链滴

Golang 入门笔记 -05- 数组和切片

作者：zyk

原文链接：https://ld246.com/article/1601268254132

来源网站：链滴

数组

初始化

数组可以保存**指定长度**的多个数据，且这些数据的类型都**相同**，数据类型可以是原始类型，如整型和符串等，也可以是自定义类型。

数组通过**索引**来访问元素，索引从 0 开始，第一个元素的索引为 0，第二个为 1，依此类推。

在 Go 语言中声明数组的格式为：

```
var variable [len]type
```

例如，声明名称为arr1，长度和类型分别为5和 int 的数组：

```
var arr1 [5]int
```

我们可以让编译器根据元素个数**自动推断**数组长度，只需要在声明长度时用 ... 替代：

```
var numArray = [...]int{1, 2, 3}
```

我们还可以根据**索引**来声明数组：

```
a := [...]string{0: "北京", 1: "上海"} // 索引 0 对应的元素为"北京"，1 对应的元素为"上海"
```

整型数组中所有元素都初始化为 0，数组 arr中第 i 个元素为arr[i - 1]，最后一个元素为 arr[len(arr) -]。

数组是**可变**的，可以通过索引对元素进行赋值：arr[1] = 1。

注意：在程序中若索引超出数组**最大有效索引**，会引发 index out of range 错误。

遍历数组

- 普通 for 循环

```
package main

import "fmt"

func main() {
    a := [...]int{2, 4, 6, 8, 10}

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

- for-range 循环

```
package main

import "fmt"

func main() {
    a := [...]int{2, 4, 6, 8, 10}

    for k, v := range a {
        fmt.Println(k, v)
    }
}
```

切片

概念

切片 slice 是对数组的引用，因此切片是一个**引用类型**（类似于python 中的 list）。

切片是一个**长度可变的数组。**

可以通过 cap() 函数来获取切片的**容量**，而 len() 函数获取的是切片的**长度**（切片保存的元素个数）
对于切片 s ，存在这样的数量关系：

```
0 <= len(s) <= cap(s)
```

声明切片：

```
var variable []type
```

可以通过类似数组的声明方式来声明切片：

```
var s = []int{1, 2, 3}
```

若 arr 是数组，可以通过**切割数组**来声明切片 s，如：

```
var arr = [...]int{1, 2, 3, 4, 5}
```

```
var s []int = arr[0:3] // 声明切片 s，s 由 arr 中索引 0 到 2 的元素构成
var s1 []int = arr[:] // s1 由 arr 所有元素构成
var s2 []int = arr[2:] // s2 由 arr 中索引 2 开始到最后的元素构成
var s3 []int = arr[:3] // s3 由 arr 中索引 0 ~ 2 的元素构成
```

用 make() 创建切片

我们可以通过 make() 来创建一个切片：

```
var s []type = make([]type, len)
```

其中，type是类型，len 是切片的长度。

我们来演示下切片的内存结构：

```
package main
import "fmt"

func main() {
   var slice1 []int = make([]int, 10)

   for i := 0; i < len(slice1); i++ {
      slice1[i] = 5 * i
   }

   for i := 0; i < len(slice1); i++ {
      fmt.Printf("Slice at %d is %d\n", i, slice1[i])
   }
   fmt.Printf("\nThe length of slice1 is %d\n", len(slice1))
   fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
}
```

运行结果为：

```
Slice at 0 is 0
Slice at 1 is 5
Slice at 2 is 10
Slice at 3 is 15
Slice at 4 is 20
Slice at 5 is 25
Slice at 6 is 30
Slice at 7 is 35
Slice at 8 is 40
Slice at 9 is 45
The length of slice1 is 10
The capacity of slice1 is 10
```

我们可以在初始化切片时候，指定切片**初始长度**和**切片容量**：

```
slice1 := make([]type, length, cap) // type 为类型，length 为初始长度, cap 为切片容量
```

切片重组

切片会**自动扩容**，我们也可以手动进行扩容，比如将切片 s1 扩展 1 位：

```
sl = sl[0:len(sl)+1]
```

切片可以反复扩容直至切片长度到达切片容量：

```
package main
import "fmt"

func main() {
   slice1 := make([]int, 0, 10)
   // load the slice, cap(slice1) is 10:
   for i := 0; i < cap(slice1); i++ {
      slice1 = slice1[0:i+1]
      slice1[i] = i
      fmt.Printf("The length of slice is %d\n", len(slice1))
   }

