

# Go 字符串及strings包常见操作

## 1、字符串<sup>Q</sup>的定义

字符串是不可变值类型，内部用指针指向 UTF-8 字节数组。

Go 语言中可以使用**反引号**或者**双引号**来定义字符串。反引号表示原生的字符串，即不进行转义。Go 语言的字符串不支持单引号

- 1

• 默认值是空字符串 ""。
- 2

• 用索引号访问某字节，如 `s[i]`。
- 3

• 不能用序号获取字节元素指针，`&s[i]` 非法。
- 4

• 不可变类型，无法修改字节数组。
- 5

• 字节数组尾部不包含 `NULL`。

- 双引号

字符串使用双引号括起来，其中的相关的**转义字符将被替换**。例如：

```
1 str := "Hello World! \n Hello Gopher! \n"
2
3 输出:
4 Hello World!
5 Hello Gopher!
```

- 反引号

字符串使用反引号括起来，其中的相关的**转义字符不会被替换**。例如：

```
1 str := `Hello World! \n Hello Gopher! \n`
2
3 输出:
4 Hello World! \nHello Gopher! \n
```

总结：双引号中的 **转义<sup>Q</sup>** 字符被替换，而反引号中原生字符串中的 `\n` 会被原样输出。

## 2、字符串转义符

Go 语言的字符串常见转义符包含回车、换行、单双引号、制表符等，如下表所示。

转义	含义
<code>\r</code>	回车符（返回行首）
<code>\n</code>	换行符（直接跳到下一行的同列位置）
<code>\t</code>	制表符
<code>'</code>	单引号
<code>"</code>	双引号
<code>\\</code>	反斜杠自身

举个例子：

```
1 fmt.Println("人生苦短\n我要转go")
```

## 3、多行字符串

Go语言中要定义一个多行字符串时，就必须使用 **反引号** 字符

```
1 s1 := `第一行
2     第二行
3     第三行
4     `
5 fmt.Println(s1)
```

反引号间换行将被作为字符串中的换行，但是所有的转义字符均无效，文本将会原样输出。

## 4、byte 和 rune 类型

单引号可直接打印字符对应的ASCII码值

组成每个字符串的元素叫做“字符”，可以通过遍历或者单个获取字符串元素获得字符。字符用单引号（'）包裹起来，如：

```
1 var s1 := 'a'      // 97
2
3 var s2 := 'A'      // 65
```

Go 语言的字符有以下两种：

- uint8类型，或者叫 byte 型，代表了ASCII码的一个字符。
- rune类型，代表一个 UTF-8字符。

字符串底层是一个byte数组，所以可以和[]byte类型相互转换。字符串是不能修改的 字符串是由byte字节组成，所以字符串的长度是byte字节的长度。 rune类型用来表示utf8字符，一个rune字符由一个或多个byte组成。

当需要处理中文、日文或者其他复合字符时，则需要用到 rune 类型。 rune 类型实际是一个 int32 。 Go 使用了特殊的 rune 类型来处理 Unicode ，让基于 Unicode 的文本处理更为方便，也可以使用 byte 型进行默认字符串处理，性能和扩展性都有照顾。

实际上，Go语言的range循环在处理字符串的时候，会自动隐式解码UTF8字符串。

```
1 // 遍历字符串
2 func traversalString() {
3     s := "pprof.cn博客"
4     for i := 0; i < len(s); i++ {           //byte
5         fmt.Printf("%v(%c) ", s[i], s[i])
6     }
7     fmt.Println()
8
9     for _, r := range s {                   //rune, 用于遍历带有中文字符的字符串
10        fmt.Printf("%v(%c) ", r, r)
11    }
12    fmt.Println()
13 }
```

输出：

```
1 112(p) 112(p) 114(r) 111(o) 102(f) 46(.) 99(c) 110(n) 229(ǎ) 141( ) 154( ) 229(ǎ) 174(®) 162(¢)
2 112(p) 112(p) 114(r) 111(o) 102(f) 46(.) 99(c) 110(n) 21338(博) 23458(客)
```

因为UTF8编码下一个中文汉字由 3~4 个字节组成，所以我们不能简单的按照字节去遍历一个包含中文的字符串，否则就会出现上面输出中第一行的结果。

字符串底层是一个byte数组，所以可以和[]byte类型相互转换。字符串是不能修改的 字符串是由byte字节组成，所以字符串的长度是byte字节的长度。 rune类型用来表示utf8字符，一个rune字符由一个或多个byte组成。

## 5、字符串遍历

### 5.1 字节长度

字符串的内容（纯字节）可以通过标准索引法来获取，在中括号 [] 内写入索引，索引从 0 开始计数：

- 字符串 str 的第 1 个字节：str[0] 第 1 个字节
- str[i - 1] 最后 1 个字节：str[len(str) - 1]
- 获取字符串所占的字节长度，如：len(str)

注意：内置的len()函数获取的是每个字符的UTF-8编码的长度和，而不是直接的字符数量。

```
1 package main
2
```

```

3 import (
4     "fmt"
5     "unicode/utf8"
6 )
7
8 func main() {
9     // 一个中文字符编码需要三个字节
10    s := "其实就是rune"
11    fmt.Println(len(s))           // "16" , 字节数, 3 * 4 + 4 = 16 字节
12    fmt.Println(utf8.RuneCountInString(s)) // "8" , 字符数, 共 8 个字符数
13 }

