Go 语言数组

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整型、字符串或者自定义类型。

相对于去声明 **number0, number1, ..., number99** 的变量，使用数组形式 **numbers[0], numbers[1] ..., numbers[99]** 更加方便且易于扩展。

数组长度必须是一个常量表达式，并且必须是一个非负整数。数组长度也是数组类型的一部分，所以[5]int和[10]int是属于不同类型的。数组的编译时值初始化是按照数组顺序完成的

数组元素可以通过索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。

元素的数目，也称为长度或者数组大小必须是固定的并且在声明该数组时就给出（编译时需要知道数组长度以便分配内存）；数组长度最大为 2Gb。

数组可以通过下标进行访问，下标是从0开始，最后一个元素下标是：len-1。

访问越界，如果下标在数组合法范围之外，则触发访问越界，会panic。

数组是值类型，赋值和传参会复制整个数组，而不是指针。因此改变副本的值，不会改变本身的值。

支持 "=="、"!=" 操作符，因为内存总是被初始化过的。

指针数组 [n]*T，数组指针 *[n]T。

声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

```
var variable_name [SIZE] variable_type
```

`variable_name`：数组名称

`SIZE`：数组长度，必须是常量

`variable_type`：数组保存元素的类型

以上为一维数组的定义方式。例如以下定义了数组 balance 长度为 10 类型为 float32：

```
var balance [10] float32
```

初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

我们也可以通过字面量在声明数组的同时快速初始化数组：

```
balance := [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

如果数组长度不确定，可以使用 **...** 代替数组的长度，编译器会根据元素个数自行推断数组的长度：

```
var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
或
balance := [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

如果设置了数组的长度，我们还可以通过指定下标来初始化元素：

```
//  将索引为 1 和 3 的元素初始化
balance := [5]float32{1:2.0,3:7.0}
```

初始化数组中 **{}** 中的元素个数不能大于 **[]** 中的数字。

如果忽略 **[]** 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
 balance[4] = 50.0
```

以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。

访问数组元素

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

```
var salary float32 = balance[9]
```

索引范围从 0 到 `len(salary)-1`。

第一个元素是 `salary[0]`，第三个元素是 `salary[2]`；总体来说索引 i 代表的元素是 `salary[i]`，最后一个元素是 `salary[len(salary)-1]`。

对索引项为 i 的数组元素赋值可以这么操作：`salary[i] = value`，所以数组是 **可变的。**

只有有效的索引可以被使用，当使用等于或者大于 `len(salary)` 的索引时：如果编译器可以检测到，会给出索引超限的提示信息；如果检测不到的话编译会通过而运行时会 panic。

以下演示了数组完整操作（声明、赋值、访问）的实例：

```
package main

import "fmt"

func main() {
   var n [10]int /* n 是一个长度为 10 的数组 */
   var i,j int

   /* 为数组 n 初始化元素 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* 设置元素为 i + 100 */
   }

   /* 输出每个数组元素的值 */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
```

```
   }
}
#结果
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

```
package main

import "fmt"

func main() {
   var i,j,k int
   // 声明数组的同时快速初始化数组
   balance := [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}

   /* 输出数组元素 */          ...
   for i = 0; i < 5; i++ {
      fmt.Printf("balance[%d] = %f\n", i, balance[i] )
   }

   balance2 := [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
   /* 输出每个数组元素的值 */
   for j = 0; j < 5; j++ {
      fmt.Printf("balance2[%d] = %f\n", j, balance2[j] )
   }

   //  将索引为 1 和 3 的元素初始化
   balance3 := [5]float32{1:2.0,3:7.0}
   for k = 0; k < 5; k++ {
      fmt.Printf("balance3[%d] = %f\n", k, balance3[k] )
   }
}
#结果
balance[0] = 1000.000000
balance[1] = 2.000000
balance[2] = 3.400000
balance[3] = 7.000000
balance[4] = 50.000000
balance2[0] = 1000.000000
balance2[1] = 2.000000
balance2[2] = 3.400000
balance2[3] = 7.000000
balance2[4] = 50.000000
balance3[0] = 0.000000
balance3[1] = 2.000000
balance3[2] = 0.000000
balance3[3] = 7.000000
```

```
balance3[4] = 0.000000
```

由于索引的存在，遍历数组的方法自然就是使用 for 结构:

- 通过 for 初始化数组项
- 通过 for 打印数组元素
- 通过 for 依次处理元素

for 循环中的条件非常重要：`i < len(arr1)`，如果写成 `i <= len(arr1)` 的话会产生越界错误。

```
for i:=0; i < len(arr1); i++ {
    arr1[i] = ...
}
```

也可以使用 for-range 的生成方式：

```
for i,_:= range arr1 {
...
}
```

在这里i也是数组的索引。

Go 语言多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

以下实例声明了三维的整型数组：

```
var threedim [5][10][4]int
```

二维数组

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

```
var arrayName [ x ][ y ] variable_type
```

variable_type 为 Go 语言的数据类型，arrayName 为数组名，二维数组可认为是一个表格，x 为行，y 为列。

二维数组中的元素可通过`a[i][j]`来访问。

示例：

```
package main

import "fmt"

func main() {
    // Step 1: 创建数组
    values := [][]int{}
```

```
    // Step 2: 使用 append() 函数向空的二维数组添加两行一维数组
    row1 := []int{1, 2, 3}
    row2 := []int{4, 5, 6}
    values = append(values, row1)
    values = append(values, row2)

    // Step 3: 显示两行数据
    fmt.Println("Row 1")
    fmt.Println(values[0])
    fmt.Println("Row 2")
    fmt.Println(values[1])

    // Step 4: 访问第一个元素
    fmt.Println("第一个元素为：")
    fmt.Println(values[0][0])
}
#结果
Row 1
[1 2 3]
Row 2
[4 5 6]
第一个元素为：
1
```