   // print the slice:
   for i := 0; i < len(slice1); i++ {
      fmt.Printf("Slice at %d is %d\n", i, slice1[i])
   }
}
```

上述代码运行结果为：

```
The length of slice is 1
The length of slice is 2
The length of slice is 3
The length of slice is 4
The length of slice is 5
The length of slice is 6
The length of slice is 7
The length of slice is 8
The length of slice is 9
The length of slice is 10
Slice at 0 is 0
Slice at 1 is 1
Slice at 2 is 2
Slice at 3 is 3
Slice at 4 is 4
Slice at 5 is 5
Slice at 6 is 6
Slice at 7 is 7
Slice at 8 is 8
Slice at 9 is 9
```

切片的复制与追加

我们可以通过 copy 函数和 append 函数实现切片元素的**复制**和**追加**，如：

```
package main
import "fmt"
```

```
func main() {
    sl_from := []int{1, 2, 3}
    sl_to := make([]int, 10)

    n := copy(sl_to, sl_from)
    fmt.Println(sl_to)
    fmt.Printf("Copied %d elements\n", n) // n == 3

    sl3 := []int{1, 2, 3}
    sl3 = append(sl3, 4, 5, 6)
    fmt.Println(sl3) // 1, 2, 3, 4, 5, 6
}
```

append 函数可以将多个**相同类型**的元素追加到切片后面，同时返回新的切片。若切片的容量不够，append 会分配新的切片保证能存储原切片和新元素。

如果想将切片 y 追加到 x 后面，只需将在 y后面添加 .. 将其扩展成列表即可，如：

```
x = append(x, y...)
```

注意：append 虽然很好用，但是如果想要理解切片追加元素的原理，可以自己来实现一个 AppendByte 方法：

```
func AppendByte(slice []byte, data ...byte) []byte {
    m := len(slice) // 原切片长度
    n := m + len(data) // 新切片长度
    if n > cap(slice) { // 若新切片长度大于切片容量，手动扩容
        newSlice := make([]byte, (n+1)*2)
        copy(newSlice, slice)
        slice = newSlice
    }
    // 复制元素
    slice = slice[0:n]
    copy(slice[m:n], data)
    return slice
}
```

字符串、数组和切片的应用

从字符串生成字节切片

若 s 为一个字符串，可以通过 c := []bytes(s) 来获取 s 对应的字节切片。也可以通过 copy 函数来实：copy (b []byte, s string)。

截取字符串

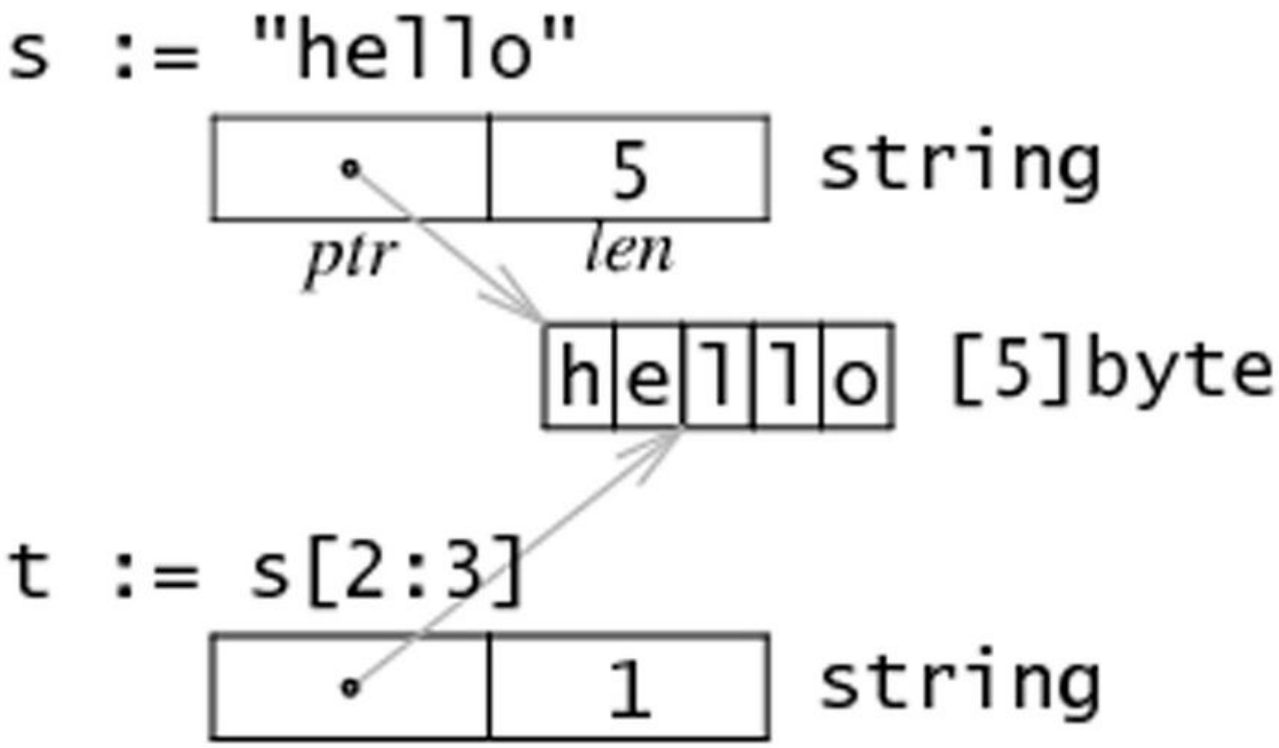
substr := str[start:end] 可以从字符串 str 获取到从索引 start 开始到 end-1 位置的子串；

str[start:] 则表示获取从 start 开始到 len(str)-1 位置的子串；

str[:end] 表示获取从 0 开始到 end-1的子串。

字符串和切片的内存结构

字符串是一个双字结构，即一个指向实际数据的**指针**和记录字符串长度的**整数**，如下图所示：



修改字符串的字符

Go 语言的字符串是**不可变**的，对于 str[index]，我们不能执行这样的语句：

