16章 正则表达式

作者 bluetea
网站:https://github.com/bluetea

16.1.2
正则表达式的对象的创建
使用%的特殊语法来创建

```
[3] pry(main)> a = %r([A-D]/d+)
=> /[A-D]\/d+/
[4] pry(main)> a = %r|[A-D]/d+|
=> /[A-D]\/d+/
[5] pry(main)> a = %r<[A-D]/d+>
=> /[A-D]\/d+/
[6] pry(main)> a = %!<[A-D]/d+!
=> "<[A-D]/d+"
[7] pry(main)> a = %r![A-D]/d+!
=> /[A-D]\/d+/
[8] pry(main)>
```

16.2 正则表达式的模式和匹配

正则表达式 =~ 字符串
1.当无法匹配是会返回nil，所以可以根据这个进行判断
2.当匹配成功，则会返回该字符串其实字符的位置
所以可以这么判断

```
1  if 正则表达式 =~ 字符串
2    匹配时的处理方法
3  else
4    不匹配时的处理方法
5  end
```

16,2.1 普通字符匹配

```
[10] pry(main)> /ABC/ =~ "ABC"
=> 0
[11] pry(main)> /ABC/ =~ "ABCDEF"
=> 0
[12] pry(main)> /ABC/ =~ "123DEF"
=> nil
[13] pry(main)> /ABC/ =~ "123ABC"
=> 3
```

16.2.2 匹配首行与尾行
^ 匹配行首
$ 匹配行尾

```
[16] pry(main)> /^ABC$/ =~ "ABC"
=> 0
[17] pry(main)> /^ABC$/ =~ "1ABC"
=> nil
[18] pry(main)> /^ABC$/ =~ "ABCc"
=> nil
[19] pry(main)> /^ABC/ =~ "ABCc"
=> 0
[20] pry(main)> /^ABC/ =~ "1ABC"
=> nil
[21] pry(main)> /ABC$/ =~ "1ABC"
=> 1
[22] pry(main)> /ABC$/ =~ "ABCd"
=> nil
```

^ 匹配行首,$ 匹配行尾,其实这种表述不准确，应该叫字符串头和尾，但是这个是因为历史原因,正则表达式只能逐行匹配字符串，不能匹配多行字符串，所以可以认为一个字符串就是一行，随着正则表达式的广泛应用，如果仍用 ^ $匹配字符串的开头和结尾，容易造成混乱，所以又另外定义了匹配字符串开头结尾的元字符 \A \z. 另外还有一个类似的\Z和\z的两者作用不同
在ruby的字符串，也就是String的对象中，所谓行就是用换行符(\n)间隔的字符串，因此模式/^ABC/可以匹配”012\nABC” ，因为第二行的开头是ABC

```
[23] pry(main)> /^ABC/ =~ "aabcc\nABC"
=> 6
[25] pry(main)> /\AABC/ =~ "aabcc\nABC"
=> nil
[26] pry(main)> /\AABC/ =~ "ABC"
=> 0
```

注意区别

```
[32] pry(main)> "def\n".gsub(/\Z/, "!")
=> "def!\n!"
[33] pry(main)> "def\n".gsub(/\z/, "!")
=> "def\n!"
```

\Z虽然也是匹配字符串末尾元字符， 但是一个特点就是， 如果字符串末尾是换行符，则匹配换行符前一个字符，我们一般很少使用\Z

16.2.3 指定匹配字符的范围

[ABC] ABC任意一个字符

[A-Z] 任意一个大写字母

[a-z] 任意一个小写字母

[0-9] 任意一个数字

[A-Za-z] 任意一个字母

[A-Za-z_] 任意一个字母和_

```
[47] pry(main)> /[A-Za-z]/ =~ "-sdfsdfsdf"
=> 1
[48] pry(main)> /[A-Za-z_]/ =~ "sdfsdfsdf"
=> 0
```

```
[53] pry(main)> /[^ABC]/ =~ "D"
=> 0
[54] pry(main)> /[^ABC]/ =~ "a"
=> 0
```

非 ABC之外的任意字符

```
[55] pry(main)> /[^A-Za-z]/ =~ "a"
=> nil
[56] pry(main)> /[^A-Za-z]/ =~ "454"
=> 0
```

16.2.4 匹配任意一个字符

. 匹配任意字符

```
[57] pry(main)> /A.C/ =~ "ABC"
=> 0
[58] pry(main)> /A.C/ =~ "AxC"
=> 0
[59] pry(main)> /A.C/ =~ "23423A2C234234"
=> 5
[60] pry(main)> /A.C/ =~ "AC"
=> nil
[61] pry(main)> /A..C/ =~ "A23C"
=> 0
[62] pry(main)> /A.C/ =~ "A23C"
=> nil
[63] pry(main)> /aaa.../ =~ "A23Caaaebd"
=> 4
```