```

## 5.2 字符数量

for range循环处理字符时，不是按照字节的方式来处理的。v 其实际上是一个rune类型值。实际上，Go语言的range循环在处理字符串的时候，会自动隐式解码UTF8字符串。如字符串含有中文等字符，我们可以看到每个中文字符的索引值相差3。

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     s := "Go语言四十二章经"
9     for k, v := range s {
10        fmt.Printf("k: %d,v: %c == %d\n", k, v, v)
11    }
12 }

```

因为一个中文字符编码需要三个字节，转化单个字节会出现乱码。如字符串含有中文等字符，我们可以看到每个中文字符的索引值相差3。

```

1 程序输出:
2 k: 0,v: G == 71
3 k: 1,v: o == 111
4 k: 2,v: 语 == 35821
5 k: 5,v: 言 == 35328
6 k: 8,v: 四 == 22235
7 k: 11,v: 十 == 21313
8 k: 14,v: 二 == 20108
9 k: 17,v: 章 == 31456
10 k: 20,v: 经 == 32463

```

## 5.3 字符串切片

在 Go 语言中，可以通过字符串切片实现获取子串的功能，切片区间可以对比数学中的区间概念来理解，它是一个左闭右开的区间（索引值非负）：

```

1 str := "hello, golang 风清扬"
2 str1 := str[:5] // 获取索引5 (不含) 之前的子串
3 str2 := str[7:] // 获取索引7 (含) 之后的子串
4 str3 := str[0:5] // 获取从索引0 (含) 到索引5 (不含) 之间的子串
5 str4 := str[:] // 打印完整的字符串
6 fmt.Println("str1:", str1)
7 fmt.Println("str2:", str2)
8 fmt.Println("str3:", str3)
9 fmt.Println("str4:", str4)

```

输出：

```

1 str1: hello
2 str2: golang 风清扬
3 str3: hello
4 str4: hello, golang 风清扬

```

## 6、字符串修改

要修改字符串，需要先将其转换成 []rune或[]byte，完成后再转换为 string。无论哪种转换，都会重新分配内存，并复制字节数组。

```

1 func changeString() {
2     s1 := "hello"
3     // 强制类型转换
4     byteS1 := []byte(s1)
5     byteS1[0] = 'H'

```

```
6      fmt.Println(string(byteS1))
7
8      s2 := "博客"
9      runeS2 := []rune(s2)
10     runeS2[0] = '狗'
11     fmt.Println(string(runeS2))
12 }
```

## 7、字符串类型转换

Go语言中只有强制类型转换，没有隐式类型转换。该语法只能在两个类型之间支持相互转换的时候使用。

强制类型转换的基本语法如下：

```
1 | T(表达式)
```

其中，T表示要转换的类型。表达式包括变量、复杂算子和函数返回值等。

比如计算直角三角形的斜边长时使用math包的Sqrt()函数，该函数接收的是float64类型的参数，而变量a和b都是int类型的，这个时候就需要将a和b强制类型转换为float64类型。

```
1  func sqrtDemo() {
2      var a, b = 3, 4
3      var c int
4      // math.Sqrt()接收的参数是float64类型，需要强制转换
5      c = int(math.Sqrt(float64(a*a + b*b)))
6      fmt.Println(c)
7  }
```

## 8、strings 包 常用操作

说明：这里说的字符，指得是 rune 类型，即一个 UTF-8 字符（Unicode 代码点）。

### 8.1 字符串比较

- 对字符串的比较除了使用内置的 == 符号来比较，strings标准库还提供了两个方法分别是 Compare 和 EqualFold
- 需要明确的是，**字符的比较是通过查出字符对应的ASCII码值，然后进行总和的计算再进行大小的比较**

#### 1) Compare

```
1  // Compare 函数，用于比较两个字符串的大小，如果两个字符串相等，返回为 0。
2  // 如果 a 小于 b，返回 -1，反之返回 1。
3  // 不推荐使用这个函数，直接使用 == != > < >= <= 等一系列运算符更加直观。
4  func Compare(a, b string) int
```

示例：比较字符串：相等输出0、大于输出1、小于输出-1

```
1  a := "gopher"
2  b := "golang"
3  fmt.Println(strings.Compare(a, b))    // 1    大于
4  fmt.Println(strings.Compare(a, a))    // 0    相等
5  fmt.Println(strings.Compare(b, a))    // -1   小于
```

#### 2) EqualFold

```
1  // EqualFold 函数，计算 s 与 t 忽略字母大小写后是否相等。
2  func EqualFold(s, t string) bool
```

- EqualFold 相比 Compare 忽略了字母大小来比较
- 返回的结果为布尔值，true代表相等，false代表不相等

```
1  fmt.Println(strings.EqualFold("GO", "go"))    // true
2  fmt.Println(strings.EqualFold("1", "一"))     // false
```

### 8.2 查询是否存在某个字符或子串

检查的字符串中是否包含有某个子串或字符的需求，比如"abcdefg"是否包含"efg"或者'a'，strings包主要提供了Contains的方法供开发者使用,返回值是布尔值。

## 1) Contains方法系

三个函数签名如下：

```
1 // 子串 substr 在 s 中, 返回 true
2 func Contains(s, substr string) bool
3
4 // chars 中任何一个 Unicode 代码点在 s 中, 返回 true
5 // 注意空字符的情况是返回false, 与上面的Contains区分开来
6 func ContainsAny(s, chars string) bool
7
8 // Unicode 代码点 r 在 s 中, 返回 true
9 func ContainsRune(s string, r rune) bool
```

示例：

- Contains

