Go语言 标准库 regexp包

正则表达式是一种进行模式匹配和文本操纵的复杂而又强大的工具。虽然正则表达式比纯粹的文本匹配效率低，但是它却更灵活，按照它的语法规则，根据需求构造出的正则表达式能够从原始文本中筛选出几乎任何你想要得到的字符组合。

Go语言通过 regexp 包为正则表达式提供了官方支持，其采用 RE2 语法，Go语言和 Perl、Python等语言的正则基本一致。

正则表达式语法规则

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）构成的文字序列，可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

字符

语法	说明	表达式示例	匹配结果
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符， 在 DOTALL 模式中也能匹配换行符	a.c	abc
\	转义字符，使后一个字符改变原来的意思； 如果字符串中有字符 * 需要匹配，可以使用 `\*` 或者字符集 [*]。	`a\.c` `a\\c`	a.c a\c

语法	说明	表达式示例	匹配结果
[...]	字符集（字符类），对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如 [abc] 或 [a-c]， 第一个字符如果是 ^ 则表示取反，如 `[^abc]` 表示除了abc之外的其他字符。	a[bcd]e	abe 或 ace 或 ade
\d	数字：[0-9]	a\dc	a1c
\D	非数字：`[^\d]`	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：`[^\s]`	a\Sc	abc
\w	单词字符：[A-Za-z0-9]	a\wc	abc
\W	非单词字符：`[^\w]`	a\Wc	a c

数量词（用在字符或 (...) 之后）

语法	说明	表达式示例	匹配结果
*	匹配前一个字符 0 或无限次	abc*	ab 或 abccc
+	匹配前一个字符 1 次或无限次	abc+	abc 或 abccc
?	匹配前一个字符 0 次或 1 次	abc?	ab 或 abc
{m}	匹配前一个字符 m 次	ab{2}c	abbc
{m,n}	匹配前一个字符 m 至 n 次，m 和 n 可以省略，若省略 m，则匹配 0 至 n 次； 若省略 n，则匹配 m 至无限次	ab{1,2}c	abc 或 abbc

边界匹配

语法	说明	表达式示例	匹配结果
^	匹配字符串开头，在多行模式中匹配每一行的开头	^abc	abc
$	匹配字符串末尾，在多行模式中匹配每一行的末尾	abc$	abc
\A	仅匹配字符串开头	\Aabc	abc
\Z	仅匹配字符串末尾	abc\Z	abc
\b	匹配 \w 和 \W 之间	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc

逻辑、分组

语法	说明	表达式示例	匹配结果
`|`	| 代表左右表达式任意匹配一个，优先匹配左边的表达式	abc|def	abc 或 def
`(...)`	括起来的表达式将作为分组，分组将作为一个整体，可以后接数量词	(abc){2}	abcabc
`(?P<name>...)`	分组，功能与 (...) 相同，但会指定一个额外的别名	`(?P<id>abc){2}`	abcabc
`\<number>`	引用编号为 `<number>` 的分组匹配到的字符串	(\d)abc\1	1abe1 或 5abc5
(?P=name)	引用别名为 `<name>` 的分组匹配到的字符串	`(?P<id>\d)abc(?P=id)`	1abe1 或 5abc5

特殊构造（不作为分组）

语法	说明	表达式示例	匹配结果
(?:...)	(...) 的不分组版本，用于使用 "|" 或后接数量词	(?:abc){2}	abcabc
(?iLmsux)	iLmsux 中的每个字符代表一种匹配模式，只能用在正则表达式的开头，可选多个	(?i)abc	AbC
(?#...)	# 后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配	a(?=\d)	后面是数字的 a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配	a(?!\d)	后面不是数字

(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配	a(?!\d)	是数字的a
语法	**说明**	**表达式示例**	**匹配结果**
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配	(?<!\d)a	前面不是数字的a

Regexp 包的使用

Go在处理正则表达式时主要使用regexp包，包中实现了正则表达式的查找、替换和模式匹配功能。

入门实例：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    text := "Hello 世界!123 Go."