这个主要用在：

1.希望指定字符数的时候

/^...$/ 可以匹配字符数为3个行

```
[64] pry(main)> /^...$/ =~ "dfgd"
=> nil
[65] pry(main)> /^...$/ =~ "dfd"
=> 0
[66] pry(main)> /^...$/ =~ "dfd "
=> nil
[67] pry(main)> /^...$/ =~ "dfd\n"
=> 0
```

2.配合元字符 *使用

16.2.5使用\反斜杠模式

\s 匹配空格(0x20)，空白符

```
[68] pry(main)> /ab\scd/ =~"abcd"
=> nil
[69] pry(main)> /ab\scd/ =~"ab cd"
=> 0
```

\d 匹配0-9的数字

```
[70] pry(main)> /\d\d\d.\d\d\d/ =~"111222"
=> nil
[71] pry(main)> /\d\d\d.\d\d\d/ =~"1111222"
=> 0
```

\w 匹配英文字母与数字

\A匹配字符串的开头

```
[74] pry(main)> /\AABC/ =~"ABC"
=> 0
[75] pry(main)> /\AABC/ =~"ABCCED"
=> 0
[76] pry(main)> /\AABC/ =~"012ABCCED"
=> nil
[77] pry(main)> /\AABC/ =~"\nABCCED"
=> nil
[78] pry(main)> /^ABC/ =~"\nABCCED"
=> 1
```

\z 匹配字符串末尾

```
[79] pry(main)> /ABC\z/ =~"\nABC"
=> 1
[80] pry(main)> /ABC\z/ =~"\nABCDEF"
=> nil
[81] pry(main)> /ABC\z/ =~"\nABC\nDEF"
=> nil
[82] pry(main)> /ABC$/ =~"\nABC\nDEF"
=> 1
```

元字符的转义

\[\^ \$

16.2.6重复

* 重复0次以上

+ 重复1次以上

? 重复0次或1次

```
[97] pry(main)> "AC".sub(/A*C/, "T")
=> "T"
[98] pry(main)> "AAAC".sub(/A*C/, "T")
=> "T"
[99] pry(main)> "AAAAB".sub(/A*C/, "T")
=> "AAAAB"
[100] pry(main)> "".sub(/A*/, "T")
=> "T"
[101] pry(main)> "AAC".sub(/AAAA*C/, "T")
=> "AAC"
[102] pry(main)> "AAC".sub(/AAA*C/, "T")
=> "T"
[103] pry(main)> "AC".sub(/AAA*C/, "T")
=> "AC"
[104] pry(main)> "AB012C".sub(/A.*C/, "T")
=> "T"
[105] pry(main)> "ABCD".sub(/A.*C/, "T")
=> "TD"
[106] pry(main)> "ACD".sub(/A.*C/, "T")
=> "TD"
[107] pry(main)> "ACDE".sub(/A.*C/, "T")
=> "TDE"
```

使用*的例子

```
[108] pry(main)> "Subject: foo".sub(/^Subject:\s*.*$/, "T")
=> "T"
[109] pry(main)> "Subject: Re: Foo".sub(/^Subject:\s*.*$/, "T")
=> "T"
[110] pry(main)> "Subject: Re^2 Foo".sub(/^Subject:\s*.*$/, "T")
=> "T"
[111] pry(main)> "in Subject: Re foo".sub(/^Subject:\s*.*$/, "T")
=> "in Subject: Re foo"
```

+表示重复1次以上，例子

```
[1] pry(main)> "A".sub(/A+/, "T")
=> "T"
[2] pry(main)> "AAAA".sub(/A+/, "T")
=> "T"
[3] pry(main)> "".sub(/A+/, "T")
=> ""
[4] pry(main)> "BBB".sub(/A+/, "T")
=> "BBB"
[5] pry(main)> "AAAC".sub(/A+/, "T")
=> "TC"
[6] pry(main)> "BC".sub(/A+/, "T")
=> "BC"
[7] pry(main)> "BC".sub(/A+C/, "T")
=> "BC"
[8] pry(main)> "AAAC".sub(/A+C/, "T")
=> "T"
[9] pry(main)> "AAAB".sub(/A+C/, "T")
=> "AAAB"
[10] pry(main)> "AAAC".sub(/AAA+C/, "T")
=> "T"
[11] pry(main)> "AC".sub(/AAA+C/, "T")
=> "AC"
[12] pry(main)> "AAC".sub(/AAA+C/, "T")
=> "AAC"
[13] pry(main)> "AAC".sub(/A.+C/, "T")
=> "T"
[14] pry(main)> "AC".sub(/A.+C/, "T")
=> "AC"
[15] pry(main)> "AdfgdgC".sub(/A.+C/, "T")
=> "T"
[16] pry(main)> "AB CD".sub(/A.+C/, "T")
=> "TD"
```

