Go语言:正则表达式

听雨诺言 6月9日

Go语言的正则表达式在regexp包中,所以在项目中使用时需要导入regexp包:

```
1 import "regexp"
```

在使用正则表达式之前，需要将模式字符串编译成正则表达式实例，这样就可以通过该实例进行相应方法的操作，并且可以被多线程安全地同时使用。regexp包提供了四种获取正则表达式实例:

- func Compile(expr string) (*Regexp, error)
- func CompilePOSIX(expr string) (*Regexp, error)
- func MustCompile(str string) *Regexp
- func MustCompilePOSIX(str string) *Regexp

区别:

Compile和CompilePOSIX函数编译了正则表达式之后，如果正则表达式不合法，会返回error，但CompilePOSIX会将语法约束到POSIX ERE（egrep）语法，并将匹配模式设置为leftmost-longest。

MustCompile和MustCompilePOSIX在编译正则表达式过程中，如果正则表达式不合法，会抛出panic异常。

语法规则

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
[...]	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]，	a[bcd]e	abe 或 ace 或 ade

	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。		
\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)	后面不是数字的 a
(?<=...)	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

匹配模式：

如果正则表达式能匹配到传入的字符串（或字节切片），则会返回true。

```
MatchString类似Match，但匹配对象是字符串
func (re *Regexp) MatchString(s string) bool {
  return re.doMatch(nil, nil, s)
}

Match检查b中是否存在匹配pattern的子序列
func (re *Regexp) Match(b []byte) bool {
  return re.doMatch(nil, b, "")
```

```
}
```

来看一下使用案例：

```
re := regexp.MustCompile(`^Golang.*架构师`)
strs := "Golangskdshsdjs.posub架构师"
str := "jhhajjjGolangsdhhshd架构师"
fmt.Println(re.MatchString(strs))  // 输出：true
fmt.Println(re.MatchString(str))  // 输出：false
```

这个例子中，模式字符串希望匹配以“Golang”这个字符串开头，中间包含一系列字符串，然后末尾包含“架构师”这个字符串，然而strs这个字符串以"Golang"开头，所以匹配成功，返回true。但是str以"jhha"开头，不满足匹配条件，所以匹配失败，返回false。

查找模式：
查找最典型的函数就是FindString和FindStringSubmatch。我们来看一下，主要有以下几类：

```
// 查找能匹配的字符串，查找所匹配的字符串的起止位置
func (re *Regexp) FindString(s string) string
func (re *Regexp) FindStringIndex(s string) (loc []int)
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)

// 查找能匹配的字符串和所有的匹配组
func (re *Regexp) FindStringSubmatch(s string) []string
func (re *Regexp) FindStringSubmatchIndex(s string) []int
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int

// 查找所有能匹配的字符串（最多查找n次。如果n为负数，则查找所有能匹配到的字符串，以切片
func (re *Regexp) FindAllString(s string, n int) []string
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int

// 查找所有能匹配的字符串（最多查找n次。如果n为负数，则查找所有能匹配到的字符串，以切片
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]int
```

FindString函数会查找第一个能被正则表达式匹配到的字符串，并返回匹配到的字符串。FindStringIndex会返回匹配到的字符串的起止位置。

```
re := regexp.MustCompile(`Go语言(.*?)架构师`)  // 此处小括号中的问号表示勉强型匹配，见下文第五点
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串333_架构师"
fmt.Println(re.FindString(str))  // 输出：Go语言_中间字符串111_架构师
fmt.Println(re.FindStringIndex(str))  // 输出：[7 44]
```

可以看到，FindString找到了字符串中**第一个**被正则表达式匹配到的字符串，FindStringIndex返回了它的起止位置（由此可见，FindStringIndex结果切片只会包含2个元素）。

