数据结构

类型

什么是类型?

内存中的二进制数据本身没什么区别，就是一串0或1的组合。

内存中有一个字节内容是0x63，它究竟是什么？字符串？字符？还是整数?

本来0x63表示数字，但是文字必须编码成为0和1的组合，才能记录在计算机系统中。在计算机世界里，一切都是数字，但是一定需要指定类型才能正确的理解它的含义。

如果0x63是整数，它就属于整数类型，它是整数类型的一个具体的实例。整数类型就是一个抽象的概念，它是对一类有着共同特征的事物的抽象概念。它展示出来就是99，因为多数情况下，程序按照人们习惯采用10进制输出。

如果0x63是byte类型或rune类型，在Go语言中，它是不同于整型的类型，但是展示出来同样是99。

如果0x63是string类型，则展示出一个字符的字符串“c”。

```
var a = 0x63
fmt.Printf("%T %[1]d %[1]c\n", a)
var b byte = 0x63
fmt.Printf("%T %[1]d %[1]c\n", b)
var c rune = '\x63'
fmt.Printf("%T %[1]d %[1]c\n", c)

var d = "\x63"
fmt.Printf("%T %[1]s\n", d)
fmt.Printf("%T %[1]s\n", string(a)) // 和上一行对比一下，体会一下

运行结果如下
int 99 c
uint8 99 c
int32 99 c
string c
string c
```

数值处理

取整

```
fmt.Println(1/2, 3/2, 5/2)
fmt.Println(-1/2, -3/2, -5/2)
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
fmt.Println(math.Ceil(2.01), math.Ceil(2.5), math.Ceil(2.8))
fmt.Println(math.Ceil(-2.01), math.Ceil(-2.5), math.Ceil(-2.8))
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
fmt.Println(math.Floor(2.01), math.Floor(2.5), math.Floor(2.8))
```

```
fmt.Println(math.Floor(-2.01), math.Floor(-2.5), math.Floor(-2.8))
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
fmt.Println(math.Round(2.01), math.Round(2.5), math.Round(2.8))
fmt.Println(math.Round(-2.01), math.Round(-2.5), math.Round(-2.8))
fmt.Println(math.Round(0.5), math.Round(1.5), math.Round(2.5),
math.Round(3.5))

运行结果
0 1 2
0 -1 -2
~~~~~~~~~~~~~~~~~~~~~~~~~~
3 3 3
-2 -2 -2
~~~~~~~~~~~~~~~~~~~~~~~~~~
2 2 2
-3 -3 -3
~~~~~~~~~~~~~~~~~~~~~~~~~~
2 3 3
-2 -3 -3
1 2 3 4
```

- / 整数除法，截取整数部分
- math.Ceil 向上取整
- math.Floor 向下取整
- math.Round 四舍五入

其它数值处理

```go
fmt.Println(math.Abs(-2.7))                                  // 绝对值
fmt.Println(math.E, math.Pi)                                 // 常数
fmt.Println(math.MaxInt16, math.MinInt16)                    // 常量，极值
fmt.Println(math.Log10(100), math.Log2(8))                   // 对数
fmt.Println(math.Max(1, 2), math.Min(-2, 3))                 // 最大值、最小值
fmt.Println(math.Pow(2, 3), math.Pow10(3))                   // 幂
fmt.Println(math.Mod(5, 2), 5%2)                             // 取模
fmt.Println(math.Sqrt(2), math.Sqrt(3), math.Pow(2, 0.5)) // 开方
```

math库中还有三角函数。

标准输入

Scan：空白字符分割，回车提交。换行符当做空白字符

