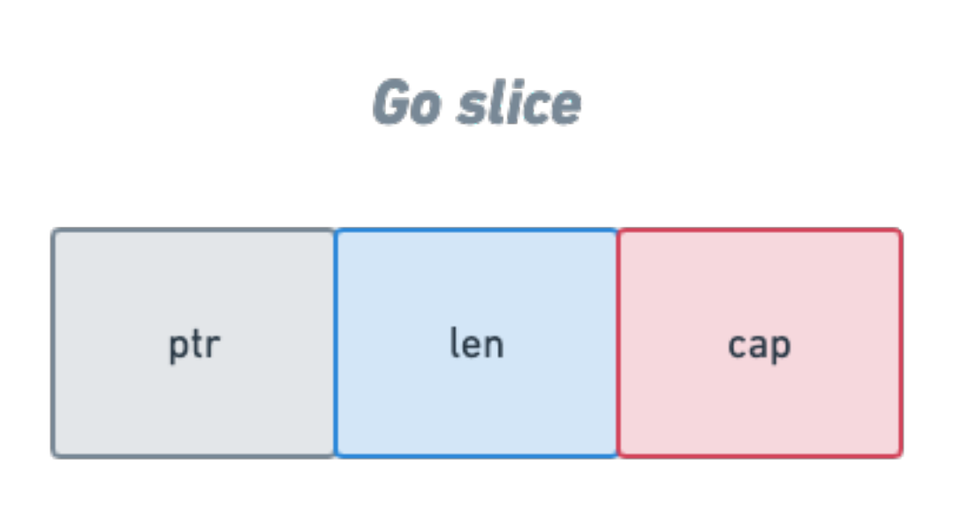
```
[1 999 3 4]
[1 2 3 4]
```

Slices

A slice declared as:

`var name []type`

is a data structure describing a piece of an array with three properties:



- `ptr` - a pointer to the underlying array
- `len` - length of the slice - number of elements in the slice
- `cap` - capacity of the slice - length of the underlying array, which is also the maximum length the slice can take (until it grows)

A slice **is not an array**. It describes a section of the underlying array stored under the `ptr` pointer.

Initializing a slice

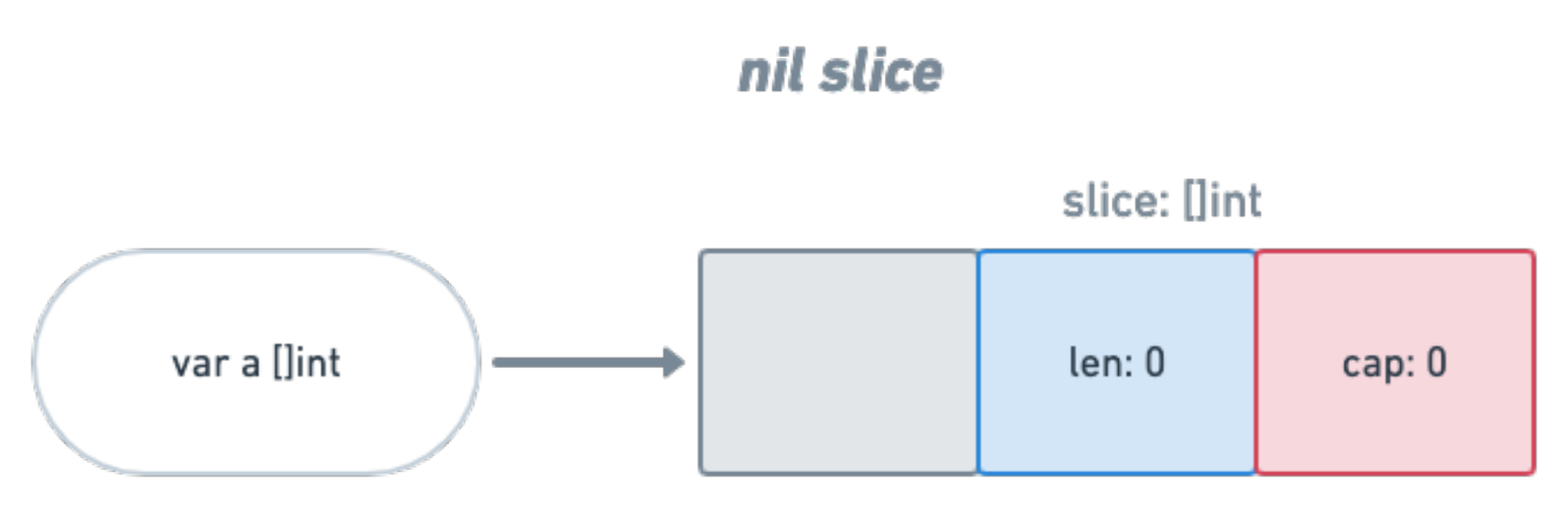
```
var a []int             // nil slice
b := []int{0, 1, 2, 3} // slice initialized with specified array
c := make([]int, 4)    // slice of size 4 initialized with zero-valued array of
d := make([]int, 4, 5) // slice of size 4 initialized with zero-valued array of

fmt.Printf("a: length: %d, capacity: %d, pointer to underlying array: %p, data:
fmt.Printf("b: length: %d, capacity: %d, pointer to underlying array: %p, data:
fmt.Printf("c: length: %d, capacity: %d, pointer to underlying array: %p, data:
fmt.Printf("d: length: %d, capacity: %d, pointer to underlying array: %p, data:
```