使用?号的例子 匹配0次或1次

16.2.7最短匹配

匹配0次以上的*, 和匹配1次以上的+ 会匹配尽可能多的字符,----贪婪模式
当我们想匹配尽可能少的字符时（懒惰模式），用一下字符
*? 0次以上重复最短的部分
+? 1次以上重复最短的部分

```
[18] pry(main)> "ABCDABCDABCD".sub(/A.*B/, "T")
=> "TCD"
[19] pry(main)> "ABCDABCDABCD".sub(/A.*?B/, "T")
=> "TCDABCDABCD"
[20] pry(main)> "ABCDABCDABCD".sub(/A.*C/, "T")
=> "TD"
[21] pry(main)> "ABCDABCDABCD".sub(/A.*?C/, "T")
=> "TDABCDABCD"
[22] pry(main)> "ABCDABCDABCD".sub(/A.+B/, "T")
=> "TCD"
[23] pry(main)> "ABCDABCDABCD".sub(/A.+?B/, "T")
=> "TCDABCD"
[24] pry(main)> "ABCDABCDABCD".sub(/A.+C/, "T")
=> "TD"
[25] pry(main)> "ABCDABCDABCD".sub(/A.+?C/, "T")
=> "TDABCDABCD"
```

16.28()与重复
使用通过()可以重复匹配多个字符

```
[26] pry(main)> "ABC".sub(/^(ABC)*$/, "T")
=> "T"
[27] pry(main)> "".sub(/^(ABC)*$/, "T")
=> "T"
[28] pry(main)> "ABCABC".sub(/^(ABC)*$/, "T")
=> "T"
[29] pry(main)> "ABCABC".sub(/^(ABC)+$/, "T")
=> "T"
[30] pry(main)> "".sub(/^(ABC)+$/, "T")
=> ""
[31] pry(main)> "ABC".sub(/^(ABC)+$/, "T")
=> "T"
[32] pry(main)> "ABC".sub(/^(ABC)?$/, "T")
=> "T"
[33] pry(main)> "".sub(/^(ABC)?$/, "T")
=> "T"
[34] pry(main)> "ABCABC".sub(/^(ABC)?$/, "T")
=> "ABCABC"
```

16.2.9 选择

```
[35] pry(main)> "ABCABC".sub(/^(ABC|DCE)$/, "T")
=> "ABCABC"
[36] pry(main)> "ABC".sub(/^(ABC|DCE)$/, "T")
=> "T"
[37] pry(main)> "DCE".sub(/^(ABC|DCE)$/, "T")
=> "T"
[38] pry(main)> "ABCABC".sub(/(ABC|DCE)/, "T")
=> "TABC"
[39] pry(main)> "ABCDCE".sub(/(ABC|DCE)/, "T")
=> "TDCE"
[40] pry(main)> "ABCDCE".sub(/(ABC|DCE)+/, "T")
=> "T"
```

16.3使用quote方法的正则表达式
quote会返回转义了元字符后的正则表达式，然后再结，然后在结合new方法生成新的正则表达式
很不常用。

16.4 正则表达式的选项
正则表达式有选项，额可以改变正则表达式的一些默认效果

1./表达式/i 忽略英文大小写的选项，指定这个选项后， 无论字符串中的字母是大写还是小写，都会被匹配

```
[15] pry(main)> "aabcd".sub(/BCD/i, "T")
=> "aaT"
```

2./表达式/x 忽略**表达式中**的空白字符以及#后面的字符的选项，指定这个选项后，可以给正则表达式写注释

```
[11] pry(main)> "AABCD".sub(/BCD/x, "T")
=> "AAT"
[12] pry(main)> "AABCD".sub(/BC D/x, "T")
=> "AAT"
[13] pry(main)> "AABCD".sub(/BC D#这个是测试/x, "T")
=> "AAT"
```

3./表达式/m 指定这个选项后，可以使用.匹配换行符

```
[7] pry(main)> "ABC\nDEF\nGHI\r".sub(/DEF.GHI/m, "T")
=> "ABC\nT\r"
[8] pry(main)> "ABC\nDEF\rGHI\r".sub(/DEF.GHI/m, "T")
=> "ABC\nT\r"
```

这几个选项都有自己的选项常量，为了配合Regexp.new方法使用的，例如
i Regexp::IGNORECASE 忽略大小写
x Regexp::EXTENDED 忽略表达式中的空白字符
m Regexp::MULTLINE 匹配多行
o 无 只使用一次内嵌表达式
例如: 创建一个 /Ruby脚本/i 的表达式

```
[16] pry(main)> Regexp.new("Ruby脚本", Regexp::IGNORECASE)
=> /Ruby脚本/i
```