```
1 // Contains
2 fmt.Println(strings.Contains("adbcadeadh", "adb")) // true, 存在
3 fmt.Println(strings.Contains("adbcadeadh", "abc")) // false, 不存在
4 // 注意空子串和空格子串的区别:
5 fmt.Println(strings.Contains("abc", "")) // true, 存在
6 fmt.Println(strings.Contains("abc", " ")) // false, 不存在
```

- ContainsAny: 但凡字符串s内包含 chars 任意一个字符（Unicode Code Point） 返回 true

```
1 // ConmtainsAny
2 /*
3 第二个参数 chars 中任意一个字符 (Unicode Code Point)
4 如果在第一个参数 s 中存在, 则返回 true, 否则返回false
5 注意空字符的情况是返回false, 与上面的Contains区分开来
6 */
7 fmt.Println(strings.ContainsAny("team", "b")) // false
8 fmt.Println(strings.ContainsAny("team", "t")) // true
9 fmt.Println(strings.ContainsAny("team", "a & e ")) // true
10 fmt.Println(strings.ContainsAny("team", "a | e ")) // true
11 fmt.Println(strings.ContainsAny("team", "s g ")) // false
12 fmt.Println(strings.ContainsAny("team", " ")) // false
13 fmt.Println(strings.ContainsAny("team", "")) // false
14 fmt.Println(strings.ContainsAny("", "")) // false
15 fmt.Println(strings.ContainsAny("team", "ea")) // true
16 fmt.Println(strings.ContainsAny("team", "eg")) // true
17 fmt.Println(strings.ContainsAny("team", "ig")) // false
18 fmt.Println(strings.ContainsAny("team", "ge")) // true
```

- ContainsRune 查询单个字符的情况

```
1 // ContainsRune
2 // 查询单个字符的情况, 注意 rune 为 单个字符, 单引号
3 fmt.Println(strings.ContainsRune("team", 'a')) // ture
4 fmt.Println(strings.ContainsRune("team", 'k')) // false
5 fmt.Println(strings.ContainsRune("team", ' ')) // false
```

## 2) Contains方法系的内部实现

- 查看这三个函数的源码，发现它们只是调用了相应的 Index 函数（子串出现的位置）
- 然后和 0 作比较返回 true 或 fales。

```
1 func Contains(s, substr string) bool {
2     return Index(s, substr) >= 0
3 }
```

## 8.3 子串出现次数（字符串匹配）

在数据结构与算法中，可能会讲解以下字符串匹配算法：

- 朴素匹配算法
- KMP 算法
- Rabin-Karp 算法
- Boyer-Moore 算法

在 Go 中，查找子串出现次数即字符串模式匹配，实现的是 Rabin-Karp 算法。Count 函数的签名如下：

```
1 func Count(s, sep string) int
```

这里要特别说明一下的是当 sep 为空(“”)时，Count 的返回值是：utf8.RuneCountInString(s) + 1

```
1 fmt.Println(strings.Count("cheese", "e")) // 3
2 fmt.Println(len("百度中国")) // 12 个字节
3 fmt.Println(strings.Count("百度中国", "")) // 5, utf8.RuneCountInString(s) + 1
4 fmt.Println(strings.Count("爱学习的你", "学")) // 1
```

- 另外，Count 是计算子串在字符串中出现的**无重叠的次数**，比如：

```
1 fmt.Println(strings.Count("fiveveve", "vev")) // 1
```

## 8.4 字符或子串在字符串中出现的位置

strings.Index可以在字符串中搜索某个**子串**，并得到对应子串起始索引下标，若不存在对应子串则返回-1。

```
1 // 在 s 中查找 sep 的第一次出现，返回第一次出现的索引
2 func Index(s, sep string) int
```

除了对子串进行搜索之外，也可以对**某个字节，字符，字符集合**进行搜索。

### 1) 获取正向索引，从左往右匹配到第一个

```
1 // 字节搜索，在 s 中查找字节 c 的第一次出现，返回第一次出现的索引
2 func IndexByte(s string, c byte) int
3
4 // 字符搜索，Unicode 代码点 r 在 s 中第一次出现的位置
5 func IndexRune(s string, r rune) int
6
7 // 字符集合搜索，匹配chars中的任何一个字符
8 // 返回chars 中任何一个 Unicode 代码点在 s 中首次出现的位置
9 func IndexAny(s, chars string) int
10
11 // 查找字符 c 在 s 中第一次出现的位置，其中 c 满足 f(c) 返回 true
12 func IndexFunc(s string, f func(rune) bool) int
```

示例：

```
1 // 匹配字符第一次出现
2 fmt.Println(strings.Index("golang go go go", "g")) // 0
3 fmt.Println(strings.Index("golang go go go", "o")) // 1
4
5 // 匹配字节，单引号
6 fmt.Println(strings.IndexByte("hello", 'o')) // 4
7
8 // 匹配字符，单引号
9 fmt.Println(strings.IndexRune("风华正茂", '风')) // 0
10
11 // 匹配字符集合
12 fmt.Println(strings.IndexAny("golang", "l&g")) // 0
13 fmt.Println(strings.IndexAny("golang", "&o")) // 1
```

### 2) 获取反向索引，从右往左匹配到最后一个

- strings包也提供了一系列函数获取对应元素的最后一个匹配项的索引下标
- 对应于上面的每个Index函数，都有一个LastIndex函数

函数声明如下：