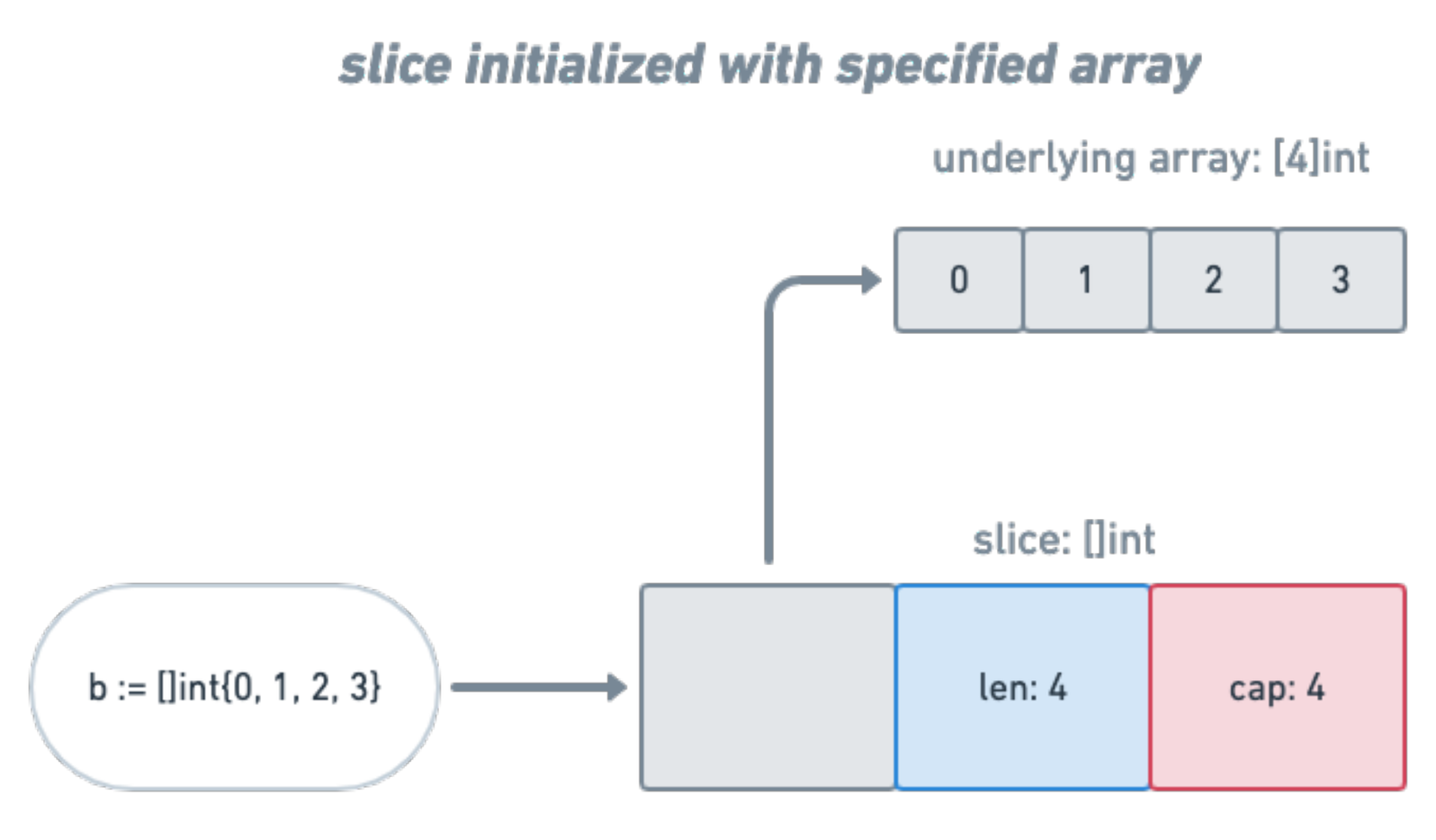
Output:

```
a: length: 0, capacity: 0, pointer to underlying array: 0x0, data: [], is nil: t
b: length: 4, capacity: 4, pointer to underlying array: 0xc00001e060, data: [0 1
c: length: 4, capacity: 4, pointer to underlying array: 0xc00001e080, data: [0 0
d: length: 4, capacity: 5, pointer to underlying array: 0xc000016180, data: [0 0
```