```go
package main

import (
    "fmt"
)

func main() {
    var n int
    var err error
```

```
    var word1, word2 string
    fmt.Print("Plz input two words: ")
    n, err = fmt.Scan(&word1, &word2) // 控制台输入时，单词之间空白字符分割
    if err != nil {
        panic(err)
    }
    fmt.Println(n)
    fmt.Printf("%T %s, %T %s\n", word1, word1, word2, word2)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    var i1, i2 int
    fmt.Println("Plz input two ints: ")
    n, err = fmt.Scan(&i1, &i2)
    if err != nil {
        panic(err)
    }
    fmt.Println(n)
    fmt.Printf("%T %[1]d, %T %[2]d", i1, i2)
}
```

如果少一个数据，Scan就会阻塞；如果输入数据多了，等下回Scan读取。例如，一次性输入a b 1 2看看效果。

Scanf：读取输入，按照格式匹配解析。如果解析失败，立即报错，那么就会影响后面的Scanf。

```
package main

import (
    "fmt"
)

func main() {
    var n int
    var err error
    var name string
    var age int
    fmt.Print("Plz input your name and age: ")
    n, err = fmt.Scanf("%s %d\n", &name, &age) // 这里要有\n以匹配回车
    if err != nil {
        panic(err)
    }
    fmt.Println(n, name, age)

    var weight, height int
    fmt.Print("weight and height: ")
    _, err = fmt.Scanf("%d %d", &weight, &height)
    if err != nil {
        panic(err)
    }
    fmt.Printf("%T %[1]d, %T %[2]d", weight, height)
}
```

`fmt.Scanf("%s,%d", &name, &age)`中%s会和后面的非空白字符分不清楚，用`abc,20`是匹配不上的，因为除空白字符外，都可以看做是字符串。所以，建议格式字符串中，一律使用空格等空白字符分割。

线性数据结构

线性表

- 线性表（简称表），是一种抽象的数学概念，是一组元素的序列的抽象，它由有穷个元素组成（0个或任意个）
- 顺序表：使用一大块连续的内存顺序存储表中的元素，这样实现的表称为顺序表，或称连续表
  - 在顺序表中，元素的关系使用顺序表的存储顺序自然地表示
- 链接表：在存储空间中将分散存储的元素链接起来，这种实现称为链接表，简称链表

数组等类型，如同地铁站排好的队伍，有序，可以插队、离队，可以索引。

链表，如同操场上**手拉手**的小朋友，有序但排列随意。或者可以想象成一串带线的珠子，随意盘放在桌上。也可以离队、插队，也可以索引。

对比体会一下，这两种数据结构的增删改查。

数组

- 长度不可变
- 内容可变
- 可索引
- 值类型
- 顺序表

定义

```go
// 注意下面2种区别
var a0 [3]int                    // 零值初始化3个元素的数组
var a1 = [3]int{}                // 零值初始化3个元素的数组
// [3]int是类型，[3]int{} 是字面量值

var a2 [3]int = [3]int{1, 3, 5} // 声明且初始化，不推荐，啰嗦
var a3 = [3]int{1, 3, 5}         // 声明且初始化，推荐


count := 3
a4 := [count] int{1,3,5} // 错误的长度类型，必须是常量，换成const
fmt.Println(a2, a3)

const count = 3
a4 := [count]int{1, 3, 5} // 正确
fmt.Println(a2, a3, a4)

a5 := [...]int {10, 30, 50} // ...让编译器确定当前数组大小
```

```
19
20 a6 := [5]int{100, 200}        // 顺序初始化前面的，其余用零值填充
21 a7 := [5]int{1: 300, 3: 400} // 指定索引位置初始化，其余用零值填充
22
23 // 二维数组
24 a8 := [2][3]int{{100}} // 两行三列 [[100 0 0] [0 0 0]]
25 // [[10 0 0] [11 12 0] [13 14 15] [16 0 0]]
26 // 多维数组，只有第一维才能用...推测
27 // 第一维有4个，第二维有3个。可以看做4行3列的表
28 a9 := [...][3]int{{10}, {11, 12}, {13, 14, 15}, {16}}
```

长度和容量

- cap即capacity，容量，表示给数组分配的内存空间可以容纳多少**个**元素
- len即length，长度，指的是容器中目前有几**个**元素

由于数组创建时就**必须确定**的元素个数，且不能改变长度，所以不需要预留多余的内存空间，因此cap和len对数组来说一样。

索引

Go语言不支持负索引。通过[index]来获取该位置上的值。索引范围就是[0, 长度-1]。

修改

