Go语言 标准库 strings包

strings标准库包主要涉及字符串的基本操作。

求长

len

由于string类型可以看成是一种特殊的slice类型，因此获取长度可以用内置的函数len；同时支持 切片操作，因此，子串获取很容易。

内置函数 `len()` 返回字符串中的字符数。当你需要强制最小或最大密码长度，或将较大的字符串截断以在特定限制内用作缩写时，此功能很有用。

`len()` 函数将计算由双引号绑定的任何字符——包括字母、数字、空白字符和符号。

求子串

是否存在某个字符或子串

Contains

是否存在某个字符或子串，子串substr在s中，返回true，判断字符串 s 是否包含子串 substr。

```
func Contains(s, substr string) bool
```

ContainsAny

chars中任何一个Unicode代码点在s中，返回true，判断字符串 s 是否包含 chars 中的任一字符。如果 chars 为空串，直接返回 false。

```
func ContainsAny(s, chars string) bool
```

ContainsRune

Unicode代码点r在s中，返回true，判断字符串 s 是否包含字符 r。

```
func ContainsRune(s string, r rune) bool
```

字符或子串在字符串中出现的位置

Index

返回子串 sep 在字符串 s 中第一次出现的索引下标，不存在返回 -1。

```
func Index(s, sep string) int
```

IndexAny

返回 chars 中任一字符在字符串 s 中第一次出现的索引下标，不存在或者 chars 为空串返回 -1

```
func IndexAny(s, chars string) int
```

IndexFunc

返回字符串 s 中第一次满足函数 f 的索引下标（该处的字符 r 满足 f(r) == true），不存在返回 -1。

```
func IndexFunc(s string, f func(rune) bool) int   //rune类型是int32别名，UTF-8字符格式编码。
```

IndexByte

返回字节 c 在字符串 s 中第一次出现的索引下标，不存在返回 -1。

```
func IndexByte(s string, c byte) int   //byte是字节类型
```

IndexRune

返回字符 r 在字符串 s 中第一次出现的索引下标，不存在返回 -1。

```
func IndexRune(s string, r rune) int
```

LastIndex

返回子串 sep 在字符串 s 中最后一次出现的索引下标，不存在返回 -1。

```
func LastIndex(s, sep string) int
```

LastIndexAny

返回 chars 中任一字符在字符串 s 中最后一次出现的索引下标，不存在或者 chars 为空串返回 -1。

```
func LastIndexAny(s, chars string) int
```

LastIndexFunc

返回字符串 s 中最后一次满足函数 f 的索引下标（该处的字符 r 满足 f(r) == true），不存在返回 -1。

```
func LastIndexFunc(s string, f func(rune) bool) int
```

LastIndexByte

返回字节 c 在字符串 s 中最后一次出现的索引下标，不存在返回 -1。

```
func LastIndexByte(s string, c byte) int
```

子串出现次数

Count

子串在s字符串中出现的次数，计算字符串 s 中不重叠的子串 sep 的数目。如果子串 sep 为空，直接返回 len(s) + 1，内部使用的 Rabin-Karp 算法进行字符串模式匹配。

```
func Count(s, sep string) int   //子串在s字符串中出现的次数
```

- 特别说明一下的是当 sep 为空时，Count 的返回值是：utf8.RuneCountInString(s) + 1
- Count 是计算子串在字符串中出现的无重叠的次数

字符串是否有某个前缀或后缀

HasPrefix

是否有指定前缀。判断字符串 s 是否有前缀子串 prefix。如果 prefix 为 ""，总是返回 true

```
func HasPrefix(s, prefix string) bool
```

HasSuffix

是否有指定后缀。判断字符串 s 是否有后缀子串 suffix。如果 suffix 为 ""，总是返回 true。

```
func HasSuffix(s, suffix string) bool
```

golang语言中的rune类型

Rune 是int32 的别名。用UTF-8 进行编码。这个类型在什么时候使用呢？例如需要遍历字符串中的字符。可以循环每个字节（仅在使用US ASCII 编码字符串时与字符等价，而它们在Go中不存在！）。因此为了获得实际的字符，需要使用rune类型。在UTF-8 世界的字符有时被称作runes。通常，当人们讨论字符时，多数是指8 位字符。UTF-8 字符可能会有32 位，称作rune。

例如：

```
 s:="Go编程" fmt.Println(len(s)) 输出结果应该是8    因为中文字符是用3个字节存的。
```

```
len(string(rune('编'))) 的结果是3
```