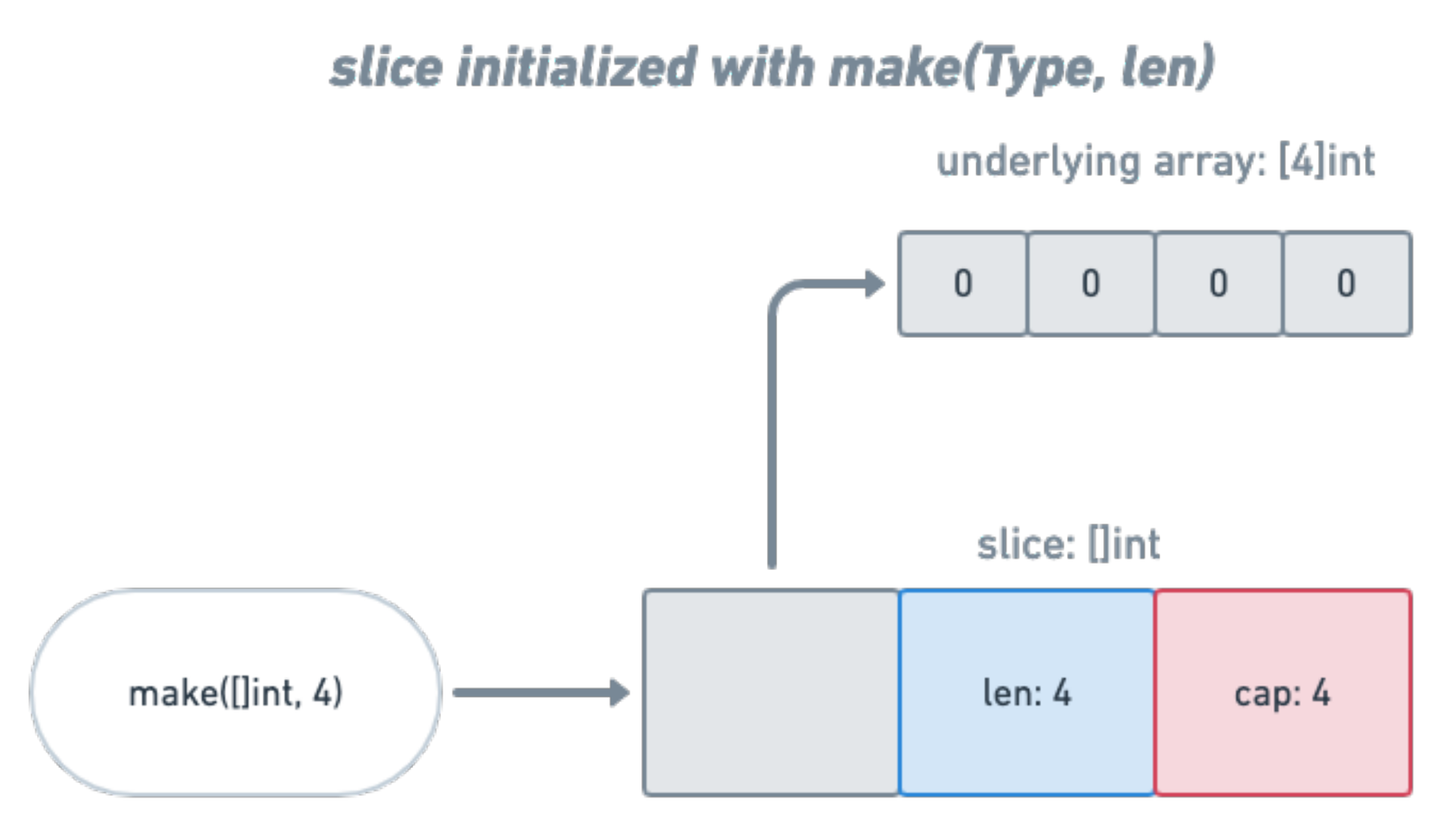
As we see in the output, `var a []int` creates a `nil` slice - a slice that has the length and capacity equal to 0, and no underlying array.



