Go语言 标准库 bytes包

bytes包提供了对字节切片进行读写操作的一系列函数，字节切片处理的函数比较多分为基本处理函数、比较函数、后缀检查函数、索引函数、分割函数、大小写处理函数和子切片处理函数等。

> 对于传入`[]byte参数(引用类型，函数中修改会改变外部变量)`的函数，都不会修改传入的参数，返回值要么是参数的副本，要么是参数的切片。(深入函数可以看到**大多数函数内部会创建新的字节切片对象，并返回**)

相关函数

转换

函数	说明
`func ToUpper(s []byte) []byte`	将 s 中的所有字符修改为大写格式返回。
`func ToLower(s []byte) []byte`	将 s 中的所有字符修改为小写格式返回
`func ToTitle(s []byte) []byte`	将 s 中的所有字符修改为标题格式返回
`func ToUpperSpecial(_case unicode.SpecialCase, s []byte) []byte`	使用指定的映射表将 s 中的所有字符修改为大写格式返回。
`func ToLowerSpecial(_case unicode.SpecialCase, s []byte) []byte`	使用指定的映射表将 s 中的所有字符修改为小写格式返回。

函数	说明
`func ToTitleSpecial(_case unicode.SpecialCase, s []byte) []byte`	使用指定的映射表将 s 中的所有字符修改为标题格式返回。
`func Title(s []byte) []byte`	将 s 中的所有单词的首字符修改为 Title 格式返回。（缺点：不能很好的处理以 Unicode 标点符号分隔的单词。）

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    var b = []byte("seafood") //强制类型转换

    a := bytes.ToUpper(b)
    fmt.Println(a, b)

    c := b[0:4]
    c[0] = 'A'
    fmt.Println(c, b)
}
#结果
[83 69 65 70 79 79 68] [115 101 97 102 111 111 100]
[65 101 97 102] [65 101 97 102 111 111 100]
```

> 可以看出函数不会修改原引用值类型。

比较

函数	说明
`func Compare(a, b []byte) int`	比较两个 []byte，nil 参数相当于空 []byte。a < b 返回 -1；a == b 返回 0；a > b 返回 1
`func Equal(a, b []byte) bool`	判断 a、b 是否相等，nil 参数相当于空 []byte。
`func EqualFold(s, t []byte) bool`	判断 s、t 是否相似，忽略大写、小写、标题三种格式的区别。

示例：

```
package main

import (
```

```
    "bytes"
    "fmt"
)

func main() {
    s1 := "ΦϕφkKK"
    s2 := "φΦϕKkK"