```
1 a5 := [...]int{10, 30, 50}
2 a5[0] += 100
```

遍历

1、索引遍历

```
1 a5 := [...]int{10, 30, 50}
2 for i := 0; i < len(a5); i++ {
3     fmt.Println(i, a5[i])
4 }
```

2、for-range遍历

```
1 a5 := [...]int{10, 30, 50}
2 for i, v := range a5 {
3     fmt.Println(i, v, a5[i])
4 }
```

内存模型

```
1 var a [3]int // 内存开辟空间存放长度为3的数组，零值填充
2 for i := 0; i < len(a); i++ {
3     fmt.Println(i, a[i], &a[i])
4 }
5 fmt.Printf("%p %p, %v\n", &a, &a[0], a)
6 a[0] = 1000
7 fmt.Printf("%p %p, %v\n", &a, &a[0], a)
8
```

```
运行结果
0 0 0xc0000101b0
1 0 0xc0000101b8
2 0 0xc0000101c0
0xc0000101b0 0xc0000101b0, [0 0 0]
0xc0000101b0 0xc0000101b0, [1000 0 0]
```

- 数组必须在编译时就确定大小，之后不能改变大小
- 数组首地址就是数组地址
- 所有元素一个接一个顺序存储在内存中
- 元素的值可以改变，但是元素地址不变

上面每个元素间隔8个字节，正好64位，符合int类型定义。

如果数据元素是字符串类型呢?

```go
var a = [3]string{"abc", "def", "xyz"} // 内存开辟空间存放长度为3的数组
for i := 0; i < len(a); i++ {
    fmt.Println(i, a[i], &a[i])
}
fmt.Printf("%p %p, %v\n", &a, &a[0], a)
a[0] = "oooooo"
fmt.Printf("%p %p, %v\n", &a, &a[0], a)

运行结果
0 abc 0xc000138480
1 def 0xc000138490
2 xyz 0xc0001384a0
0xc000138480 0xc000138480, [abc def xyz]
0xc000138480 0xc000138480, [oooooo def xyz]
```

- 数组首地址就是数组地址
- 所有元素顺序存储在内存中
- 元素的值可以改变，但是元素地址不变

每个元素间隔16个字节，为什么？“abc“是几个字节？这说明什么?

值类型

```go
package main

import "fmt"

// 提前认识一下函数，不会就抄
func showAddr(arr [3]int) [3]int {
    fmt.Printf("%v, %p\n", arr, &arr)
    return arr
}

func main() {
    a1 := [...]int{10, 30, 50}
    fmt.Printf("%v, %p\n", a1, &a1)
    a2 := a1
```

```
    fmt.Printf("%v, %p\n", a2, &a2)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    a3 := showAddr(a1)
    fmt.Printf("%v, %p\n", a3, &a3)
}

结果如下
[10 30 50], 0xc00000c1c8
[10 30 50], 0xc00000c1f8
~~~~~~~~~~~~~~~~~~~~~~~~~~~
[10 30 50], 0xc00000c240
[10 30 50], 0xc00000c228
```