```
1 // 查找最后一次出现的位置
2 func LastIndex(s, sep string) int
3
4 func LastIndexByte(s string, c byte) int
5
6 func LastIndexAny(s, chars string) int
7
8 func LastIndexFunc(s string, f func(rune) bool) int
```

示例：

```
1 // 匹配字符最后一次出现
2 fmt.Println(strings.LastIndex("golang go go go", "go")) // 13
3 fmt.Println(strings.LastIndexByte("golang go go go", 'o')) // 14
4
5 // 匹配字节，单引号
6 fmt.Println(strings.LastIndexByte("hello", 'o')) // 4
7
8 // 匹配字符集合
9 fmt.Println(strings.LastIndexAny("golang", "l&g")) // 5
10 fmt.Println(strings.LastIndexAny("golang", "&o")) // 1
```

- 需要再次注意的是，`IndexAny` 和 `LastIndexAny` 返回的是Unicode码点对应的索引这
- 比如“你”和“好”都占用3个字节，“你好”匹配“好”，返回的就是3。

```
1 s := "你好帅！你好帅！"
2 fmt.Printf("%s的字节长度: %d\n", s, len(s)) // 24
3 fmt.Println(strings.LastIndexAny(s, "你")) // 12
```

### 3) IndexFunc 和 LastIndexFunc

`IndexFunc` 和 `LastIndexFunc` ,用途是查找**字符在字符串中**第一次出现的位置，其中**字符**满足 **func函数** 并返回 **true**

```
1 // 查找字符串中汉字第一次出现的位置，没有汉字返回-1
2 han := func(c rune) bool {
3     return unicode.Is(unicode.Han, c) // 汉字
4 }
5
6 // 无匹配
7 fmt.Println(strings.IndexFunc("hello golang", han)) // -1
8
9 // 匹配 "你"
10 fmt.Println(strings.LastIndexFunc("你好, Golang", han)) // 3
11 // 匹配 "好"
12 fmt.Println(strings.IndexFunc("你好, Golang", han)) // 0
```

## 8.5 字符串分割为[]string

字符串分割很常见，一般分割后会返回对应的字符串切片，strings包提供了六个三组分割函数：Fields 和 FieldsFunc、Split 和 SplitAfter、SplitN 和 SplitAfterN。

### 1) Fields 和 FieldsFunc

函数的签名如下：

```
1 func Fields(s string) []string
2 func FieldsFunc(s string, f func(rune) bool) []string
```

#### Fields

- `Fields` 用一个或多个连续的空格分隔字符串 s，返回子字符串的切片。
- 如果字符串 s 只包含空格，则返回空列表 ([]string 的长度为 0) 。
- 其中，空格的定义是 `unicode.IsSpace`

常见间隔符包括：'\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+00A0 (NBSP)

由于是用空格分隔，因此结果中不会含有空格或空子字符串，例如：

```
1 fmt.Printf("Fields are: %q\n", strings.Fields(" foo bar baz ")) //Fields are: ["foo" "bar" "baz"]
2 fmt.Printf("Fields are: %q\n", strings.Fields(" ")) //Fields are: []
```

FieldsFunc

- FieldsFunc 用这样的 Unicode 代码点 c 进行分隔：满足 f© 返回 true，该函数返回 []string。
- 如果字符串 s 中所有的代码点 (unicode code points) 都满足 f© 或者 s 是空，则 FieldsFunc 返回空 slice。
- 也就是说，我们可以通过实现一个回调函数来指定分隔字符串 s 的字符。
- 比如上面的例子，我们通过 FieldsFunc 来实现：

```
1 // FieldsFunc are: ["foo" "bar" "baz"]
2 fmt.Printf("FieldsFunc are: %q\n", strings.FieldsFunc(" foo bar baz ", unicode.IsSpace))
3
4 // FieldsFunc are: ["a" "b" "c"]
5 fmt.Printf("FieldsFunc are: %q\n", strings.FieldsFunc("a+b+c", func(r rune) bool {
6     return r=='+'
7 })))
```

- 通过查看源码发现，实际上，Fields 函数就是调用 FieldsFunc 实现的：

```
1 func Fields(s string) []string {
2     return FieldsFunc(s, unicode.IsSpace)
3 }
```

2) Split 和 SplitAfter、SplitN 和 SplitAfterN

将这四个函数放在一起讲，是因为它们都是通过一个同一个内部函数来实现的。它们的函数签名及其实现：

```
1 func Split(s, sep string) []string { return genSplit(s, sep, 0, -1) }
2 func SplitAfter(s, sep string) []string { return genSplit(s, sep, len(sep), -1) }
3 func SplitN(s, sep string, n int) []string { return genSplit(s, sep, 0, n) }
4 func SplitAfterN(s, sep string, n int) []string { return genSplit(s, sep, len(sep), n) }
```

它们都调用了 genSplit 函数。

- 这四个函数都是通过 sep 进行分割，返回 []string。
- 如果 sep 为空，相当于分成一个个的 UTF-8 字符，如 Split("abc","")，得到的是[a b c]。
- Split(s, sep) 和 SplitN(s, sep, -1) 等价；
- SplitAfter(s, sep) 和 SplitAfterN(s, sep, -1) 等价。

Split