如果想要获得我们想要的情况的话，需要先转换为rune切片再使用内置的len函数

```
fmt.Println(len([]rune(s))) 结果就是 4 了。
```

所以用string存储 unicode 的话，如果有中文，按下标是访问不到的，因为你只能得到一个 byte 。 要想访问中文的话，还是要用rune切片，这样就能按下表访问。

字符串转换

ToUpper

原始字符串的所有字母都转换为大写字母。因为字符串是不可变的数据类型，所以返回的字符串将是一个新字符串。字符串中的任何非字母字符都不会更改。返回将字符串 s 的所有字母都转为对应的大写的新字符串

```
func ToUpper(s string) string
```

ToLower

原始字符串的所有字母都转换为小写字母。因为字符串是不可变的数据类型，所以返回的字符串将是一个新字符串。字符串中的任何非字母字符都不会更改。返回将字符串 s 的所有字母都转为对应的小写的新字符串

```
func ToLower(s string) string
```

ToTitle

将s中的所有字符修改为其Title格式，大部分字符的Title格式就是Upper格式，只有少数字符的Title格式是特殊字符。

```
func ToTitle(s string) string
```

ToUpperSpecial

将 s 中的所有字符修改为其大写格式，优先使用 _case 中的规则进行转换。

```
func ToUpperSpecial(_case unicode.SpecialCase, s string) string
```

ToLowerSpecial

将 s 中的所有字符修改为其小写格式，优先使用 _case 中的规则进行转换。

```
func ToLowerSpecial(_case unicode.SpecialCase, s string) string
```

ToTitleSpecial

将 s 中的所有字符修改为其 Title 格式，优先使用 _case 中的规则进行转换。

```
func ToTitleSpecial(_case unicode.SpecialCase, s string) string
```

Title

Title 将 s 中的所有单词的首字母修改为其 Title 格式，Title 规则不能正确处理 Unicode 标点符号。

```
func Title(s string) string
```

比较

Compare

按照字典序比较两个字符串大小，a = b 返回 0，a < b 返回 -1，a > b 返回 1，不推荐使用这个函数，直接使用 =、>、< 比较会更加直观。

```
func Compare(a, b string) int  //返回不相等-1或者  相等0
```

EqualFold

判断两个字符串（不区分大小写）是否相同。

```
func EqualFold(s, t string) bool
```

清理

Trim

返回将字符串 s 前后端所有 cutset 包含的字符都去除的新字符串。

```
func Trim(s string, cutset string) string
```

TrimLeft

返回将字符串 s 前端所有 cutset 包含的字符都去除的新字符串。

```
func TrimLeft(s string, cutset string) string
```

TrimRight

返回将字符串 s 后端所有 cutset 包含的字符都去除的新字符串。

```
func TrimRight(s string, cutset string) string
```

TrimFunc

返回将字符串 s 前后端字符 r（满足 f(r) = true）都去除的新字符串。

```
func TrimFunc(s string, f func(rune) bool) string
```

TrimLeftFunc

返回将字符串 s 前端字符 r（满足 f(r) = true）都去除的新字符串。

```
func TrimLeftFunc(s string, f func(rune) bool) string
```

TrimRightFunc

返回将字符串 s 后端字符 r（满足 f(r) = true）都去除的新字符串。

```
func TrimRightFunc(s string, f func(rune) bool) string
```

TrimSpace

返回将字符串 s 前后端所有空白字符（unicode.IsSpace 指定）都去除的新字符串。

```
func TrimSpace(s string) string //字符串前后空格
fmt.Println(strings.TrimSpace(" \t\n a lone gopher \n\t\r\n"))  输出：a lone
gopher
```

TrimPrefix

返回将字符串 s 可能的前缀子串 prefix 去除的新字符串。

```
func TrimPrefix(s, prefix string) string
```

TrimSuffix

返回将字符串 s 可能的后缀子串 suffix 去除的新字符串。

```
func TrimSuffix(s, suffix string) string
```