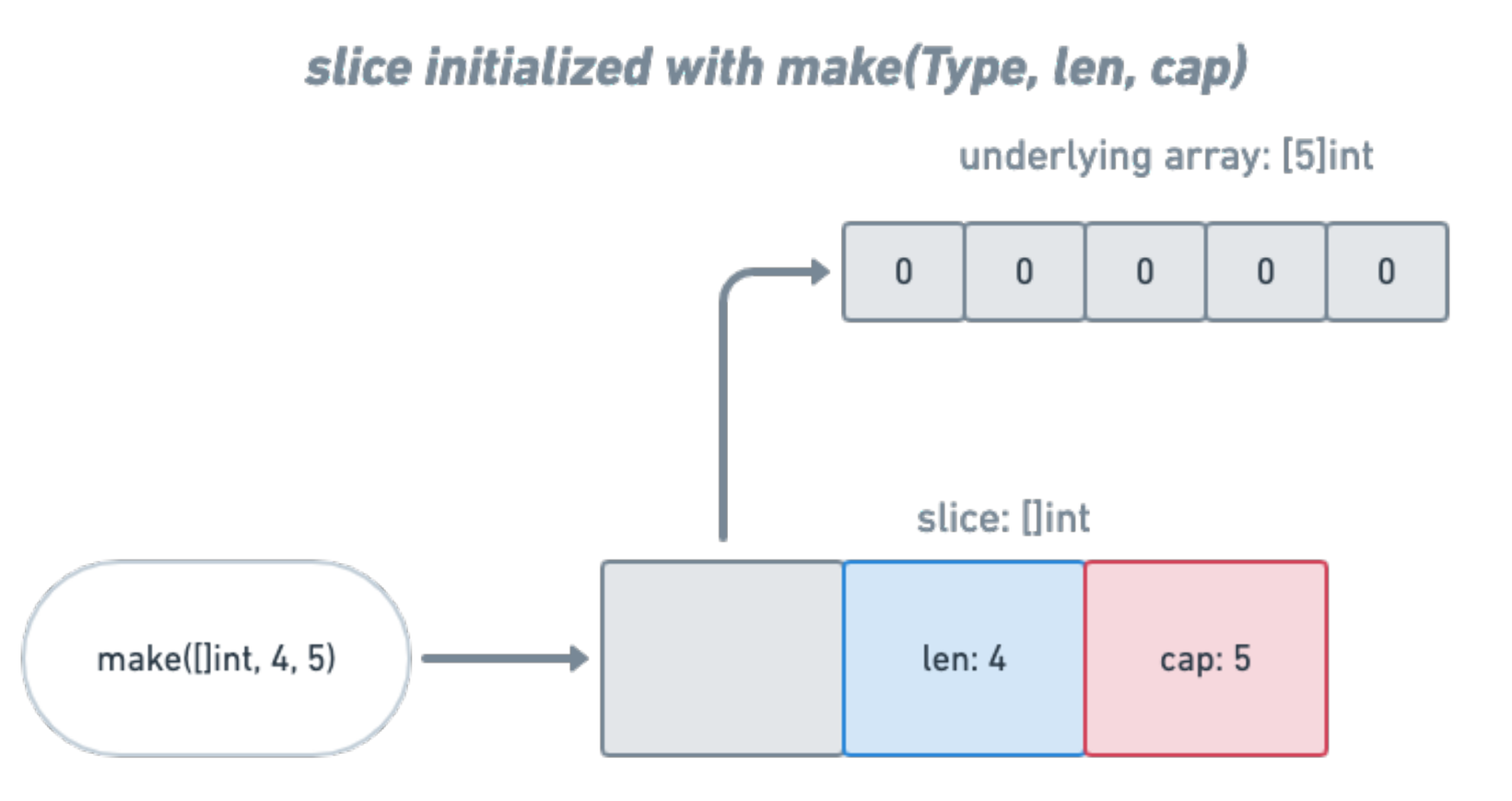
Initializing a slice with the specified array, i.e., `b := []int{0, 1, 2, 3}`, creates a new slice with capacity and length taken from the underlying array.