```
1 fmt.Printf("%q\n", strings.Split("a,b,c,d", ",")) // ["a" "b" "c" "d"]
2 fmt.Printf("%q\n", strings.Split("你是一个大帅比，你就是，"你")) // [" " "是一个大帅比，" "就是"]
3 fmt.Printf("%q\n", strings.Split(" a bc", "")) // [" " "a" " " "b" "c"]
4 fmt.Printf("%q\n", strings.Split("", "xx")) // [""]
```

SplitAfter

那么，Split 和 SplitAfter 有啥区别呢？通过这两句代码的结果就知道它们的区别了：

```
1 fmt.Printf("%q\n", strings.Split("foo,bar,baz", ","))
2 fmt.Printf("%q\n", strings.SplitAfter("foo,bar,baz", ","))
```

输出：

```
1 ["foo" "bar" "baz"]
2 ["foo," "bar," "baz"]
```

也就是说，Split 会将 s 中的 sep 去掉，而 SplitAfter 会保留 sep。

SplitN和SplitAfterN

- 带 N 的方法可以通过最后一个参数 n 控制返回的结果中的 slice 中的元素个数



- 当  $n < 0$  时，返回所有的子字符串；
- 当  $n == 0$  时，返回的结果是 `nil`；
- 当  $n > 0$  时，表示返回的 `slice` 中最多只有  $n$  个元素，其中，最后一个元素不会分割，比如：

```
1 // SplintN, Split 会将 s 中的 sep 去掉
2 // n > 0, 表示返回的 slice 中最多只有 n 个元素, 其中, 最后一个元素不会分割
3 fmt.Printf("%q\n", strings.SplitN("cat,pig,bird", ",", 1)) // ["cat,pig,bird"]
4 fmt.Printf("%q\n", strings.SplitN("cat,pig,bird", ",", 2)) // ["cat" "pig","bird"]
5 fmt.Printf("%q\n", strings.SplitN("cat,pig,bird", ",", 3)) // ["cat" "pig" "bird"]
6 fmt.Printf("%q\n", strings.SplitN("cat,pig,bird", ",", 4)) // ["cat" "pig" "bird"]
7
8 // n = 0, 返回的结果是 nil
9 fmt.Printf("%q\n", strings.SplitN("cat, pig, alex", ",", 0)) // []
10
11 // n < 0, 返回所有的子字符串
12 fmt.Printf("%q\n", strings.SplitN("cat, pig, alex", ",", -1)) // ["cat" " pig" " alex"]
```

```
1 // SplintAfterN, SplitAfter 会保留 sep
2 // n > 0, 表示返回的 slice 中最多只有 n 个元素, 其中, 最后一个元素不会分割
3 fmt.Printf("%q\n", strings.SplitAfterN("cat,pig,bird", ",", 1)) // ["cat,pig,bird"]
4 fmt.Printf("%q\n", strings.SplitAfterN("cat,pig,bird", ",", 2)) // ["cat,", "pig,bird"]
5 fmt.Printf("%q\n", strings.SplitAfterN("cat,pig,bird", ",", 3)) // ["cat,", "pig,", "bird"]
6 fmt.Printf("%q\n", strings.SplitAfterN("cat,pig,bird", ",", 4)) // ["cat,", "pig,", "bird"]
7
8 // n = 0, 返回的结果是 nil
9 fmt.Printf("%q\n", strings.SplitAfterN("cat, pig, alex", ",", 0)) // []
10
11 // n < 0, 返回所有的子字符串
12 fmt.Printf("%q\n", strings.SplitAfterN("cat, pig, alex", ",", -1)) // ["cat" " pig" " alex"]
```

## 8.6 字符串是否有某个前缀或后缀

函数源码如下：

```
1 // s 中是否以 prefix 开始
2 func HasPrefix(s, prefix string) bool {
3     return len(s) >= len(prefix) && s[0:len(prefix)] == prefix
4 }
5
6 // s 中是否以 suffix 结尾
7 func HasSuffix(s, suffix string) bool {
8     return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
9 }
```

- 注：如果 `prefix` 或 `suffix` 为 `""`，返回值总是 `true`。

```
1 fmt.Println(strings.HasPrefix("Gopher", "Go")) // true
2 fmt.Println(strings.HasPrefix("Gopher", "C")) // false
3 fmt.Println(strings.HasPrefix("Gopher", "")) // true
4 fmt.Println(strings.HasSuffix("Amigo", "go")) // true
5 fmt.Println(strings.HasSuffix("Amigo", "Ami")) // false
6 fmt.Println(strings.HasSuffix("Amigo", "")) // true
```

## 8.7 字符串 JOIN 操作

`Join` 函数用法简单，将字符串数组（或 `slice`）连接起来可以通过 `Join` 实现，函数签名如下：

```
1 func Join(a []string, sep string) string
```

示例如下：

```
1 fmt.Println(strings.Join([]string{"name=xxx", "age=xx"}, "&")) // name=xxx&age=xx
```

假如没有这个库函数，我们自己实现一个，我们会这么实现：

```
1 func Join(str []string, sep string) string {
2     // 特殊情况应该做处理
3     if len(str) == 0 {
```