```
str[index] = 'A'
```

编译器会报 cannot assign to str[i] 错误。

因此，如果要修改字符串中的字符，需要先将**字符串**转换成**字节数组**，然后修改**字节数组**中的元素来现修改字符串字符的目的，最后再将**字节数组**转换成**字符串**。例如：

```
s := "hello"
c := []byte(s)
c[0] = 'm'
s2 := string(c) // "mello"
```

append 函数

1. 切片 b 的元素追加到切片 a 之后：a = append(a, b...)
2. 复制切片 a 的元素到新的切片 b 上：

```
b = make([]T, len(a))
copy(b, a)
```

3. 删除位于索引 i 的元素：a = append(a[:i], a[i+1:]...)

4. 切除切片 a 中从索引 i 至 j 位置的元素：a = append(a[:i], a[j:]...)
5. 为切片 a 扩展 j 个元素长度：a = append(a, make([]T, j)...)
6. 在索引 i 的位置插入元素 x：a = append(a[:i], append([]T{x}, a[i:]...)...)
7. 在索引 i 的位置插入长度为 j 的新切片：a = append(a[:i], append(make([]T, j), a[i:]...)...)
8. 在索引 i 的位置插入切片 b 的所有元素：a = append(a[:i], append(b, a[i:]...)...)
9. 取出位于切片 a 最末尾的元素 x：x, a = a[len(a)-1], a[:len(a)-1]
10. 将元素 x 追加到切片 a：a = append(a, x)

切片和垃圾回收

切片的底层指向一个数组，该数组的实际容量可能大于切片的长度。只有没有任何切片指向该数组时底层数组才会被释放，这有可能会导致占用多余内存。

示例：

函数 FindDigits 将一个文件加载到内存，然后搜索其中所有的数字并返回一个切片：

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

上述代码中返回了 []byte，它指向底层整个文件的数据。若该切片不被释放，垃圾回收器就不能释放个文件所占用的内存。为了避免这个问题，可以复制我们需要的部分到一个新切片：

```
func FindDigits(filename string) []byte {
  b, _ := ioutil.ReadFile(filename)
  b = digitRegexp.Find(b)
  c := make([]byte, len(b))
  copy(c, b)
  return c
}
```