可以看出a1、a2、a3、a4的地址都不一样，最不可思议的是，a2 := a1后两个变量地址也不一样。

这说明，Go语言在这些地方对数组进行了值拷贝，都生成了一份副本。

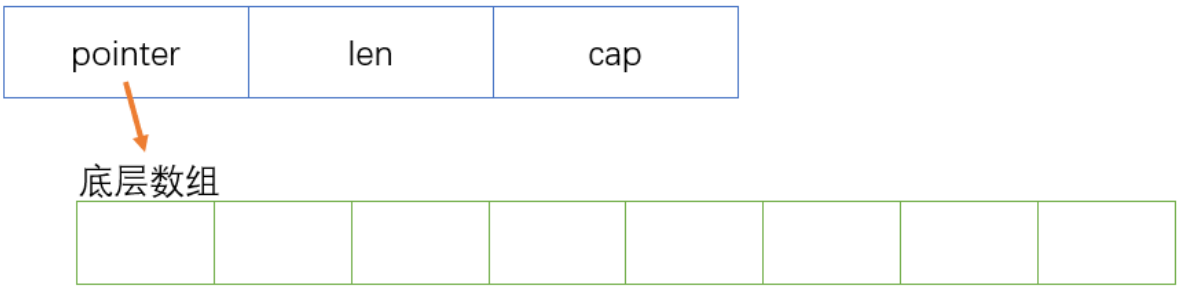
切片

- 长度可变
- 内容可变
- 引用类型
- 底层基于数组

定义

```
var s1 []int                // 长度、容量为0的切片，零值
var s2 = []int{}            // 长度、容量为0的切片，字面量定义
var s3 = []int{1, 3, 5}     // 字面量定义，长度、容量都是3
var s4 = make([]int, 0)     // 长度、容量为0的切片，make([]T, length)
var s5 = make([]int, 3, 5) // 长度为3，容量为5，底层数组为长度为5，元素长度为3，所以显示[0, 0, 0]
```

内存模型



切片本质是对底层数组一个连续**片段**的引用。此片段可以是整个底层数组，也可以是由起始和终止索引标识的一些项的子集。

```
// https://github.com/golang/go/blob/master/src/runtime/slice.go
type slice struct {
    array unsafe.Pointer
    len int
    cap int
}
```

```
a := []int{1, 3, 5, 7}
fmt.Printf("%v, %p, %p", a, &a, &a[0])

结果如下
[1 3 5 7], 0xc000004078, 0xc000012200
```

&a是切片结构体的地址，&a[0]是底层数组的地址。

追加

append：在切片的尾部追加元素，长度加1。

增加元素后，有可能超过当前容量，导致切片扩容。

长度和容量

```
s1 := make([]int, 3, 5)
fmt.Printf("s1 %p, %p, l=%-2d, c=%-2d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
s2 := append(s1, 1, 2) // append返回一个新的切片
fmt.Printf("s1 %p, %p, l=%-2d, c=%-2d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
fmt.Printf("s2 %p, %p, l=%-2d, c=%-2d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
```

```
目前没有超过容量，底层共用同一个数组，但是，对底层数组使用的片段不一样
s1 0xc000008078, 0xc00000e390, l=3 , c=5 , [0 0 0]
s1 0xc000008078, 0xc00000e390, l=3 , c=5 , [0 0 0]
s2 0xc0000080a8, 0xc00000e390, l=5 , c=5 , [0 0 0 1 2]
```

```
s3 := append(s1, -1)
fmt.Printf("s1 %p, %p, l=%-2d, c=%-2d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
fmt.Printf("s2 %p, %p, l=%-2d, c=%-2d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
fmt.Printf("s3 %p, %p, l=%-2d, c=%-2d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
```

```
目前三个切片底层用同一个数组，只不过长度不一样
s1 0xc000008078, 0xc00000e390, l=3 , c=5 , [0 0 0]
s2 0xc0000080a8, 0xc00000e390, l=5 , c=5 , [0 0 0 -1 2]
s3 0xc0000080f0, 0xc00000e390, l=4 , c=5 , [0 0 0 -1]
```

```
s4 := append(s3, 3, 4, 5)
fmt.Printf("s1 %p, %p, l=%-2d, c=%-2d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
fmt.Printf("s2 %p, %p, l=%-2d, c=%-2d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
fmt.Printf("s3 %p, %p, l=%-2d, c=%-2d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
fmt.Printf("s4 %p, %p, l=%-2d, c=%-2d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)
```

```
底层数组变了，容量也增加了
s1 0xc000008078, 0xc00000e390, l=3 , c=5 , [0 0 0]
s2 0xc0000080a8, 0xc00000e390, l=5 , c=5 , [0 0 0 -1 2]
s3 0xc0000080f0, 0xc00000e390, l=4 , c=5 , [0 0 0 -1]
s4 0xc000008150, 0xc000012280, l=7 , c=10, [0 0 0 -1 3 4 5]
```

```
s5 := append(s4, 6, 7, 8, 9)
fmt.Printf("s1 %p, %p, l=%-2d, c=%-2d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
fmt.Printf("s2 %p, %p, l=%-2d, c=%-2d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
fmt.Printf("s3 %p, %p, l=%-2d, c=%-2d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
fmt.Printf("s4 %p, %p, l=%-2d, c=%-2d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)
fmt.Printf("s5 %p, %p, l=%-2d, c=%-2d, %v\n", &s5, &s5[0], len(s5), cap(s5), s5)
```

```
s1 0xc000008078, 0xc00000e390, l=3 , c=5 , [0 0 0]
s2 0xc0000080a8, 0xc00000e390, l=5 , c=5 , [0 0 0 -1 2]
s3 0xc0000080f0, 0xc00000e390, l=4 , c=5 , [0 0 0 -1]
s4 0xc000008150, 0xc000012280, l=7 , c=10, [0 0 0 -1 3 4 5]
s5 0xc0000081c8, 0xc0000260a0, l=11, c=20, [0 0 0 -1 3 4 5 6 7 8 9]
```