```

4     return ""
5 }
6 if len(str) == 1 {
7     return str[0]
8 }
9 buffer := bytes.NewBufferString(str[0])
10 for _, s := range str[1:] {
11     buffer.WriteString(sep)
12     buffer.WriteString(s)
13 }
14 return buffer.String()
15 }

```

这里，我们使用了 bytes 包的 Buffer 类型，避免大量的字符串连接操作（因为 Go 中字符串是不可变的）。我们再看一下标准库的实现：

```

1 func Join(a []string, sep string) string {
2     if len(a) == 0 {
3         return ""
4     }
5     if len(a) == 1 {
6         return a[0]
7     }
8     n := len(sep) * (len(a) - 1)
9     for i := 0; i < len(a); i++ {
10         n += len(a[i])
11     }
12
13     b := make([]byte, n)
14     bp := copy(b, a[0])
15     for _, s := range a[1:] {
16         bp += copy(b[bp:], sep)
17         bp += copy(b[bp:], s)
18     }
19     return string(b)
20 }
21

```

标准库的实现没有用 bytes 包\*\*，当然也不会简单的通过 + 号连接字符串\*\*。Go 中是不允许循环依赖的，标准库中很多时候会出现代码拷贝，而不是引入某个包。这里 Join 的实现方式挺好，我个人猜测，不直接使用 bytes 包，也是不想依赖 bytes 包（其实 bytes 中的实现也是 copy 方式）。

## 8.8 字符串重复几次

函数签名如下：

```
1 func Repeat(s string, count int) string
```

将 s 重复 count 次，如果 count 为负数或返回值长度 len(s)\*count 超出 string 上限会导致 panic，这个函数使用很简单：

```
1 fmt.Println(strings.Repeat(" ", 5) + " Go " + strings.Repeat(" ", 5)) // ***** Go *****
```

## 8.9 字符替换

函数签名如下：

```
1 func Map(mapping func(rune) rune, s string) string
```

- map 函数，将 s 的每一个字符按照 mapping 的规则做映射替换
- 如果 mapping 返回值 <0，则舍弃该字符
- 该方法只能对每一个字符做处理，但处理方式很灵活，可以方便的过滤，筛选汉字等

示例：

```

1 mapping := func(r rune) rune {
2     switch {
3     case r >= 'A' && r <= 'Z': // 大写字母转小写
4         return r + 32
5     case r >= 'a' && r <= 'z': // 小写字母不处理
6         return r

```

```

7     case unicode.Is(unicode.Han, r): // 汉字换行
8         return '\n'
9     }
10    return -1 // 过滤所有非字母、汉字的字符
11 }
12 fmt.Println(strings.Map(mapping, "Hello你#¥%.....\n ('World\n,好Hello^(&(*界gopher...)"))

```

输出：

```

1 hello
2 world
3 hello
4 gopher

```

## 8.10 字符串子串替换

- 进行字符串替换时，考虑到性能问题，能不用正则尽量别用，应该用这里的函数。
- 字符串替换的函数签名如下：

```

1 // 用 new 替换 s 中的 old，一共替换 n 个。
2 // 如果 n < 0，则不限制替换次数，即全部替换
3 func Replace(s, old, new string, n int) string
4
5 // 该函数内部直接调用了函数 Replace(s, old, new, -1)
6 func ReplaceAll(s, old, new string) string

```

示例如下：

```

1 fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))           // oinky oinky oink
2 fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))       // moo moo moo
3 fmt.Println(strings.ReplaceAll("oink oink oink", "oink", "mmm"))        // mmm mmm mmm

```

如果我们希望一次替换多个，比如我们希望替换 `This is <b>HTML</b>` 中的 `<` 和 `>` 为 `<`  和 `>` ，可以调用上面的函数两次。但标准库提供了另外的方法进行这种替换。

## 8.11 字符串大小写转换

- 大小写转换包含了 4 个相关函数
- ToLower, ToUpper 用于大小写转换
- ToLowerSpecial, ToUpperSpecial 可以转换特殊字符的大小写

函数签名如下：

```

1 func ToLower(s string) string
2 func ToLowerSpecial(c unicode.SpecialCase, s string) string
3 func ToUpper(s string) string
4 func ToUpperSpecial(c unicode.SpecialCase, s string) string

```

示例如下：

```

1 fmt.Println(strings.ToLower("HELLO WORLD"))           // hello world
2 fmt.Println(strings.ToLower("Ã Ä Å Æ"))              // ä å æ Æ
3
4 fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "壹"))          // 壹
5 fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "HELLO WORLD")) // hello world
6
7 fmt.Println(strings.ToLower("Önnek İş"))              // önnék iş
8 fmt.Println(strings.ToLowerSpecial(unicode.TurkishCase, "Önnek İş"))    // önnék iş
9
10 fmt.Println(strings.ToUpper("hello world"))           // HELLO WORLD
11 fmt.Println(strings.ToUpper("ä å æ Æ"))              // Ä Å Æ Æ
12
13 fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "—"))          // —
14 fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "hello world")) // HELLO WORLD
15
16 fmt.Println(strings.ToUpper("örnek iş"))              // ÖRNEK İŞ
17 fmt.Println(strings.ToUpperSpecial(unicode.TurkishCase, "örnek iş"))    // ÖRNEK İŞ
18

```

## 8.12 字符串标题处理

- 标题处理包含 3 个相关函数
- 其中 `Title` 会将 `s` 每个单词的**首字母大写**，不处理该单词的后续字符。
- `ToTitle` 将 `s` 的**每个字母大写**。
- `ToTitleSpecial` 将 `s` 的每个字母大写，并且会将一些特殊字母转换为其对应的特殊大写字母。

函数签名如下：