    // 查找连续的小写字母
    reg := regexp.MustCompile(`[a-z]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["ello" "o"]

    // 查找连续的非小写字母
    reg = regexp.MustCompile(`[^a-z]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["H" " 世界! 123 G" "."]

    // 查找连续的单词字母
    reg = regexp.MustCompile(`[\w]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["Hello" "123" "Go"]

    // 查找连续的非单词字母、非空白字符
    reg = regexp.MustCompile(`[^\w\s]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["世界!" "."]

    // 查找连续的大写字母
    reg = regexp.MustCompile(`[[:upper:]]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["H" "G"]

    // 查找连续的非 ASCII 字符
    reg = regexp.MustCompile(`[[:^ascii:]]+`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
```

```
// ["世界! "]

// 查找连续的标点符号
reg = regexp.MustCompile(`[\pP]+`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["! " "."]

// 查找连续的非标点符号字符
reg = regexp.MustCompile(`[\PP]+`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello 世界" "123 Go"]

// 查找连续的汉字
reg = regexp.MustCompile(`[\p{Han}]+`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["世界"]

// 查找连续的非汉字字符
reg = regexp.MustCompile(`[\P{Han}]+`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello " "! 123 Go."]

// 查找 Hello 或 Go
reg = regexp.MustCompile(`Hello|Go`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello" "Go"]

// 查找行首以 H 开头，以空格结尾的字符串
reg = regexp.MustCompile(`^H.*\s`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello 世界! 123 "]

// 查找行首以 H 开头，以空白结尾的字符串（非贪婪模式）
reg = regexp.MustCompile(`(?U)^H.*\s`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello "]

// 查找以 hello 开头（忽略大小写），以 Go 结尾的字符串
reg = regexp.MustCompile(`(?i:^hello).*Go`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello 世界! 123 Go"]

// 查找 Go.
reg = regexp.MustCompile(`\QGo.\E`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Go."]

// 查找从行首开始，以空格结尾的字符串（非贪婪模式）
reg = regexp.MustCompile(`(?U)^.* `)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
// ["Hello "]

// 查找以空格开头，到行尾结束，中间不包含空格字符串
reg = regexp.MustCompile(` [^ ]*$`)
fmt.Printf("%q\n", reg.FindAllString(text, -1))
```

```
    // [" Go."]

    // 查找“单词边界”之间的字符串
    reg = regexp.MustCompile(`(?U)\b.+\b`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["Hello" " 世界！" "123" " " "Go"]

    // 查找连续 1 次到 4 次的非空格字符，并以 o 结尾的字符串
    reg = regexp.MustCompile(`[^ ]{1,4}o`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["Hello" "Go"]

    // 查找 Hello 或 Go
    reg = regexp.MustCompile(`(?:Hell|G)o`)
    fmt.Printf("%q\n", reg.FindAllString(text, -1))
    // ["Hello" "Go"]

    // 查找 Hello 或 Go，替换为 Hellooo、Gooo
    reg = regexp.MustCompile(`(?:Hell|G)o`)
    fmt.Printf("%q\n", reg.ReplaceAllString(text, "${n}ooo"))
    // "Hellooo 世界！123 Gooo."

    // 交换 Hello 和 Go
    reg = regexp.MustCompile(`(Hello)(.*)(Go)`)
    fmt.Printf("%q\n", reg.ReplaceAllString(text, "$3$2$1"))
    // "Go 世界！123 Hello."

    // 特殊字符的查找
    reg =
regexp.MustCompile(`[\\f\\t\\n\\r\\v\\123\\x7F\\x{10FFFF}\\^\\$\\.\\*\\+\\?\\
{\\}\\(\\)\\[\\]\\|]`)
    fmt.Printf("%q\n", reg.ReplaceAllString("\f\t\n\r\v\123\x7F\U0010FFFF\\^$.*+?
{}()[]|", "-"))
}
```

匹配指定类型的字符串：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    buf := "abc azc a7c aac 888 a9c  tac"

    //解析正则表达式，如果成功返回解释器
    reg1 := regexp.MustCompile(`a.c`)
    if reg1 == nil {
        fmt.Println("regexp err")
        return
    }
```

```
    //根据规则提取关键信息
    result1 := reg1.FindAllStringSubmatch(buf, -1)
    fmt.Println("result1 = ", result1)
}
```