    // 看看 s1 里面是什么
    for _, c := range s1 {
        fmt.Printf("%-5x", c)
    }
    fmt.Println()
    // 看看 s2 里面是什么
    for _, c := range s2 {
        fmt.Printf("%-5x", c)
    }
    fmt.Println()
    // 看看 s1 和 s2 是否相似
    fmt.Println(bytes.EqualFold([]byte(s1), []byte(s2)))
}
#结果
3a6  3c6  3d5  20   6b   4b   212a
3d5  3a6  3c6  20   212a 6b   4b
true
```

清理

函数	说明
`func Trim(s []byte, cutset string) []byte`	去掉 s 两边包含在 cutset 中的字符（返回 s 的切片）
`func TrimLeft(s []byte, cutset string) []byte`	去掉 s 左边包含在 cutset 中的字符（返回 s 的切片）
`func TrimRight(s []byte, cutset string) []byte`	去掉 s 右边包含在 cutset 中的字符（返回 s 的切片）
`func TrimFunc(s []byte, f func(rune) bool) []byte`	去掉 s 两边符合 f函数====返回值是true还是false要求的字符（返回 s 的切片）
`func TrimLeftFunc(s []byte, f func(r rune) bool) []byte`	去掉 s左边符合 f函数====返回值是true还是false要求的字符（返回 s 的切片）
`func TrimRightFunc(s []byte, f func(r rune) bool) []byte`	去掉 s右边符合 f函数====返回值是true还是false要求的字符（返回 s 的切片）
`func TrimSpace(s []byte) []byte`	去掉 s 两边的空白（unicode.IsSpace）（返回 s 的切片）
`func TrimPrefix(s, prefix []byte) []byte`	去掉 s 的前缀 prefix（返回 s 的切片）
`func TrimSuffix(s, suffix []byte)`	去掉 s 的后缀 suffix（返回 s 的切片）

函数	说明
`[]byte`	去掉 s 的后缀 suffix（返回 s 的切片）

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    bs := [][]byte{ //[][]byte 字节切片 二维数组
        []byte("Hello World !"),
        []byte("Hello 世界! "),
        []byte("hello golang ."),
    }

    f := func(r rune) bool {
        return bytes.ContainsRune([]byte("!! . "), r) //判断r字符是否包含在"!! . "内
    }

    for _, b := range bs { //range bs  取得下标和[]byte
        fmt.Printf("去掉两边: %q\n", bytes.TrimFunc(b, f)) //去掉两边满足函数的字符
    }

    for _, b := range bs {
        fmt.Printf("去掉前缀: %q\n", bytes.TrimPrefix(b, []byte("Hello "))) //去掉
前缀
    }
}
#结果
去掉两边: "Hello World"
去掉两边: "Hello 世界"
去掉两边: "hello golang"
去掉前缀: "World !"
去掉前缀: "世界! "
去掉前缀: "hello golang ."
```

拆合

函数	说明
`func Split(s, sep []byte) [][]byte`	Split 以 sep 为分隔符将 s 切分成多个子串，结果不包含分隔符。如果 sep 为空，则将 s 切分成 Unicode 字符列表。
`func SplitN(s, sep []byte, n int) [][]byte`	SplitN 可以指定切分次数 n，超出 n 的部分将不进行切分。
`func SplitAfter(s, sep []byte) [][]byte`	功能同 Split，只不过结果包含分隔符（在各个子串尾部）。
`func SplitAfterN(s, sep []byte, n int) [][]byte`	功能同 SplitN，只不过结果包含分隔符（在各个子串尾部）。

`func Fields(s []byte) [][]byte`	以连续空白为分隔符将 s 切分成多个子串，结果不包含分隔符。
`func FieldsFunc(s []byte, f func(rune) bool) [][]byte`	以符合 f 的字符为分隔符将 s 切分成多个子串，结果不包含分隔符。
`func Join(s [][]byte, sep []byte) []byte`	以 sep 为连接符，将子串列表 s 连接成一个字节串。
`func Repeat(b []byte, count int) []bytec`	将子串 b 重复 count 次后返回。

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    b := []byte("  Hello   World !  ")
    fmt.Printf("b: %q\n", b)
    fmt.Printf("%q\n", bytes.Split(b, []byte{' '}))

    fmt.Printf("%q\n", bytes.Fields(b))