- append一定返回一个新的切片
- append可以增加若干元素
  - 如果增加元素时，当前长度 + 新增个数 <= cap则不扩容
    - 原切片使用原来的底层数组，返回的新切片也使用这个底层数组
    - 返回的新切片有新的长度
    - 原切片长度不变
  - 如果增加元素时，当前长度 + 新增个数 > cap则需要扩容
    - 生成新的底层数组，新生成的切片使用该新数组，将旧元素复制到新数组，其后追加新元素
    - 原切片底层数组、长度、容量不变

扩容策略

https://go.dev/src/runtime/slice.go

（老版本）实际上，当扩容后的cap<1024时，扩容翻倍，容量变成之前的2倍；当cap>=1024时，变成之前的1.25倍。

（新版本1.18+）阈值变成了256，当扩容后的cap<256时，扩容翻倍，容量变成之前的2倍；当cap>=256时，`newcap += (newcap + 3*threshold) / 4 计算后就是 newcap = newcap + newcap/4 + 192`，即1.25倍后再加192。

扩容是创建新的内部数组，把原内存数据拷贝到新内存空间，然后在新内存空间上执行元素追加操作。

切片频繁扩容成本非常高，所以尽量早估算出使用的大小，一次性给够，建议使用make。常用`make([]int, 0, 100)`。

思考一下：如果`s1 := make([]int, 3, 100)`，然后对s1进行append元素，会怎么样？

引用类型

```go
package main

import (
    "fmt"
)

func showAddr(s []int) []int {
    fmt.Printf("s  %p, %p, %d, %d, %v\n", &s, &s[0], len(s), cap(s), s)
    // 修改一个元素
    if len(s) > 0 {
        s[0] = 123
    }
    return s
}

func main() {
    s1 := []int{10, 20, 33}
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    s2 := s1
    fmt.Printf("s2 %p, %p, %d, %d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    s3 := showAddr(s1)
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    fmt.Printf("s2 %p, %p, %d, %d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
    fmt.Printf("s3 %p, %p, %d, %d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
}

运行结果
s1 0xc000008078, 0xc0000101b0, 3, 3, [10 20 33]
s2 0xc0000080a8, 0xc0000101b0, 3, 3, [10 20 33]
~~~~~~~~~~~~~~~~~~~~~~~~~~~
s  0xc0000080f0, 0xc0000101b0, 3, 3, [10 20 33]
s1 0xc000008078, 0xc0000101b0, 3, 3, [123 20 33]
s2 0xc0000080a8, 0xc0000101b0, 3, 3, [123 20 33]
```