匹配 a 和 c 中间包含一个数字的字符串：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    buf := "abc azc a7c aac 888 a9c  tac"

    //解析正则表达式，如果成功返回解释器
    reg1 := regexp.MustCompile(`a[0-9]c`)

    if reg1 == nil { //解释失败，返回nil
        fmt.Println("regexp err")
        return
    }

    //根据规则提取关键信息
    result1 := reg1.FindAllStringSubmatch(buf, -1)
    fmt.Println("result1 = ", result1)
}
```

使用 \d 来匹配 a 和 c 中间包含一个数字的字符串：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {

    buf := "abc azc a7c aac 888 a9c  tac"

    //解析正则表达式，如果成功返回解释器
    reg1 := regexp.MustCompile(`a\dc`)
    if reg1 == nil { //解释失败，返回nil
        fmt.Println("regexp err")
        return
    }

    //根据规则提取关键信息
    result1 := reg1.FindAllStringSubmatch(buf, -1)
```

```
    fmt.Println("result1 = ", result1)
}
```

匹配字符串中的小数：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    buf := "43.14 567 agsdg 1.23 7. 8.9 1sdljgl 6.66 7.8   "

    //解释正则表达式
    reg := regexp.MustCompile(`\d+\.\d+`)
    if reg == nil {
        fmt.Println("MustCompile err")
        return
    }

    //提取关键信息
    //result := reg.FindAllString(buf, -1)
    result := reg.FindAllStringSubmatch(buf, -1)
    fmt.Println("result = ", result)
}
```

匹配 div 标签中的内容：

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    // 原生字符串
    buf := `

<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <title>C语言中文网 | Go语言入门教程</title>
</head>
<body>
    <div>Go语言简介</div>
    <div>Go语言基本语法
    Go语言变量的声明
    Go语言教程简明版
    </div>
    <div>Go语言容器</div>
    <div>Go语言函数</div>
```

```
</body>
</html>
    `

    //解释正则表达式
    reg := regexp.MustCompile(`<div>(?s:(.*?))</div>`)
    if reg == nil {
        fmt.Println("MustCompile err")
        return
    }

    //提取关键信息
    result := reg.FindAllStringSubmatch(buf, -1)

    //过滤<></>
    for _, text := range result {
        fmt.Println("text[1] = ", text[1])
    }
}
```

通过 Compile 方法返回一个 Regexp 对象，实现匹配，查找，替换相关的功能：

```
package main
import (
    "fmt"
    "regexp"
    "strconv"
)
func main() {
    //目标字符串
    searchIn := "John: 2578.34 William: 4567.23 Steve: 5632.18"
    pat := "[0-9]+.[0-9]+"          //正则

    f := func(s string) string{
        v, _ := strconv.ParseFloat(s, 32)
        return strconv.FormatFloat(v * 2, 'f', 2, 32)
    }
    if ok, _ := regexp.Match(pat, []byte(searchIn)); ok {
        fmt.Println("Match Found!")
    }
    re, _ := regexp.Compile(pat)
    //将匹配到的部分替换为 "##.#"
    str := re.ReplaceAllString(searchIn, "##.#")
    fmt.Println(str)
    //参数为函数时
    str2 := re.ReplaceAllStringFunc(searchIn, f)
    fmt.Println(str2)
}
```

Regexp 包的函数和方法

官方文档：

```
go doc regexp/syntax
```

Match

```
func Match(pattern string, b []byte) (matched bool, err error)
```

判断在 b 中能否找到正则表达式 pattern 所匹配的子串。

pattern：要查找的正则表达式

b：要在其中进行查找的 []byte

matched：返回是否找到匹配项

err：返回查找过程中遇到的任何错误

此函数通过调用 Regexp 的方法实现

```
func main() {
    fmt.Println(regexp.Match("H.* ", []byte("Hello World!")))
}

// true
```