    f := func(r rune) bool {
        return bytes.ContainsRune([]byte(" !"), r)
    }
    fmt.Printf("%q\n", bytes.FieldsFunc(b, f))
}
#结果
b: "  Hello   World !  "
["" "" "Hello" "" "" "World" "!" "" ""]
["Hello" "World" "!"]
["Hello" "World"]
```

字串

函数	说明
`func HasPrefix(s, prefix []byte) bool`	判断 s 是否有前缀 prefix
`func HasSuffix(s, suffix []byte) bool`	判断 s 是否有后缀 suffix
`func Contains(b, subslice []byte) bool`	判断 b 中是否包含子串 subslice
`func ContainsRune(b []byte, r rune) bool`	判断 b 中是否包含子串 字符 r
`func ContainsAny(b []byte, chars string) bool`	判断 b 中是否包含 chars 中的任何一个字符
`func Index(s, sep []byte) int`	查找子串 sep在 s 中第一次出现的位置，找不到则返回 -1
`func IndexByte(s []byte, c byte) int`	查找子串 字节 c在 s 中第一次出现的位置，找不到则返回 -1
`func IndexRune(s []byte, r rune) int`	查找子串字符 r在 s 中第一次出现的位置，找不到则返回 -1
`func IndexAny(s []byte, chars string) int`	查找 chars 中的任何一个字符在 s 中第一次出现的位置，找不到则返回 -1。
`func IndexFunc(s []byte, f func(rune) bool) int`	查找符合 f 的字符在 s 中第一次出现的位置，找不到则返回 -1。
`func LastIndex(s, sep []byte) int`	功能同上，只不过查找最后一次出现的位置。
`func LastIndexByte(s []byte, c byte) int`	功能同上，只不过查找最后一次出现的位置。
`unc LastIndexAny(s []byte, chars string) int`	功能同上，只不过查找最后一次出现的位置。
`func LastIndexFunc(s []byte, f func(r rune) bool) int`	功能同上，只不过查找最后一次出现的位置。

函数	说明
`func Count(s, sep []byte) int`	获取 sep 在 s 中出现的次数（sep 不能重叠）。

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    b := []byte("hello golang") //字符串强转为byte切片
    sublice1 := []byte("hello")
    sublice2 := []byte("Hello")
    fmt.Println(bytes.Contains(b, sublice1)) //true
    fmt.Println(bytes.Contains(b, sublice2)) //false

    s := []byte("hellooooooooo")
    sep1 := []byte("h")
    sep2 := []byte("l")
    sep3 := []byte("o")
    fmt.Println(bytes.Count(s, sep1)) //1
    fmt.Println(bytes.Count(s, sep2)) //2
    fmt.Println(bytes.Count(s, sep3)) //9
}
#结果
true
false
1
2
9
```

替换

函数	说明
`func Replace(s, old, new []byte, n int) []byte`	将 s 中前 n 个 old 替换为 new，n < 0 则替换全部。
`func Map(mapping func(r rune) rune, s []byte) []byte`	将 s 中的字符替换为 mapping® 的返回值，如果 mapping 返回负值，则丢弃该字符。
`func Runes(s []byte) []rune`	将 s 转换为 []rune 类型返回

示例：

```
package main

import (
    "bytes"
```

```
    "fmt"
)

func main() {
    s := []byte("hello,world")
    old := []byte("o")
    news := []byte("ee")
    fmt.Println(string(bytes.Replace(s, old, news, 0)))  //hello,world
    fmt.Println(string(bytes.Replace(s, old, news, 1)))  //hellee,world
    fmt.Println(string(bytes.Replace(s, old, news, 2)))  //hellee,weerld
    fmt.Println(string(bytes.Replace(s, old, news, -1))) //hellee,weerld

    s1 := []byte("你好世界")
    r := bytes.Runes(s1)
    fmt.Println("转换前字符串的长度：", len(s1)) //12
    fmt.Println("转换后字符串的长度：", len(r))  //4
}
#结果
hello,world
hellee,world
hellee,weerld
hellee,weerld
转换前字符串的长度： 12
转换后字符串的长度： 4
```

常用函数总结

常用方法	函数	说明
Contains	`func Contains(b, subslice []byte) bool`	判断 b 中是否包含子串 subslice
Count	`func Count(s, sep []byte) int`	获取 sep 在 s 中出现的次数（sep 不能重叠）。
Repeat	`func Repeat(b []byte, count int) []bytec`	将子串 b 重复 count 次后返回。
Replace	`func Replace(s, old, new []byte, n int) []byte`	将 s 中前 n 个 old 替换为 new，n < 0 则替换全部。
Runes	`func Runes(s []byte) []rune`	将 s 转换为 []rune 类型返回
Join	`func Join(s [][]byte, sep []byte) []byte`	以 sep 为连接符，将子串列表 s 连接成一个字节串。