拆合函数

Fields

返回将字符串按照空白（unicode.IsSpace确定，可以是一到多个连续的空白字符）分割的多个字符串。

如果字符串全部是空白或者是空字符串的话，会返回空切片。

```
func Fields(s string) []string
```

FieldsFunc

返回将字符串按照分隔符 r（满足 f(r) == true）分割的多个字符串。

如果字符串全部是分隔符或者是空字符串的话，会返回空切片。

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

Split

用去除每一个 sep 的方式对字符串 s 进行分割，会分割到结尾，返回分割出的所有子串组成的切片。每一个 sep 都会进行一次分割，即使两个 sep 相邻，也会进行两次分割。如果 sep 空串，Split 会将字符串 s 分割为一个字符一个子串。

```
func Split(s, sep string) []string { return genSplit(s, sep, 0, -1) }
```

SplitAfter

用在每一个 sep 后面切割的方式对字符串 s 进行分割，会分割到结尾，返回分割出的所有子串组成的切片，每一个 sep 都会进行一次分割，即使两个 sep 相邻，也会进行两次分割。

```
func SplitAfter(s, sep string) []string { return genSplit(s, sep, len(sep), -1)
}
```

SplitN

类似 Split，但是参数 n 决定分割后的切片大小。n < 0：等同 Split(s, sep)；n == 0：返回空切片；n > 0：最多分割出 n 个子串，最后一个子串包含未进行切割的部分。如果 sep 空串，Split 会将字符串 s 分割为一个字符一个子串。

```
func SplitN(s, sep string, n int) []string { return genSplit(s, sep, 0, n) }
```

SplitAfterN

类似 SplitAfter，但是参数 n 决定分割后的切片大小。n < 0：等同 SplitAfter(s, sep)；n == 0：返回空切片；n > 0：最多分割出 n 个子串，最后一个子串包含未进行切割的部分。

```
func SplitAfterN(s, sep string, n int) []string { return genSplit(s, sep,
len(sep), n) }
```

Join

连接字符串，将一系列字符串连接为一个新的字符串，之间用 sep 来分隔。

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

Repeat

重复串联

返回 count 个字符串 s 串联后的新字符串，count 不能传负数。

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

替换

Replace

返回将字符串 s 中前 n 个不重叠子串 old 都替换为子串 new 的新字符串，如果 n < 0 会替换所有子串 old

```
func Replace(s, old, new string, n int) string
```

ReplaceAll

返回将字符串 s 中所有不重叠子串 old 都替换为子串 new 的新字符串，相当于使用 Replace 时n < 0

```
func ReplaceAll(s, old, new string) string
```

Map

字符映射替换，

返回对字符串 s 中每一个字符 r 执行 mapping(r) 操作后的新字符串。

```
func Map(mapping func(rune) rune, s string) string   //满足函数实现的进行替换
```

复制

clone

通过 copy 函数对原始字符串进行复制，得到一份新的 []byte 数据。
通过 *(string)(unsafe.Pointer(&b)) 进行指针操作，实现 byte 到 string 的零内存复制转换。

```
func Clone(s string) string {
 if len(s) == 0 {
  return ""
 }
 b := make([]byte, len(s))
 copy(b, s)
 return *(*string)(unsafe.Pointer(&b))
}
```

Replacer 类型

`strings.Replacer` 类型用于进行一系列字符串的替换，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。

```
type Replacer struct {
    // 内含隐藏或非导出字段
}

// 使用提供的多组 old、new 字符串对创建并返回一个 *Replacer。替换是依次进行的，匹配时不会重叠
func NewReplacer(oldnew ...string) *Replacer

// 返回 s 的所有替换进行完后的拷贝
func (r *Replacer) Replace(s string) string

// 向 w 中写入 s 的所有替换进行完后的拷贝
func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
```

示例：

```
package main

import (
    "fmt"
    "os"
    "strings"
)
```

```
func main() {
    // 将所有的 "<" 替换为 "&lt;"、">" 替换为 "&gt;"
    r := strings.NewReplacer("<", "&lt;", ">", "&gt;")
    fmt.Println(r.Replace("This is <b>HTML</b>!")) // This is
&lt;b&gt;HTML&lt;/b&gt;!

    // 向标准输出中写入替换后的字符串
    r.WriteString(os.Stdout, "This is <b>HTML</b>!") // This is
&lt;b&gt;HTML&lt;/b&gt;!
}
```