MatchReader

```
func MatchReader(pattern string, r io.RuneReader) (matched bool, err error)
```

判断在 r 中能否找到正则表达式 pattern 所匹配的子串

pattern：要查找的正则表达式

r：要在其中进行查找的 RuneReader 接口

matched：返回是否找到匹配项

```
func main() {
    r := bytes.NewReader([]byte("Hello World!"))
    fmt.Println(regexp.MatchReader("H.* ", r))
// true
}
```

MatchString函数

判断在 s 中能否找到正则表达式 pattern 所匹配的子串

MarchString函数接收一个要查找的正则表达式和目标字符串，并根据匹配结果返回true或false，函数定义如下：

```
func MatchString(pattern string, s string) (matched bool, err error)
```

pattern：要查找的正则表达式

s：要在其中进行查找的字符串

对于目标字符串“hello world”，我们通过MatchString函数匹配其中的“hello”字符串，并返回匹配结果：

```
package main

import (
   "fmt"
   "regexp"
)

func main() {

   targetString := "hello world"
   matchString := "hello"

   match, err := regexp.MatchString(matchString, targetString)

   if err != nil {
      fmt.Println(err)
   }
   fmt.Println(match)
}
#结果
true
```

QuoteMeta

```
func QuoteMeta(s string) string
```

QuoteMeta 将字符串 s 中的“特殊字符”转换为其“转义格式”，例如，QuoteMeta(`[foo]`)返回``。 特殊字符有：.+*?()|[]{}^$，这些字符用于实现正则语法，所以当作普通字符使用时需要转换。

```
func main() {
    fmt.Println(regexp.QuoteMeta("(?P:Hello) [a-z]"))
}
```

Compile

```
func Compile(expr string) (*Regexp, error)
```

Compile 用来解析正则表达式 expr 是否合法，如果合法，则返回一个 Regexp 对象，Regexp 对象可以在任意文本上执行需要的操作。

```
func main() {
    reg, err := regexp.Compile(`\w+`)
    fmt.Printf("%q,%v\n", reg.FindString("Hello World!"), err)
}
```

CompilePOSIX

```
func CompilePOSIX(expr string) (*Regexp, error)
```

CompilePOSIX 的作用和 Compile 一样，不同的是，CompilePOSIX 使用 POSIX 语法，同时，它采用最左最长方式搜索，而 Compile 采用最左最短方式搜索，POSIX 语法不支持 Perl 的语法格式：\d、\D、\s、\S、\w、\W。

```
func main() {
    reg, err := regexp.CompilePOSIX(`[[:word:]]+`)
    fmt.Printf("%q,%v\n", reg.FindString("Hello World!"), err)
}
```

MustCompile

```
func MustCompile(str string) *Regexp
```

MustCompile 的作用和 Compile 一样，不同的是，当正则表达式 str 不合法时，MustCompile 会抛出异常，而 Compile 仅返回一个 error 值。

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindString("Hello World!"))
}
```

MustCompilePOSIX

```
func MustCompilePOSIX(str string) *Regexp
```

MustCompilePOSIX 的作用和 CompilePOSIX 一样，不同的是，当正则表达式 str 不合法时，MustCompilePOSIX 会抛出异常，而 CompilePOSIX 仅返回一个 error 值。

```
func main() {
    reg := regexp.MustCompilePOSIX(`[[:word:]].+ `)
    fmt.Printf("%q\n", reg.FindString("Hello World!"))
}
```

Find

```
func (re *Regexp) Find(b []byte) []byte
```

在 b 中查找 re 中编译好的正则表达式，并返回第一个匹配的内容

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Printf("%q", reg.Find([]byte("Hello World!")))
}
```

FindString

```
func (re *Regexp) FindString(s string) string
```

在 s 中查找 re 中编译好的正则表达式，并返回第一个匹配的内容

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindString("Hello World!"))
}
```

FindAll

```
func (re *Regexp) FindAll(b []byte, n int) [][]byte
```

在 b 中查找 re 中编译好的正则表达式，并返回所有匹配的内容，{{匹配项}, {匹配项}, ...}，只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项。

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Printf("%q", reg.FindAll([]byte("Hello World!"), -1))
}
```