A slice can also be initialized with the built-in `make()` function that takes the type of a slice as the first argument and the length as the second. The resulting slice has a capacity equals to the length, and the underlying array is initialized with zero values.



There is also an alternative version of the `make()` function with three arguments: the first is the type of a slice, the second is the length, and the third is the capacity. In this way, you can create a slice with a capacity greater than the length.



Properties of slices

- A slice is automatically resized when the `append()` function is called. Resizing here means that the `append()` adds new elements to the end of the slice, and if there is not sufficient capacity in the underlying array, a new array will be allocated. The `append()` function always returns a new, updated slice, so if you want to resize a slice `s` it is necessary to store the result in the same variable `s`.
- Slices are not comparable and simple equality comparison `a == b` is not possible. See how to compare slices.
- Initializing a slice with `var name []type` creates a `nil` slice that has length and capacity equal to 0 and no underlying array. See what is the difference between nil and empty slices.
- Just like arrays (and everything in Go), slices are **passed by value**. When you assign a slice to a new variable, the `ptr`, `len`, and `cap` are copied, including the `ptr` pointer that will **point to the same underlying array**. If you modify the copied slice, you modify the same shared array which makes all changes visible in the old and new slices:

```
var a []int = []int{1, 2, 3, 4}
b := a
a[1] = 999
fmt.Println(a)
fmt.Println(b)
```

Output:

```
[1 999 3 4]
[1 999 3 4]
```

Length and capacity

You already know that capacity is the size of the slice's underlying array and length is the number of the slice elements, but what is the relationship between them? To understand this better, let's analyze the re-slicing and appending operations.

Re-slicing

Re-slicing is an operation that creates a new slice from an existing one or an array. To "slice" an array or "re-slice" an existing slice, use a half-open range with two indices separated by a colon:

```
var arr [4]int = [4]int{1, 2, 3, 4}
a := arr[1:3]
fmt.Printf("a: length: %d, capacity: %d, data: %v\n", len(a), cap(a), a)
```

Output:

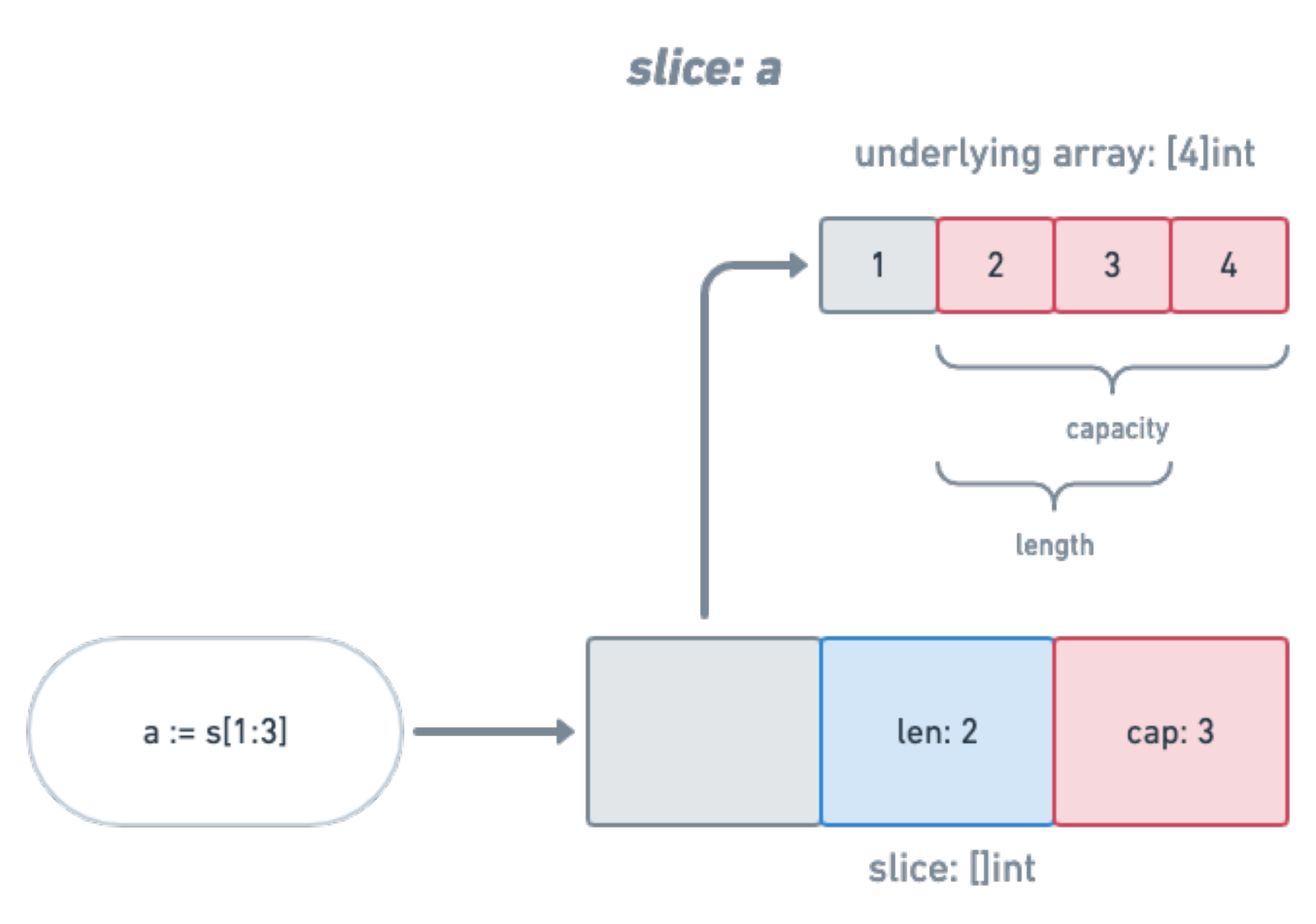
```
a: length: 2, capacity: 3, data: [2 3]
```

We get the same results for the slice:

```
var s []int = []int{1, 2, 3, 4}
a := s[1:3]
fmt.Printf("a: length: %d, capacity: %d, data: %v\n", len(a), cap(a), a)
```