```
s3 0xc0000080d8, 0xc0000101b0, 3, 3, [123 20 33]
```

这说明，底层数组是同一份，修改切片中的某个已有元素，那么所有切片都能看到。

那如果在上面showAddr函数中对切片增加一个元素会怎么样呢?

增加一个元素会导致扩容，会怎么样呢？请先在脑中思考

```go
package main

import (
    "fmt"
)

func showAddr(s []int) []int {
    fmt.Printf("s  %p, %p, %d, %d, %v\n", &s, &s[0], len(s), cap(s), s)
    // // 修改一个元素
    // if len(s) > 0 {
    //  s[0] = 123
    // }
    s = append(s, 100, 200) // 覆盖s，请问s1会怎么样
    fmt.Printf("s  %p, %p, %d, %d, %v\n", &s, &s[0], len(s), cap(s), s)
    return s
}

func main() {
    s1 := []int{10, 20, 30}
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    s2 := s1
    fmt.Printf("s2 %p, %p, %d, %d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    s3 := showAddr(s1)
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    fmt.Printf("s2 %p, %p, %d, %d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
    fmt.Printf("s3 %p, %p, %d, %d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
}
```

```
运行结果
s1 0xc000008078, 0xc0000101b0, 3, 3, [10 20 30]
s2 0xc0000080a8, 0xc0000101b0, 3, 3, [10 20 30]
~~~~~~~~~~~~~~~~~~~~~~~~~~
s  0xc0000080f0, 0xc0000101b0, 3, 3, [10 20 30]
s  0xc0000080f0, 0xc00000e390, 5, 6, [10 20 30 100 200]
s1 0xc000008078, 0xc0000101b0, 3, 3, [10 20 30]
s2 0xc0000080a8, 0xc0000101b0, 3, 3, [10 20 30]
s3 0xc0000080d8, 0xc00000e390, 5, 6, [10 20 30 100 200]
```

可以看到showAddr传入s1，但是返回的s3已经和s1不共用同一个底层数组了，分道扬镳了。

其实这里还是值拷贝，不过拷贝的是切片的标头值（Header）。标头值内指针也被复制，刚复制完大家指向同一个底层数组罢了。但是仅仅知道这些不够，因为一旦操作切片时扩容了，或另一个切片增加元素，那么就不能简单归结为“切片是引用类型，拷贝了地址”这样简单的话来解释了。要具体问题，具体分析。

Go语言中全都是值传递，整型、数组这样的类型的值是完全复制，slice、map、channel、interface、function这样的引用类型也是值拷贝，不过复制的是标头值。

截取子切片

切片可以通过指定索引区间获得一个子切片，格式为slice[start:end]，规则就是前包后不包。

```go
package main

import (
    "fmt"
)

func main() {
    s1 := []int{10, 30, 50, 70, 90} // 容量、长度为5，索引0、1、2、3、4
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    s2 := s1 // 和s1共用底层数组
    fmt.Printf("s2 %p, %p, %d, %d, %v\n", &s2, &s2[0], len(s2), cap(s2), s2)
    s3 := s1[:] // 和s1共用底层数组，从头到尾元素都要
    fmt.Printf("s3 %p, %p, %d, %d, %v\n", &s3, &s3[0], len(s3), cap(s3), s3)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    s4 := s1[1:] // 掐头，容量、长度都为4，首地址偏移1个元素，共用底层数组
    fmt.Printf("s4 %p, %p, %d, %d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)
    s5 := s1[1:4] // 掐头去尾，容量为4，长度为3，首地址偏移1个元素，共用底层数组
    fmt.Printf("s5 %p, %p, %d, %d, %v\n", &s5, &s5[0], len(s5), cap(s5), s5)
    s6 := s1[:4] // 去尾，容量为5，长度为4，首地址不变，共用底层数组
    fmt.Printf("s6 %p, %p, %d, %d, %v\n", &s6, &s6[0], len(s6), cap(s6), s6)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    s7 := s1[1:1] // 首地址偏移1个元素，长度为0，容量为4，共用底层数组
    // fmt.Printf("s7 %p, %p, %d, %d, %v\n", &s7, &s7[0], len(s7), cap(s7), s7) // 由于长度为0，所以不能s7[0]报错
    fmt.Printf("s7 %p, %d, %d, %v\n", &s7, len(s7), cap(s7), s7)
    // 请问，为s7增加一个元素，s1、s7分别是什么？
    s8 := s1[4:4] // 首地址偏移4个元素，长度为0，容量为1，因为最后一个元素没在切片中，共用底层数组
    fmt.Printf("s8 %p, %d, %d, %v\n", &s8, len(s8), cap(s8), s8)
    s9 := s1[5:5] // 首地址偏移5个元素，长度为0，容量为0，共用底层数组
    fmt.Printf("s9 %p, %d, %d, %v\n", &s9, len(s9), cap(s9), s9)
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    s9 = append(s9, 11) // 增加元素会怎么样？s1、s9分别是什么？
    fmt.Printf("s9 %p, %p, %d, %d, %v\n", &s9, &s9[0], len(s9), cap(s9), s9)
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
}
```