FindAllString

```
func (re *Regexp) FindAllString(s string, n int) []string
```

在 s 中查找 re 中编译好的正则表达式，并返回所有匹配的内容，{匹配项, 匹配项, ...}，只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项。

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Printf("%q", reg.FindAllString("Hello World!", -1))
}
```

FindIndex

```
func (re *Regexp) FindIndex(b []byte) (loc []int)
```

在 b 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置，{起始位置, 结束位置}

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindIndex([]byte("Hello World!")))
}
```

FindStringIndex函数

在 s 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置

FindStringIndex函数接收一个目标字符串，并返回第一个匹配的起始位置和结束位置，函数定义如下：

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

由于FindStringIndex函数是Regexp结构体的成员函数，需要对正则表达式进行编译，编译成功后方能使用。

通常使用Compile或MustCompile函数进行编译。

1. Compile函数：若正则表达式未通过编译，则返回错误。
2. MustCompile函数：若正则表达式未通过编译，则引发panic。

对于目标字符串“hello world”，我们通过FindStringIndex函数匹配其中的“hello”字符串对应的起始和结束位置：

```
package main

import (
   "fmt"
   "regexp"
)

func main() {

   targetString := "hello world"
   re := regexp.MustCompile(`(\w)+`)

   res := re.FindStringIndex(targetString)
   fmt.Println(res)
}
```

FindReaderIndex

```
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
```

在 r 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置，{起始位置, 结束位置}

```
func main() {
    r := bytes.NewReader([]byte("Hello World!"))
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindReaderIndex(r))
}
```

FindAllIndex

```
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
```

在 b 中查找 re 中编译好的正则表达式，并返回所有匹配的位置， {{起始位置, 结束位置}, {起始位置, 结束位置}, ...}，只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindAllIndex([]byte("Hello World!"), -1))
}
```

FindAllStringIndex

```
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
```

在 s 中查找 re 中编译好的正则表达式，并返回所有匹配的位置，{{起始位置, 结束位置}, {起始位置, 结束位置}, ...}，只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项

```
func main() {
    reg := regexp.MustCompile(`\w+`)
    fmt.Println(reg.FindAllStringIndex("Hello World!", -1))
}
```

FindSubmatch

```
func (re *Regexp) FindSubmatch(b []byte) [][]byte
```

在 b 中查找 re 中编译好的正则表达式，并返回第一个匹配的内容， 同时返回子表达式匹配的内容，{{完整匹配项}, {子匹配项}, {子匹配项}, ...}

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Printf("%q", reg.FindSubmatch([]byte("Hello World!")))
}
```

FindStringSubmatch

```
func (re *Regexp) FindStringSubmatch(s string) []string
```

在 s 中查找 re 中编译好的正则表达式，并返回第一个匹配的内容，同时返回子表达式匹配的内容，{完整匹配项, 子匹配项, 子匹配项, ...}

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Printf("%q", reg.FindStringSubmatch("Hello World!"))
    // ["Hello" "H" "o"]
}
```

FindAllSubmatch

```
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
```

在 b 中查找 re 中编译好的正则表达式，并返回所有匹配的内容

同时返回子表达式匹配的内容

{

{{完整匹配项}, {子匹配项}, {子匹配项}, ...},

{{完整匹配项}, {子匹配项}, {子匹配项}, ...},

...

}

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Printf("%q", reg.FindAllSubmatch([]byte("Hello World!"), -1))
    // [["Hello" "H" "o"] ["World" "W" "d"]]
}
```

FindAllStringSubmatch

```
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
```

在 s 中查找 re 中编译好的正则表达式，并返回所有匹配的内容，同时返回子表达式匹配的内容

{

{完整匹配项, 子匹配项, 子匹配项, ...},

{完整匹配项, 子匹配项, 子匹配项, ...},

...