Output:

```
a: length: 2, capacity: 3, data: [2 3]
```

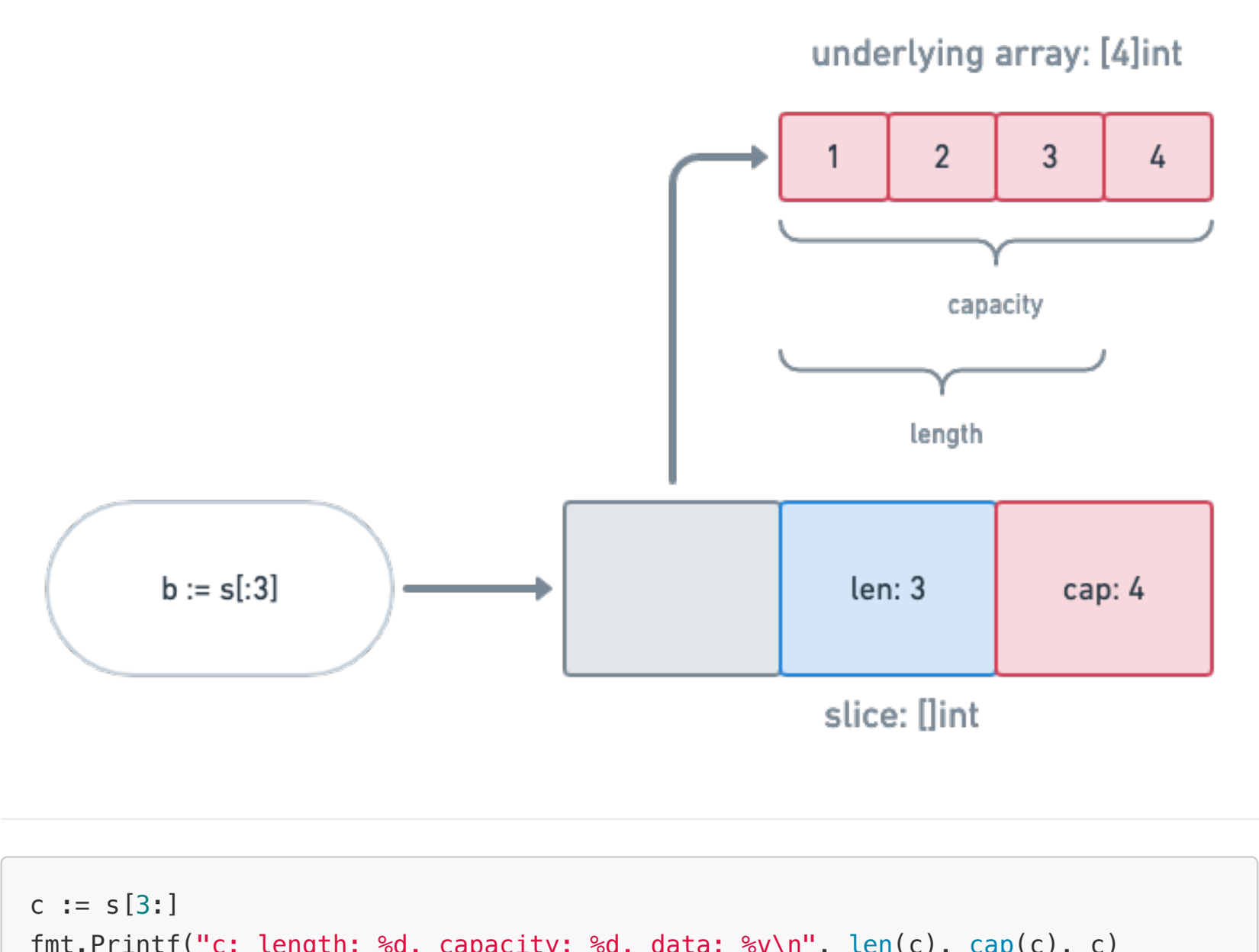


Re-slicing a slice or an array creates a new slice with length given by indices range and capacity equal to the number of elements in the underlying array from the index of the first element of the slice to the end of the array. See two more examples of re-slicing operation - for range without the first index `s[:3]`, and without the last index `s[3:]`:

```
b := s[:3]
fmt.Printf("b: length: %d, capacity: %d, data: %v\n", len(b), cap(b), b)
```

Output:

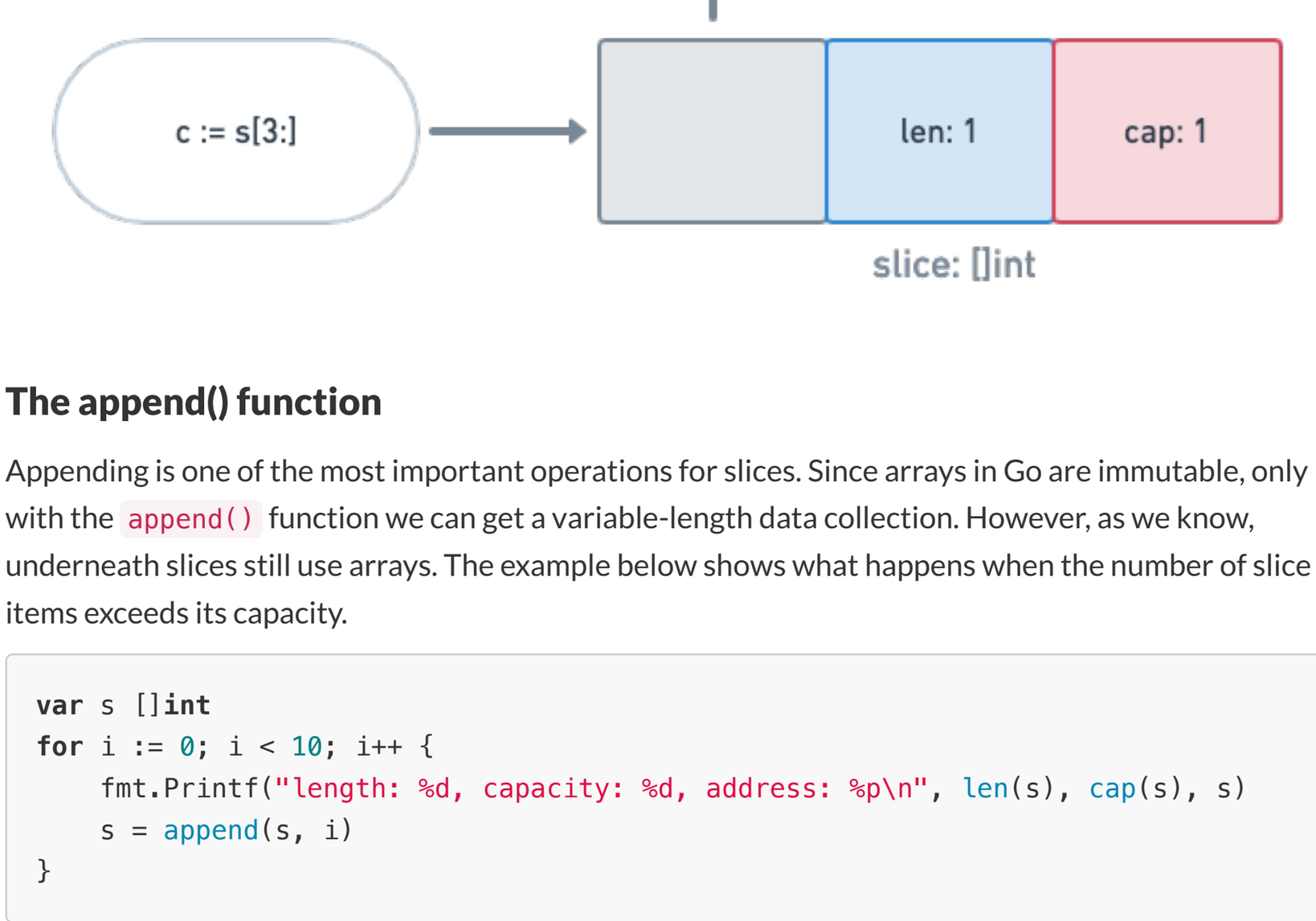
```
b: length: 3, capacity: 4, data: [1 2 3]
```




```
c := s[3:]
fmt.Printf("c: length: %d, capacity: %d, data: %v\n", len(c), cap(c), c)
```

Output:

```
c: length: 1, capacity: 1, data: [4]
```



The append() function

Appending is one of the most important operations for slices. Since arrays in Go are immutable, only with the `append()` function we can get a variable-length data collection. However, as we know, underneath slices still use arrays. The example below shows what happens when the number of slice items exceeds its capacity.

```
var s []int
for i := 0; i < 10; i++ {
    fmt.Printf("length: %d, capacity: %d, address: %p\n", len(s), cap(s), s)
    s = append(s, i)
}
```

Output:

```
length: 0, capacity: 0, address: 0x0
length: 1, capacity: 1, address: 0xc00001c0a0
length: 2, capacity: 2, address: 0xc00001c0b0
length: 3, capacity: 4, address: 0xc00001e080
length: 4, capacity: 4, address: 0xc00001e080
length: 5, capacity: 8, address: 0xc000020140
length: 6, capacity: 8, address: 0xc000020140
length: 7, capacity: 8, address: 0xc000020140
length: 8, capacity: 8, address: 0xc000020140
length: 9, capacity: 16, address: 0xc000026200
```

As you can see in the output, every time the length of the slice is beyond its capacity (the length of the underlying array), the `append()` function expands the slice by allocating a new underlying array of twice its size and copying all of its elements there. Notice that the pointer to the underlying array changes with each change in capacity.

Conclusion

To understand the length and capacity of slices in Go, it is important to understand how slices work and what is the difference between slices and arrays. Slices are built on top of arrays to provide variable-length data collections. They consist of three elements - a pointer to the underlying array (underneath, slices use arrays as data storage), the length of the slice, and the capacity - the size of the underlying array. These 3 properties are copied when a slice value is passed, but the new pointer always points to the same shared array. The `append()` function makes slices expandable, creating a powerful and expressive data structure, one of the most used in Go.

Related

Copy a slice in Go
Learn how to make a deep copy of a slice
introduction slice
November 25, 2021

Check if the slice contains the given value in Go
Learn how to write a function that checks if a slice has a specific value
introduction slice
November 5, 2021

Convert string to []byte or []byte to string in Go
Learn the difference between a string and a byte slice
introduction strings slice
October 13, 2021