创建一个/Ruby脚本/im的表达式

```
[17] pry(main)> Regexp.new("Ruby脚本", Regexp::IGNORECASE | Regexp::MULTILINE)
=> /Ruby脚本/mi
```

16.5捕获
除了检查字符串外，正则表达式有一个捕获功能
意思就是正则表达式从匹配部分取出其中部分， 保存到 $数字形式的变量中，可以获取到表达式中()扩住的部分字符集

```
[18] pry(main)> /(.)(.)(.)/ =~ "ABC"
=> 0
[19] pry(main)> first = $1
=> "A"
[20] pry(main)> second = $2
=> "B"
[21] pry(main)> thrid = $3
=> "C"
[22] pry(main)> forth = $4
=> nil
```

()也用于将多个模式整理为一个，一样也是捕获的

```
[26] pry(main)> "ABCABC".sub(/(ABC)+/, "T")
=> "T"
[27] pry(main)> $1
=> "ABC"
```

如果修改程序的正则表达式也会更改顺序，所以索引也会修改，这回带来不便，所以可以使用(?:)来过滤不需要捕获的

```
[32] pry(main)> /(.)(\d\d)+(.)/ =~ "123456"
=> 0
[33] pry(main)> $1
=> "1"
[34] pry(main)> $2
=> "45"
[35] pry(main)> $3
=> "6"
[36] pry(main)> /(.)(?:\d\d)+(.)/ =~ "123456"
=> 0
[37] pry(main)> $1
=> "1"
[38] pry(main)> $2
=> "6"
```

除了$数字，还有$` $& 和$' 分别代表 匹配部分前字符，匹配部分字符, 和匹配部分后的字符串，为了，如下：

```
[41] pry(main)> /C./ =~ "ABCDEF"
=> 2
[42] pry(main)> $`
=> "AB"
[43] pry(main)> $&
=> "CD"
[44] pry(main)> $'
=> "EF"
```

使用正则表达式的方法。
sub方法和gsub方法。
sub方法只替换首次匹配的内容
gsub方法替换所有匹配内容

```
[47] pry(main)> str = "abc   def  g   hi"
=> "abc   def  g   hi"
[48] pry(main)> str.sub(/\s+/, " ")
=> "abc def  g   hi"
[49] pry(main)> str.gsub(/\s+/, " ")
=> "abc def g hi"
```

sub和 gsub方法还可以使用块

```
[53] pry(main)> str.sub(/.a/) do |matched|
[53] pry(main)*   '<'+matched.upcase+'>'
[53] pry(main)* end
=> "ab<RA>catabra"
[54] pry(main)> str.gsub(/.a/) do |matched|
[54] pry(main)*   '<'+matched.upcase+'>'
[54] pry(main)* end
=> "ab<RA><CA><TA>b<RA>"
```

16.6.2 scan方法

scan方法能像gsub方法那样获取匹配部分的字符， 但不能做置换操作，因此当要对匹配部分做某种处理可以用这个scan方法

```
[58] pry(main)> str.scan(/.a/) {|matched| p matched}
"ra"
"ca"
"ta"
"ra"
=> "abracatabra"
```

```
[65] pry(main)> str.scan(/(.)(a)/) {|a,b| puts a+b}
ra
ca
ta
ra
=> "abracatabra"
```

16.7正则表达式的例子，找出包含url的行


[2] pry(main)> %r|http://([^/]*)| =~ "http://www.sina.com.cn"
=> 0
[3] pry(main)> %r|http://([^/]*)| =~ "https://www.sina.com.cn"
=> nil
[4] pry(main)> %r|https?://([^/]*)| =~ "https://www.sina.com.cn"
=> 0
[5] pry(main)> %r|https?://([^/]*)/| =~ "https://www.sina.com.cn"
=> nil
[6] pry(main)> %r|https?://([^/]*)/?| =~ "https://www.sina.com.cn"
=> 0
[7] pry(main)> %r|https?://([^/]*)/?| =~ "https://www.sina.com.cn/"
=> 0
[8] pry(main)> %r|https?://([^/]*)/?| =~ "https://www.sina.com.cn/adsfda"
=> 0
[9] pry(main)> %r|https?://([^/]*)/?| =~ "https://www.sina.com.cn/test"
=> 0
[10] pry(main)> %r|https?://([^/]*)/?| =~ "https://www.sina.com.cn/test/"
=> 0
[11] pry(main)> %r|https?://([^/]*)/?| =~ "http://www.sina.com.cn/test/"
=> 0
[12] pry(main)> $1
=> "www.sina.com.cn"