}

只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Printf("%q", reg.FindAllStringSubmatch("Hello World!", -1))
    // [["Hello" "H" "o"] ["World" "W" "d"]]
}
```

FindSubmatchIndex

```
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

在 b 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置， 同时返回子表达式匹配的位置， {完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...}

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Println(reg.FindSubmatchIndex([]byte("Hello World!")))
    // [0 5 0 1 4 5]
}
```

FindStringSubmatchIndex

```
func (re *Regexp) FindStringSubmatchIndex(s string) []int
```

在 s 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置，同时返回子表达式匹配的位置

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...}

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Println(reg.FindStringSubmatchIndex("Hello World!"))
    // [0 5 0 1 4 5]
}
```

FindReaderSubmatchIndex

```
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
```

在 r 中查找 re 中编译好的正则表达式，并返回第一个匹配的位置，同时返回子表达式匹配的位置

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...}

```
func main() {
    r := bytes.NewReader([]byte("Hello World!"))
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Println(reg.FindReaderSubmatchIndex(r))
    // [0 5 0 1 4 5]
}
```

FindAllSubmatchIndex

```
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
```

在 b 中查找 re 中编译好的正则表达式，并返回所有匹配的位置，同时返回子表达式匹配的位置

{

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...},

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...},

...

}

只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Println(reg.FindAllSubmatchIndex([]byte("Hello World!"), -1))
    // [[0 5 0 1 4 5] [6 11 6 7 10 11]]
}
```

FindAllStringSubmatchIndex

```
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]int
```

在 s 中查找 re 中编译好的正则表达式，并返回所有匹配的位置， 同时返回子表达式匹配的位置

{

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...},

{完整项起始, 完整项结束, 子项起始, 子项结束, 子项起始, 子项结束, ...},

...

}

只查找前 n 个匹配项，如果 n < 0，则查找所有匹配项

```
func main() {
    reg := regexp.MustCompile(`(\w)(\w)+`)
    fmt.Println(reg.FindAllStringSubmatchIndex("Hello World!", -1))
    // [[0 5 0 1 4 5] [6 11 6 7 10 11]]
}
```

Expand

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, match []int)
[]byte
```

将 template 的内容经过处理后，追加到 dst 的尾部。

template 中要有 $1、$2、${name1}、${name2} 这样的“分组引用符”

match 是由 FindSubmatchIndex 方法返回的结果，里面存放了各个分组的位置信息，如果 template 中有“分组引用符”，则以 match 为标准，在 src 中取出相应的子串，替换掉 template 中的 $1、$2 等引用符号。

```
func main() {
    reg := regexp.MustCompile(`(\w+),(\w+)`)
    src := []byte("Golang,World!") // 源文本
    dst := []byte("Say: ") // 目标文本
    template := []byte("Hello $1, Hello $2") // 模板
    match := reg.FindSubmatchIndex(src) // 解析源文本
    // 填写模板，并将模板追加到目标文本中
    fmt.Printf("%q", reg.Expand(dst, template, src, match))
}
```

ExpandString

```
func (re *Regexp) ExpandString(dst []byte, template string, src string, match
[]int) []byte
```

功能同 Expand 一样，只不过参数换成了 string 类型

```
func main() {
    reg := regexp.MustCompile(`(\w+),(\w+)`)
    src := "Golang,World!" // 源文本
    dst := []byte("Say: ") // 目标文本(可写)
    template := "Hello $1, Hello $2" // 模板
    match := reg.FindStringSubmatchIndex(src) // 解析源文本
    // 填写模板，并将模板追加到目标文本中
    fmt.Printf("%q", reg.ExpandString(dst, template, src, match))
}
```

LiteralPrefix

```
func (re *Regexp) LiteralPrefix() (prefix string, complete bool)
```

LiteralPrefix 返回所有匹配项都共同拥有的前缀(去除可变元素)

prefix：共同拥有的前缀

complete：如果 prefix 就是正则表达式本身，则返回 true，否则返回 false

```
func main() {
    reg := regexp.MustCompile(`Hello[\w\s]+`)
    fmt.Println(reg.LiteralPrefix())

    reg = regexp.MustCompile(`Hello`)
    fmt.Println(reg.LiteralPrefix())
}
```