可以看出，切这个操作都是从同一个底层数组上取的段，所以子切片和原始切片共用同一个底层数组

- start默认为0，end默认为len(slice)即**切片长度**
- 通过指针确定底层数组从哪里开始共享
- 长度为end-start
- 容量是底层数组从偏移的元素到结尾还有几个元素

数组也可以切片，会生成新的切片

```go
package main

import (
    "fmt"
)

func main() {
    s1 := [5]int{10, 30, 50, 70, 90} // 容量、长度为5，索引0、1、2、3、4
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)

    s4 := s1[1:] // 掐头，容量、长度都为4，首地址偏移1个元素，以s1为底层数组
    fmt.Printf("s4 %p, %p, %d, %d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)

    s4[0] = 100 // s1、s4分别是什么？
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    fmt.Printf("s4 %p, %p, %d, %d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)

    s4 = append(s4, 11, 22) // 是否扩容？会怎样？
    fmt.Printf("s1 %p, %p, %d, %d, %v\n", &s1, &s1[0], len(s1), cap(s1), s1)
    fmt.Printf("s4 %p, %p, %d, %d, %v\n", &s4, &s4[0], len(s4), cap(s4), s4)
}
```

切片总结：

- 使用slice[start:end]表示切片，切片长度为end-start，前包后不包
- start缺省，表示从索引0开始
- end缺省，表示取到末尾，包含最后一个元素，特别注意这个值是**len(slice)即切片长度**，**不是容量**
  - a1[5:]相当于a1[5:len(a1)]
- start和end都缺省，表示从头到尾
- start和end同时给出，要求end >= start
  - start、end**最大都不可以超过容量值**
  - 假设当前容量是8，长度为5，有以下情况
    - a1[:8]，可以，end最多写成8(因为后不包)，a1[:9]不可以
    - a1[8:]，不可以，end缺省为5，等价于a1[8:5]
    - a1[8:8]，可以，但这个切片容量和长度都为0了
    - a1[7:7]，可以，但这个切片长度为0，容量为1
    - a1[0:0]，可以，但这个切片长度为0，容量为8
    - a1[:8]，可以，这个切片长度为8，容量为8，这8个元素都是原序列的
    - a1[1:5]，可以，这个切片长度为4，容量为7，相当于跳过了原序列第一个元素
- 切片刚产生时，和原序列（数组、切片）**开始**共用同一个底层数组，但是**每一个切片都自己独立保存着指针、cap和len**
- 一旦一个切片扩容，就和原来共用一个底层数组的序列分道扬镳，从此陌路

常见线性数据结构

- 数组array
- 链表Linked List
- 栈Stack
- 队列Queue

问题

1. 数组、链表各自有什么优劣？
2. 队列的作用？FIFO、LIFO是什么，应如何实现？