Reader 类型

`strings.Reader` 类型通过从一个字符串读取数据，实现了 `io.Reader`、`io.Seeker`、`io.ReaderAt`、`io.WriterTo`、`io.ByteScanner`、`io.RuneScanner` 接口。

```
type Reader struct {
    s        string // 要读取的字符串
    i        int64  // 当前读取的偏移量位置，从 i 处开始读取数据
    prevRune int    // 读取的前一个字符的偏移量位置，小于 0 表示之前未读取字符
}

// 创建一个从 s 读取数据的 Reader。本函数类似 bytes.NewBufferString，但是更有效率，且为只读的
func NewReader(s string) *Reader

// 返回 r 包含的字符串还没有被读取的部分的长度
func (r *Reader) Len() int

// 从 r 中读取最多 len(b) 字节数据并写入 b，返回读取的字节数和可能遇到的任何错误。如果无可读数据返回 0 个字节，且返回值 err 为 io.EOF
func (r *Reader) Read(b []byte) (n int, err error)

// 将索引位置 off 之后的所有数据写入到 b 中，返回读取的字节数和读取过程中遇到的错误
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)

// 读取并返回一个字节。如果没有可用的数据，会返回错误
func (r *Reader) ReadByte() (b byte, err error)

// 撤消前一次的 ReadByte 操作，即偏移量向前移动一个字节，UnreadByte 操作前必须要有 ReadByte 操作
func (r *Reader) UnreadByte() error

// 读取并返回一个字符写入到 ch 中，size 为 ch 的字节大小，err 返回读取过程遇到的错误
func (r *Reader) ReadRune() (ch rune, size int, err error)

// 撤销前一次的 ReadRune 操作，即偏移量向前移动一个字符，UnreadRune 操作前必须要有 ReadRune 操作
func (r *Reader) UnreadRune() error

// 设置下一次读的位置。offset 为相对偏移量，而 whence 决定相对位置：0 为相对文件开头，1 为相对当前位置，2 为相对文件结尾
// 返回新的偏移量（相对开头）和可能的错误
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

```
// 将底层数据切换为 s，同时复位所有标记（读取位置等信息）
func (r *Reader) Reset(s string)

// 从 r 中读取数据写入接口 w 中，返回读取的字节数和可能遇到的任何错误
func (r *Reader) WriteTo(w io.Writer) (n int64, err error)
```

示例：

```
package main

import (
    "fmt"
    "log"
    "os"
    "strings"
)

func main() {
    s := "hello world!!!"
    r := strings.NewReader(s)

    // 返回未读的数据长度
    fmt.Println(r.Len()) // 14

    // 读取三个字节到字节切片
    byteSlice := make([]byte, 3)
    numBytesRead, err := r.Read(byteSlice)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Read %d bytes: %s\n", numBytesRead, byteSlice) // Read 3 bytes:
hel

    // 读取一个字节，如果读取不成功会返回 Error
    myByte, err := r.ReadByte()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Read 1 byte: %c\n", myByte) // Read 1 byte: l

    // 撤消前一次的 ReadByte 操作，偏移量会向前移动一个字节
    err = r.UnreadByte()
    if err != nil {
        log.Fatal(err)
    }

    // 返回未读的数据长度
    fmt.Println(r.Len()) // 11

    // 将剩下未读的数据写入标准输出中
    num, err := r.WriteTo(os.Stdout) // lo world!!!
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Printf("Read %d bytes\n", num) // Read 11 bytes
}
```

Builder 类型

在字符串拼接时可以通过`strings.Builder`的写入方法来高效构建字符串，它最小化了内存拷贝。

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

// 预分配内存
func (b *Builder) Grow(n int)

// 返回当前 b 底层用于存储数据的 []byte 切片的长度和容量
func (b *Builder) Len() int
func (b *Builder) Cap() int

// 将当前 b 清空
func (b *Builder) Reset()

// 往当前 b 中写入不同类型的数据，返回写入数据的字节大小和发生的错误
func (b *Builder) Write(p []byte) (int, error)
func (b *Builder) WriteByte(c byte) error
func (b *Builder) WriteRune(r rune) (int, error)
func (b *Builder) WriteString(s string) (int, error)

// 将当前 b 中存储的数据转换为字符串输出
func (b *Builder) String() string
```

示例：

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var b strings.Builder
    // 四种写入方法
    b.Write([]byte("hello"))
    b.WriteByte(' ')
    b.WriteRune('您')
    b.WriteString("好")

    for i := 1; i <= 3; i++ {
        // strings.Builder 实现了 io.Writer 接口
        fmt.Fprintf(&b, "%d...", i)
    }
    fmt.Println(b.String()) // hello 您好1...2...3...
    fmt.Println(b.Len())    // 24
```

```
    fmt.Println(b.Cap())    // 48
}
```