Longest

```
func (re *Regexp) Longest()
```

切换到“贪婪模式”

```
func main() {
    text := `Hello World, 123 Go!`
    pattern := `(?U)H[\w\s]+o` // 正则标记“非贪婪模式”(?U)
    reg := regexp.MustCompile(pattern)
    fmt.Printf("%q\n", reg.FindString(text))

    reg.Longest() // 切换到“贪婪模式”
    fmt.Printf("%q\n", reg.FindString(text))
}
```

Match

```
func (re *Regexp) Match(b []byte) bool
```

判断在 b 中能否找到匹配项

```
func main() {
    b := []byte(`Hello World`)
    reg := regexp.MustCompile(`Hello\w+`)
    fmt.Println(reg.Match(b))

    reg = regexp.MustCompile(`Hello[\w\s]+`)
    fmt.Println(reg.Match(b))
}
```

MatcReader

```
func (re *Regexp) MatchReader(r io.RuneReader) bool
```

判断在 r 中能否找到匹配项

```
func main() {
    r := bytes.NewReader([]byte(`Hello World`))
    reg := regexp.MustCompile(`Hello\w+`)
    fmt.Println(reg.MatchReader(r))

    r.Seek(0, 0)
    reg = regexp.MustCompile(`Hello[\w\s]+`)
    fmt.Println(reg.MatchReader(r))
}
```

MatchString

```
func (re *Regexp) MatchString(s string) bool
```

判断在 s 中能否找到匹配项

```
func main() {
    s := `Hello World`
    reg := regexp.MustCompile(`Hello\w+`)
    fmt.Println(reg.MatchString(s))

    reg = regexp.MustCompile(`Hello[\w\s]+`)
    fmt.Println(reg.MatchString(s))
}
```

NumSubexp

```
func (re *Regexp) NumSubexp() int
```

统计正则表达式中的分组个数(不包括“非捕获的分组”)

```
func main() {
    reg := regexp.MustCompile(`(?U)(?:Hello)(\s+)(\w+)`)
    fmt.Println(reg.NumSubexp())
}
```

ReplaceAll

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
```

在 src 中搜索匹配项，并替换为 repl 指定的内容，全部替换，并返回替换后的结果

```
func main() {
    b := []byte("Hello World, 123 Go!")
    reg := regexp.MustCompile(`(Hell|G)o`)
    rep := []byte("${1}ooo")
    fmt.Printf("%q\n", reg.ReplaceAll(b, rep))
}
```

ReplaceAllString

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

在 src 中搜索匹配项，并替换为 repl 指定的内容，全部替换，并返回替换后的结果

```
func main() {
    s := "Hello World, 123 Go!"
    reg := regexp.MustCompile(`(Hell|G)o`)
    rep := "${1}ooo"
    fmt.Printf("%q\n", reg.ReplaceAllString(s, rep))
}
```

ReplaceAllLiteral

```
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
```

在 src 中搜索匹配项，并替换为 repl 指定的内容，如果 repl 中有“分组引用符”($1、$name)，则将“分组引用符”当普通字符处理，全部替换，并返回替换后的结果

```
func main() {
    b := []byte("Hello World, 123 Go!")
    reg := regexp.MustCompile(`(Hell|G)o`)
    rep := []byte("${1}ooo")
    fmt.Printf("%q\n", reg.ReplaceAllLiteral(b, rep))
}
```

ReplaceAllLiteralString

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
```

在 src 中搜索匹配项，并替换为 repl 指定的内容，如果 repl 中有“分组引用符”($1、$name)，则将“分组引用符”当普通字符处理，全部替换，并返回替换后的结果

```
func main() {
    s := "Hello World, 123 Go!"
    reg := regexp.MustCompile(`(Hell|G)o`)
    rep := "${1}ooo"
    fmt.Printf("%q\n", reg.ReplaceAllLiteralString(s, rep))
}
```

ReplaceAllFunc

```
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []byte) []byte
```