示例：

```
package main

import (
    "bytes"
    "fmt"
```

```
)

func main() {
    //Contains
    b := []byte("hello world") //字符串强转为byte切片
    sublice1 := []byte("hello w")
    sublice2 := []byte("Hello w")
    fmt.Println(bytes.Contains(b, sublice1)) //true
    fmt.Println(bytes.Contains(b, sublice2)) //false

    //Count
    s := []byte("hellooooooooo")
    sep1 := []byte("h")
    sep2 := []byte("l")
    sep3 := []byte("o")
    fmt.Println(bytes.Count(s, sep1)) //1
    fmt.Println(bytes.Count(s, sep2)) //2
    fmt.Println(bytes.Count(s, sep3)) //9

    //Repeat
    b = []byte("hi")
    fmt.Println(string(bytes.Repeat(b, 1))) //hi
    fmt.Println(string(bytes.Repeat(b, 3))) //hihihi

    //Replace
    s = []byte("hello,world")
    old := []byte("o")
    news := []byte("ee")
    fmt.Println(string(bytes.Replace(s, old, news, 0)))  //hello,world
    fmt.Println(string(bytes.Replace(s, old, news, 1)))  //hellee,world
    fmt.Println(string(bytes.Replace(s, old, news, 2)))  //hellee,weerld
    fmt.Println(string(bytes.Replace(s, old, news, -1))) //hellee,weerld

    //Runes
    s = []byte("你好世界")
    r := bytes.Runes(s)
    fmt.Println("转换前字符串的长度：", len(s)) //12
    fmt.Println("转换后字符串的长度：", len(r)) //4

    //Join
    s2 := [][]byte{[]byte("你好"), []byte("世界")}
    sep4 := []byte(",")
    fmt.Println(string(bytes.Join(s2, sep4))) //你好,世界
    sep5 := []byte("#")
    fmt.Println(string(bytes.Join(s2, sep5))) //你好#世界

}
#结果
true
false
1
2
9
hi
hihihi
```

```
hello,world
hellee,world
hellee,weerld
hellee,weerld
转换前字符串的长度： 12
转换后字符串的长度： 4
你好,世界
你好#世界
```

Buffer类型

缓冲区是具有读取和写入方法的可变大小的字节缓冲区。Buffer的零值是准备使用的空缓冲区。

声明Buffer的方法

方法	说明
`var b bytes.Buffer`	直接定义一个Buffer变量，不用初始化，可以直接使用
`b := new(bytes.Buffer)`	使用New返回Buffer变量
`b := bytes.NewBuffer(s []byte)`	从一个[]byte切片，构造一个Buffer
`b := bytes.NewBufferString(s string)`	从一个string变量，构造一个Buffer

往Buffer中写入数据

方法	说明
`b.Write(d []byte)`	将切片d写入Buffer数据
`b.WriteString(s string)`	将字符串s写入Buffer尾部
`b.WriteByte(c byte)`	将字符c写入Buffer尾部
`b.WriteRune(r rune)`	将一个rune类型的数据放到缓冲器的尾部
`b.WriteTo(w io.Writer)`	将Buffer中的内容输出到实现了io.Writer接口的可写入对象中

从Buffer中读取数据到指定容器

```
c := make([]byte, 8)
```

方法	说明
`b.Read(c)`	一次读取8个byte到c容器中，每次读取新的8个byte覆盖c中原来的内容
`b.ReadByte()`	读取第一个byte，b的第一个byte被拿掉，赋值给 a => `a, _ := b.ReadByte()`
`b.ReadRune()`	读取第一个rune，b的第一个rune被拿掉，赋值给 r => `r, _ := b.ReadRune()`
`b.ReadBytes(delimiter byte)`	需要一个byte作为分隔符，读的时候从缓冲器里找第一个出现的分隔符（delim），找到后，把从缓冲器头部开始到分隔符之间的所有byte进行返回，作为byte类型的slice，返回后，缓冲器也会空掉一部分
`b.ReadString(delimiter byte)`	需要一个byte作为分隔符，读的时候从缓冲器里找第一个出现的分隔符（delim），找到后，把从缓冲器头部开始到分隔符之间的所有byte进行返回，作为字符串返回，返回后，缓冲器也会空掉一部分
`b.ReadFrom(i io.Reader)`	从一个实现io.Reader接口的r，把r里的内容读到缓冲器里，n返回读的数量