```
1 func Title(s string) string
2 func ToTitle(s string) string
3 func ToTitleSpecial(c unicode.SpecialCase, s string) string
```

举例如下：

```
1      fmt.Println(strings.Title("hElLo wOrLd"))           // HELLo wOrLd
2      fmt.Println(strings.ToTitle("hElLo wOrLd"))         // HELLO WORLD
3      fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "hElLo wOrLd")) // HELLO WORLD
4
5      fmt.Println(strings.Title("ääää öööö êêêê"))        // Ääää Öööö Êêêê
6      fmt.Println(strings.ToTitle("ääää öööö êêêê"))      // ÄÄÄÄ ÖÖÖÖ ÊÊÊÊ
7      fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "ääää öööö êêêê")) // ÄÄÄÄ ÖÖÖÖ ÊÊÊÊ
8
9      fmt.Println(strings.Title("dünyanın ilk borsa yapısı Aizonai kabul edilir"))
10     // Dünyanın İlk Borsa Yapısı Aizonai Kabul Edilir
11     fmt.Println(strings.ToTitle("dünyanın ilk borsa yapısı Aizonai kabul edilir"))
12     // DÜNYANIN ILK BORSA YAPISI AIZONAI KABUL EDILIR
13     fmt.Println(strings.ToTitleSpecial(unicode.TurkishCase, "dünyanın ilk borsa yapısı Aizonai kabul edilir"))
14     // DÜNYANIN İLK BORSA YAPISI AIZONAI KABUL EDİLİR
15
```

## 8.13 字符串修剪

函数签名：

```
1 // 将 s 左侧和右侧中匹配 cutset 中的任一字符的字符去掉
2 func Trim(s string, cutset string) string
3 // 将 s 左侧的匹配 cutset 中的任一字符的字符去掉
4 func TrimLeft(s string, cutset string) string
5 // 将 s 右侧的匹配 cutset 中的任一字符的字符去掉
6 func TrimRight(s string, cutset string) string
7 // 如果 s 的前缀为 prefix 则返回去掉前缀后的 string，否则 s 没有变化。
8 func TrimPrefix(s, prefix string) string
9 // 如果 s 的后缀为 suffix 则返回去掉后缀后的 string，否则 s 没有变化。
10 func TrimSuffix(s, suffix string) string
11 // 将 s 左侧和右侧的间隔符去掉。常见间隔符包括: '\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL)
12 func TrimSpace(s string) string
13 // 将 s 左侧和右侧的匹配 f 的字符去掉
14 func TrimFunc(s string, f func(rune) bool) string
15 // 将 s 左侧的匹配 f 的字符去掉
16 func TrimLeftFunc(s string, f func(rune) bool) string
17 // 将 s 右侧的匹配 f 的字符去掉
18 func TrimRightFunc(s string, f func(rune) bool) string
```

包含了 9 个相关函数用于修剪字符串。

举例如下：

```
1 x := "!!!@@@你好,!@#$ Gophers####$"
2 fmt.Println(strings.Trim(x, "@#$_!%^&*()_+="-"))
3 fmt.Println(strings.TrimLeft(x, "@#$_!%^&*()_+="-"))
4 fmt.Println(strings.TrimRight(x, "@#$_!%^&*()_+="-"))
5 fmt.Println(strings.TrimSpace("\t\n Hello, Gophers \n\t\r\n"))
6 fmt.Println(strings.TrimPrefix(x, "!"))
7 fmt.Println(strings.TrimSuffix(x, "$"))
8
9 f := func(r rune) bool {
10     return !unicode.Is(unicode.Han, r) // 非汉字返回 true
11 }
12 fmt.Println(strings.TrimFunc(x, f))
13 fmt.Println(strings.TrimLeftFunc(x, f))
14 fmt.Println(strings.TrimRightFunc(x, f))
```

输出如下：

```
1 | 你好,!@#$ Gophers
2 | 你好,!@#$ Gophers####$$
3 | !!!@@@你好,!@#$ Gophers
4 | Hello, Gophers
5 | !!!@@@你好,!@#$ Gophers####$$
6 | !!!@@@你好,!@#$ Gophers####$$
7 | 你好
8 | 你好,!@#$ Gophers####$$
9 | !!!@@@你好
10 |
```

### 8.14 Replacer 类型

- 这是一个结构，没有导出任何字段
- 实例化通过 `func NewReplacer(oldnew ...string) *Replacer` 函数进行，其中不定参数 **oldnew** 是 old-new 对，即进行多个替换。
- 如果 **oldnew** 长度与奇数，会导致 panic.

示例：

```
1 | r := strings.NewReplacer("<", "&lt;", ">", "&gt;")
2 | fmt.Println(r.Replace("This is <b>HTML</b>!"))
```

输出结果：

```
1 | This is &lt;b&gt;HTML&lt;/b&gt;!
```

另外，**Replacer** 还提供了另外一个方法，它在替换之后将结果写入 **io.Writer** 中。

```
1 | func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
```

### 8.15 Builder 类型

Builder 结构如下：