FindStringSubmatch函数不仅会查找第一个能被正则表达式匹配到的字符串，还会找出其中匹配组所匹配到的字符串（即正则表达式中小括号里的内容），会放在切片中一起返回。FindStringSubmatchIndex不仅会返回匹配到的字符串的起止位置，还会返回匹配组所匹配到的字符串起止位置。

```
re := regexp.MustCompile(`Go语言(.*?)架构师`)
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串333_架构师"
fmt.Println(re.FindStringSubmatch(str))  // 输出：[Go语言_中间字符串111_架构师 _中间字符串111_]
fmt.Println(re.FindStringSubmatchIndex(str))  // 输出：[7 44 15 35]
```

FindStringSubmatch结果中的第一个元素“Go语言_中间字符串111_架构师”就是整个正则表达式所匹配到的第一个字符串；结果中的第二个元素就是匹配组（即正则表达式的小括号内的内容）所匹配到的第一个字符串。匹配组是干啥用的？匹配组就是在整个正则表达式的匹配结果上，再进行的一次匹配。

和FindString不一样，FindString会查找**第一个**能被正则表达式匹配到的字符串。FindAllString函数会查找**n个（n是FindAllString的第二个参数）**能被正则表达式匹配到的字符串，并返回所有匹配到的字符串所组成的切片。FindAllStringIndex会返回所有匹配到的字符串的起止位置，结果是个二维切片。如果n为整数，则最多匹配n次；如果n为负数，则会返回所有匹配结果。

```
re := regexp.MustCompile(`Go语言(.*?)架构师`)
```

```
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串333_架构师"
fmt.Println(re.FindAllString(str, -1))  // 输出：[Go语言_中间字符串111_架构师 Go语言_中间字符串333_架构师]
fmt.Println(re.FindAllStringIndex(str, -1))  // 输出：[[7 44] [64 101]]
```

第二个参数传入-1，表示要返回所有匹配结果。可以看到，有匹配结果时，FindAllStringIndex的结果是个二维切片。

FindAllStringSubmatch和FindAllStringSubmatchIndex

和第二类一样，都是在FindAllString的结果上，再返回匹配组所匹配到的内容。看一下例子：

```
re := regexp.MustCompile(`Go语言(.*?)架构师`)
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串333
fmt.Println(re.FindAllStringSubmatch(str, -1))  // 输出：[[Go语言_中间字符串111_
fmt.Println(re.FindAllStringSubmatchIndex(str, -1))  // 输出：[[7 44 15 35] [64
```

对于字符串str，FindAllString能查找到两个匹配结果，在这两个匹配结果上，FindAllStringSubmatch再返回匹配组所匹配到的内容，那么结果就是下面这个二维切片了：

```
[[Go语言_中间字符串111_架构师 _中间字符串111_] [Go语言_中间字符串333_架构师 _中间字符串333_]]
```

这四类函数在解析爬虫网页的时候特别有用。

替换模式：

替换是正则表达式中另一个非常有用的功能。替换主要有三个函数：

ReplaceAllString

ReplaceAllString会把第一个参数所表示的字符串中所有匹配到的内容用第二个参数代替。在第二个参数中，可以使用$符号来引用匹配组所匹配到的内容。$0表示第0个匹配组所匹配到的内容，即整个正则表达式所匹配到的内容。$1表示第一个匹配组所匹配到的内容，即正则表达式中第一个小括号内的正则表达式所匹配到的内容。

```
func (re *Regexp) ReplaceAllString(src, repl string) string
ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。
```

```
在替换时，repl中的'$'符号会按照Expand方法的规则进行解释和替换，

例如$1会被替换为第一个分组匹配结果
re := regexp.MustCompile(`Go语言(.*?)架构师`)
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串33
fmt.Println(re.ReplaceAllString(str, "$1"))

输出：
前缀_____中间字符串111_____中间字符串222_____中间字符串333_
```

这个例子把str字符串中所有被正则表达式所匹配到的内容用第一个匹配组的内容进行替换。“Go语言(.*?)架构师”这个正则表达式能匹配到字符串“Go语言_中间字符串111_架构师”，而第一个匹配组（即.*?）所匹配到的内容是“_中间字符串111_”，所以最终结果就是“前缀____中间字符串111_____中间字符串222_____中间字符串333_”。