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    rd := bytes.NewBufferString("Hello World!")
    buf := make([]byte, 6)
    // 获取数据切片
    b := rd.Bytes()
    // 读出一部分数据，看看切片有没有变化
    rd.Read(buf)
    fmt.Printf("%s\n", rd.String())
    fmt.Printf("%s\n\n", b)

    // 写入一部分数据，看看切片有没有变化
    rd.Write([]byte("abcdefg"))
```

```
    fmt.Printf("%s\n", rd.String())
    fmt.Printf("%s\n\n", b)

    // 再读出一部分数据，看看切片有没有变化
    rd.Read(buf)
    fmt.Printf("%s\n", rd.String())
    fmt.Printf("%s\n", b)
}
#结果
World!
Hello World!

World!abcdefg
Hello World!

abcdefg
Hello World!
```

其他相关方法

方法	说明
`func (b *Buffer) Len() int`	未读取部分的数据长度
`func (b *Buffer) Cap() int`	获取缓存的容量
`func (b *Buffer) Next(n int) []byte`	读取前 n 字节的数据并以切片形式返回，如果数据长度小于 n，则全部读取。切片只在下一次读写操作前合法。
`func (b *Buffer) Bytes() []byte`	引用未读取部分的数据切片（不移动读取位置）
`func (b *Buffer) String() string`	返回未读取部分的数据字符串（不移动读取位置）
`func (b *Buffer) Grow(n int)`	自动增加缓存容量，以保证有 n 字节的剩余空间。如果 n 小于 0 或无法增加容量则会 panic。

方法	说明
`func (b *Buffer) Truncate(n int)`	将数据长度截短到 n 字节，如果 n 小于 0 或大于 Cap 则 panic。
`func (b *Buffer) Reset()`	重设缓冲区，清空所有数据（包括初始内容）。

Reader类型

Reader实现了io.Reader, io.ReaderAt, io.WriterTo, io.Seeker, io.ByteScanner, io.RuneScanner接口，Reader是只读的、可以seek。

相关方法

方法	说明
`func NewReader(b []byte) *Reader`	将 b 包装成 bytes.Reader 对象。
`func (r *Reader) Len() int`	返回未读取部分的数据长度
`func (r *Reader) Size() int64`	返回底层数据的总长度，方便 ReadAt 使用，返回值永远不变。
`func (r *Reader) Reset(b []byte)`	将底层数据切换为 b，同时复位所有标记（读取位置等信息）。

示例：

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    data := "123456789"
    //通过[]byte创建Reader
    re := bytes.NewReader([]byte(data))
    //返回未读取部分的长度
    fmt.Println("re len : ", re.Len())
    //返回底层数据总长度
    fmt.Println("re size : ", re.Size())

    fmt.Println("---------------")

    buf := make([]byte, 2)
    for {
        //读取数据
        n, err := re.Read(buf)
        if err != nil {
```

```
            break
        }
        fmt.Println(string(buf[:n]))
    }

}
#结果
re len :  9
re size :  9
---------------
12
34
56
78
9
```

> **Reader是只读的、但是是可以seek的。**

示例:

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    data := "123456789"
    //通过[]byte创建Reader
    re := bytes.NewReader([]byte(data))

    buf := make([]byte, 2)

    re.Seek(0, 0)
    //设置偏移量
    for {
        //一个字节一个字节的读
        b, err := re.ReadByte()
        if err != nil {
            break
        }
        fmt.Println(string(b))
    }
    fmt.Println("----------------")

    re.Seek(0, 0)
    off := int64(0)
    for {
        //指定偏移量读取
        n, err := re.ReadAt(buf, off)
        if err != nil {
            break
        }
        off += int64(n)
```

```
        fmt.Println(off, string(buf[:n]))
    }

}
#结果
1
2
3
4
5
6
7
8
9
----------------
2 12
4 34
6 56
8 78
```