```
1 | type Builder struct {
2 |     addr *Builder // of receiver, to detect copies by value
3 |     buf  []byte
4 | }
```

- 该类型实现了 io 包下的 **Writer**，**ByteWriter**，**StringWriter** 等接口，可以向该对象内写入数据
- **Builder** 没有实现 **Reader** 等接口，所以该类型不可读，但提供了 **String** 方法可以获取对象内的数据。

函数签名：

```
1 | // 该方法向 b 写入一个字节
2 | func (b *Builder) WriteByte(c byte) error
3 |
4 | // WriteRune 方法向 b 写入一个字符
5 | func (b *Builder) WriteRune(r rune) (int, error)
6 |
7 | // WriteRune 方法向 b 写入字节数组 p
8 | func (b *Builder) Write(p []byte) (int, error)
9 |
10 | // WriteRune 方法向 b 写入字符串 s
11 | func (b *Builder) WriteString(s string) (int, error)
12 |
13 | // Len 方法返回 b 的数据长度。
14 | func (b *Builder) Len() int
15 |
16 | // Cap 方法返回 b 的 cap。
17 | func (b *Builder) Cap() int
18 |
19 | // Grow 方法将 b 的 cap 至少增加 n（可能会更多）。如果 n 为负数，会导致 panic。
20 | func (b *Builder) Grow(n int)
21 |
22 | // Reset 方法将 b 清空 b 的所有内容。
```

```

23 func (b *Builder) Reset()
24
25 // String 方法将 b 的数据以 string 类型返回。
26 func (b *Builder) String() string

```

- `Builder` 有 4 个与写入相关的方法，这 4 个方法的 `error` 都总是为 `nil`。
- `Builder` 的 `cap` 会自动增长，一般不需要手动调用 `Grow` 方法。
- `String` 方法可以方便的获取 `Builder` 的内容。

示例：

```

1 b := strings.Builder{}
2 _ = b.WriteByte('7')
3 n, _ := b.WriteRune('夕')
4 fmt.Println(n)
5 n, _ = b.Write([]byte("Hello, World"))
6 fmt.Println(n)
7 n, _ = b.WriteString("你好，世界")
8 fmt.Println(n)
9 fmt.Println(b.Len())
10 fmt.Println(b.Cap())
11 b.Grow(100)
12 fmt.Println(b.Len())
13 fmt.Println(b.Cap())
14 fmt.Println(b.String())
15 b.Reset()
16 fmt.Println(b.String())

```

输出结果：

```

1 3
2 12
3 15
4 31
5 32
6 31
7 164
8 7夕Hello, World你好，世界

```

## 8.16 Reader 类型

看到名字就能猜到，这是实现了 `io` 包中的接口。

它实现了：

- `io.Reader` (Read 方法)
- `io.ReaderAt` (ReadAt 方法)
- `io.Seeker` (Seek 方法)
- `io.WriterTo` (WriteTo 方法)
- `io.ByteReader` (ReadByte 方法)
- `io.ByteScanner` (ReadByte 和 UnreadByte 方法)
- `io.RuneReader` (ReadRune 方法)
- `io.RuneScanner` (ReadRune 和 UnreadRune 方法)

Reader 结构如下：

```

1 type Reader struct {
2     s      string // Reader 读取的数据来源
3     i      int // current reading index (当前读的索引位置)
4     prevRune int // index of previous rune; or < 0 (前一个读取的 rune 索引位置)
5 }

```

可见 `Reader` 结构没有导出任何字段，而是提供一个实例化方法：

```

1 func NewReader(s string) *Reader

```

该方法接收一个字符串，返回的 Reader 实例就是从该参数字符串读取数据。在后面学习了 bytes 包之后，可以知道 bytes.NewBufferString 有类似的功能，不过，如果只是为了读取，NewReader 会更高效。

其他方法不介绍了，都是之前接口的实现，有兴趣的可以看看源码实现，大部分都是根据 i、prevRune 两个属性来控制。

## 9、字符串相关补充

标准库中有四个包对字符串处理尤为重要：bytes、strings、strconv和unicode包。

- **strings包**提供了许多如字符串的查询、替换、比较、截断、拆分和合并等功能。
- **bytes包**也提供了很多类似功能的函数，但是针对和字符串有着相同结构的[]byte类型。因为字符串是只读的，因此逐步构建字符串会导致很多分配和复制。在这种情况下，使用**bytes.Buffer类型**将会更有效，稍后我们将展示。
- **strconv包**提供了布尔型、整型数、浮点数和对应**字符串的相互转换**，还提供了双引号转义相关的转换。
- **unicode包**提供了IsDigit、IsLetter、IsUpper和IsLower等类似功能，它们用于给**字符分类**。

strings 包提供了很多操作字符串的简单函数，通常一般的字符串操作需求都可以在这个包中找到。

下面简单举几个例子：

- 判断是否以某字符串打头/结尾 strings.HasPrefix(s, prefix string) bool strings.HasSuffix(s, suffix string) bool
- 字符串分割 strings.Split(s, sep string) []string
- 返回子串索引 strings.Index(s, substr string) int strings.LastIndex 最后一个匹配索引
- 字符串连接 strings.Join(a []string, sep string) string 另外可以直接使用"+"来连接两个字符串
- 字符串替换 strings.Replace(s, old, new string, n int) string
- 字符串转化为大小写 strings.ToUpper(s string) string strings.ToLower(s string) string
- 统计某个字符在字符串出现的次数 strings.Count(s, substr string) int
- 判断字符串的包含关系 strings.Contains(s, substr string) bool