在 src 中搜索匹配项，然后将匹配的内容经过 repl 处理后，替换 src 中的匹配项，如果 repl 的返回值中有“分组引用符”($1、$name)，则将“分组引用符”当普通字符处理，全部替换，并返回替换后的结果

```
func main() {
    s := []byte("Hello World!")
    reg := regexp.MustCompile("(H)ello")
    rep := []byte("$0$1")
    fmt.Printf("%s\n", reg.ReplaceAll(s, rep))

    fmt.Printf("%s\n", reg.ReplaceAllFunc(s,
        func(b []byte) []byte {
            rst := []byte{}
            rst = append(rst, b...)
            rst = append(rst, "$1"...)
            return rst
        }))
}
```

ReplaceAllStringFunc

```
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string) string)
string
```

在 src 中搜索匹配项，然后将匹配的内容经过 repl 处理后，替换 src 中的匹配项，如果 repl 的返回值中有“分组引用符”($1、$name)，则将“分组引用符”当普通字符处理，全部替换，并返回替换后的结果

```
func main() {
    s := "Hello World!"
    reg := regexp.MustCompile("(H)ello")
    rep := "$0$1"
    fmt.Printf("%s\n", reg.ReplaceAllString(s, rep))

    fmt.Printf("%s\n", reg.ReplaceAllStringFunc(s,
        func(b string) string {
            return b + "$1"
        }))
}
```

Split

```
func (re *Regexp) Split(s string, n int) []string
```

在 s 中搜索匹配项，并以匹配项为分割符，将 s 分割成多个子串，最多分割出 n 个子串，第 n 个子串不再进行分割，如果 n < 0，则分割所有子串，返回分割后的子串列表

```
func main() {
    s := "Hello World\tHello\nGolang"
    reg := regexp.MustCompile(`\s`)
    fmt.Printf("%q\n", reg.Split(s, -1))
}
```

String

```
func (re *Regexp) String() string
```

返回 re 中的“正则表达式”字符串

```
func main() {
    re := regexp.MustCompile("Hello.*$")
    fmt.Printf("%s\n", re.String())
}
```

SubexpNames

```
func (re *Regexp) SubexpNames() []string
```

返回 re 中的分组名称列表，未命名的分组返回空字符串

返回值[0] 为整个正则表达式的名称

返回值[1] 是分组 1 的名称

返回值[2] 是分组 2 的名称

……

```
func main() {
    re := regexp.MustCompile("(?PHello) (World)")
    fmt.Printf("%q\n", re.SubexpNames())
}
```

小结

- 正则表达式是符合一定规则的表达式，用于匹配字符串中字符组合的模式。
- 正则表达式的设计思想就是使用一些描述性的符号和文字为字符串定义一个规则。凡是符合这个规则的，程序就认为文本是“匹配”的，否则就认为文本是“不匹配”的。
- Go在处理正则表达式时主要使用regexp包，包中实现了正则表达式的查找、替换和模式匹配功能。

常用正则表达式参考

E-mail 地址

```
^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$
```

URL 地址

```
^(https?:\/\/)?([\da-z.-]+)\.([a-z.]{2,6})([\/\w .-])\/?$
```

匹配首尾空白字符的正则表达式

```
^\s|\s$
```

手机号码

```
^(13[0-9]|14[0-9]|15[0-9]|166|17[0-9]|18[0-9]|19[8|9])\d{8}$
```

电话号码

```
^(\d{3,4}-)?\d{7,8}$
```

18位身份证号码（数字、字母x结尾）

```
^((\d{18})|([0-9x]{18})|([0-9X]{18}))$
```

账号是否合法（5～16字节，允许字母、数字、下划线，以字母开头）

```
^[a-zA-Z][a-zA-Z0-9_]{4,15}$
```

一年的12个月（01～09和1～12）

```
^(0?[1-9]|1[0-2])$
```

日期格式（2018-01-01只做粗略匹配，格式不限制，二月有30天等）

```
^\d{4}-\d{1,2}-\d{1,2}$
```

一个月的31天（01～09和1～31）

```
^((0?[1-9])|((1|2)[0-9])|30|31)$
```

IP 地址

```
^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$
```