ReplaceAllLiteralString

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
ReplaceAllLiteralString返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。repl参
case:
re := regexp.MustCompile(`Go语言(.*?)架构师`)
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串333
fmt.Println(re.ReplaceAllLiteralString(str, "$1"))
输出：
前缀____$1____中间字符串222____$1
```

ReplaceAllStringFunc

```
ReplaceAllLiteral返回src的一个拷贝，
将src中所有re的匹配结果（设为matched）都替换为repl(matched)。
repl返回的字符串被直接使用，不会使用Expand进行扩展。
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string) string) s

re := regexp.MustCompile(`Go语言(.*?)架构师`)
str := "前缀____Go语言_中间字符串111_架构师____中间字符串222____Go语言_中间字符串33
fmt.Println(re.ReplaceAllStringFunc(str, func(s string) string {
  return "Q" + s + "Q"
```

```
})
输出：
前缀____QGo语言_中间字符串111_架构师Q____中间字符串222____QGo语言_中间字符串333_架构
```

三种匹配模式：

任何语言的正则表达式匹配都避免不了正则表达式的三种匹配模式：贪婪型、勉强型、占有型。

贪婪型属于正常的表示（平时写的那些），勉强型则在后面加个“问号”，占有型加个“加号”，都只作用于前面的问号、星号、加号、大括号，因为前面如果没有这些，就变成普通的问号和加号了（也就是变成贪婪型了）。

- 贪婪型匹配模式表示尽可能多的去匹配字符。
- 勉强型匹配模式表示尽可能少的去匹配字符。
- 占有型匹配模式表示尽可能做完全匹配。

贪婪型
贪婪型匹配模式的正则表达式形式为**星号或者加号**。我们知道，星号表示匹配0个或任意多个字符，加号表示匹配1个或者任意多个字符。
贪婪型匹配，先一直匹配到最后，发现最后的字符不匹配时，往前退一格再尝试匹配，不匹配时再退一格。

看一下例子就很明白了：

```
re := regexp.MustCompile(`我是.*字符串`)
str := "我是第1个字符串_我是第2个字符串"
fmt.Println(re.FindString(str))  // 输出：我是第1个字符串_我是第2个字符串
```

这个例子的输出是“我是第1个字符串_我是第2个字符串”，而不是“我是第1个字符串”。这个结果和勉强型匹配形成了强烈的对比。

勉强型
勉强型匹配模式的正则表达式形式为**星号或者加号**，后面再**加个问号**（注意与贪婪型的区别）。我们知道，星号表示匹配0个或任意多个字符，加号表示匹配1个或者任意多个字符。后面加个问号表示尽可能少的去匹配字符。

看一下例子就很明白了，还是上面那个例子，在正则表达式的星号后面加个问号：

```
re := regexp.MustCompile(`我是.*?字符串`)
str := "我是第1个字符串_我是第2个字符串"
fmt.Println(re.FindString(str))  // 输出：我是第1个字符串
```

这个例子的输出是“我是第1个字符串”。和贪婪型匹配结果形成了强烈的对比。

占有型

占有型匹配模式的正则表达式形式为**星号或者加号**，后面再**加个加号**（注意与贪婪型、勉强型的区别）。我们知道，星号表示匹配0个或任意多个字符，加号表示匹配1个或者任意多个字符。后面加个加号表示正则表达式必须完全匹配整个字符串。

Go语言中正则表达式没有“占有型”，如果想实现完全匹配，在正则表达式中使用“^”和“$”表示首尾就好了。
我们可以尝试一下，还是上面那个例子，在正则表达式的星号后面加个加号结果报错：

```
re := regexp.MustCompile(`我是.*+字符串`)
str := "我是第1个字符串_我是第2个字符串"
fmt.Println(re.FindString(str))
```