初始化二维数组

多维数组可通过大括号来初始值。以下实例为一个 3 行 4 列的二维数组：

```
a := [3][4]int{
 {0, 1, 2, 3} ,   /*  第一行索引为 0 */
 {4, 5, 6, 7} ,   /*  第二行索引为 1 */
 {8, 9, 10, 11},   /* 第三行索引为 2 */
}
```

> **注意：**以上代码中倒数第二行的 **}** 必须要有逗号，因为最后一行的 **}** 不能单独一行。也可以写成这样：
>
> ```
> a := [3][4]int{
> {0, 1, 2, 3} ,   /*  第一行索引为 0 */
> {4, 5, 6, 7} ,   /*  第二行索引为 1 */
> {8, 9, 10, 11}}   /* 第三行索引为 2 */
> ```

以下实例初始化一个 2 行 2 列 的二维数组：

```
package main

import "fmt"

func main() {
    // 创建二维数组
    sites := [2][2]string{}

    // 向二维数组添加元素
```

```
    sites[0][0] = "张三"
    sites[0][1] = "李四"
    sites[1][0] = "王五"
    sites[1][1] = "包子"

    // 显示结果
    fmt.Println(sites)
}
#结果
[[张三 李四] [王五 包子]]
```

访问二维数组

二维数组通过指定坐标来访问。如数组中的行索引与列索引，例如：

```
val := a[2][3]
或
var value int = a[2][3]
```

以上实例访问了二维数组 val 第三行的第四个元素。

二维数组可以使用循环嵌套来输出元素：

```
package main

import "fmt"

func main() {
   /* 数组 - 5 行 2 列*/
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int

   /* 输出数组元素 */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
#结果
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

以下实例创建各个维度元素数量不一致的多维数组：

```
package main
```

```
import "fmt"

func main() {
    // 创建空的二维数组
    animals := [][]string{}

    // 创建三一维数组，各数组长度不同
    row1 := []string{"fish", "shark", "eel"}
    row2 := []string{"bird"}
    row3 := []string{"lizard", "salamander"}

    // 使用 append() 函数将一维数组添加到二维数组中
    animals = append(animals, row1)
    animals = append(animals, row2)
    animals = append(animals, row3)

    // 循环输出
    for i := range animals {
        fmt.Printf("Row: %v\n", i)
        fmt.Println(animals[i])
    }
}
#结果
Row: 0
[fish shark eel]
Row: 1
[bird]
Row: 2
[lizard salamander]
```

range 方式循环二维数组：

```
package main

import "fmt"

func main() {
  arr := [...][]int{
    {1, 2, 3, 4},
    {10, 20, 30, 40},
  }
  for i := range arr {
    for j := range arr[i] {
        fmt.Println(arr[i][j])
    }
  }
}
#结果
1
2
3
4
10
20
```

```
30
40
```

Go 语言向函数传递数组

如果你想向函数传递数组参数，你需要在函数定义时，声明形参为数组，我们可以通过以下两种方式来声明

方式一

形参设定数组大小：

```
void myFunction(param [10]int)
{
.
.
.
}
```

方式二

形参未设定数组大小：

```
void myFunction(param []int)
{
.
.
.
}
```

```
func main() {
    var array = []int{1, 2, 3, 4, 5}
    /* 未定义长度的数组只能传给不限制数组长度的函数 */
    setArray(array)
    /* 定义了长度的数组只能传给限制了相同数组长度的函数 */
    var array2 = [5]int{1, 2, 3, 4, 5}
    setArray2(array2)
}

func setArray(params []int) {
    fmt.Println("params array length of setArray is : ", len(params))
}

func setArray2(params [5]int) {
    fmt.Println("params array length of setArray2 is : ", len(params))
}
#结果
params array length of setArray is :  5
params array length of setArray2 is :  5
初始化数组长度后，元素可以不进行初始化，或者不进行全部初始化，但未进行数组大小初始化的数组初始化结果元素大小就为多少。
```

把一个大数组传递给函数会消耗很多内存。有两种方法可以避免这种现象：

- 传递数组的指针
- 使用数组的切片

```
package main
import "fmt"
// Go 语言的数组是值，其长度是其类型的一部分，作为函数参数时，是 值传递，函数中的修改对调用者不可见
func change1(nums [3]int) {
    nums[0] = 4
}
// 传递进来数组的内存地址，然后定义指针变量指向该地址，则会改变数组的值
func change2(nums *[3]int) {
    nums[0] = 5
}
// Go 语言中对数组的处理，一般采用 切片 的方式，切片包含对底层数组内容的引用，作为函数参数时，类似于 指针传递，函数中的修改对调用者可见
func change3(nums []int) {
    nums[0] = 6
}
func main() {
    var nums1 = [3]int{1, 2, 3}
    var nums2 = []int{1, 2, 3}
    change1(nums1)
    fmt.Println(nums1)  //  [1 2 3]
    change2(&nums1)
    fmt.Println(nums1)  //  [5 2 3]
    change3(nums2)
    fmt.Println(nums2)  //  [6 2 3]
}
```