正则表达式

Regular Expression Tutorial

张子阳

jimmyzhang.cnblogs.com

jimmy_dev@163.com

目录

引言

正则表达式看上去并不像某种语言或者某个技术那么重要，仅靠它们你无法写出一个应用程序来。然而，它们却总是在你的开发过程中如影随形，不管是进行表单验证，还是高亮显示搜索结果，又或者是进行 URL 地址映射，总是需要使用它们。几乎所有的语言都对它进行了不同程度的支持，由此，足见其在文本匹配这一领域中的地位。

正则表达式应该算是开发人员应该掌握的一个轻量级的技能，然而，它稀奇古怪地匹配模式总让人们联想起外星语言。本文中，我期望能和大家一道，通过丰富地例子，由浅入深地对正则表达式作一个复习和回顾。

NOTE：本文章节的编排顺序和风格主要参考 Ben Forta 的《Sams Teach Yourself Regular Expressions in 10 Minutes》。

什么是正则表达式？

正则表达式也叫做匹配模式(Pattern)，它由一组具有特定含义的字符串组成，通常用于匹配和替换文本。

通常情况下，如果一个文本中出现了多个匹配，正则表达式返回第一个匹配，如果将 global 设置为 true，则会返回全部匹配；匹配模式是大小写敏感的，如果设置了 ignore case 为 true，则将忽略大小写区别。

本文中，有时会将 正则表达式 简称为 表达式、匹配模式 或者 模式，它们可以互换。本文默认 global 和 ignore case 均为 true。

准备工作

在开始进行之前，我们首先需要一份工具来对表达式进行测试，在本文中，使用的是 RegexTester 这个小程序，它的使用非常的简单明了，相信你随便摆弄下就会用了，这里就不对其使用再做说明了。

可以点这里下载：http://www.cnblogs.com/Files/JimmyZhang/RegexTester.zip

匹配单个字符

1. 匹配固定单个字符

所有的 **单个** 大小写字母、数字，以及后面将要讲述的特殊字符，都是一个正则表达式，它们只能匹配单个字符，且这个字符与它本身相同，例如，对于表达式“i”：

Text

```
Jimmy is a junior developer and jimmy lives in xi'an.
```

RegEx

```
i
```

Result

```
Jimmy is a junior developer and jimmy lives in xi'an.
```

需要注意：对于表达式“m”来说，它的匹配是jimmy，请注意这里的匹配方式：并不是一个“m”一次匹配了字符串“mm”，而是一个“m”分两次匹配了单个字符“m”，不管这两个“m”是紧挨着 或者 如同上面的“i”那样分散在句子的不同地方。

将多个固定单个字符进行组合就构成了一个匹配固定字符串的表达式。例如：“Jimmy”，它也是一个正则表达式，它由多个匹配固定单个字符的表达式组成，它只可以匹配任何与它完全相同的文本：

Text

```
Jimmy is a junior developer and jimmy lives in xi'an.
```

RegEx

```
Jimmy
```

Result

```
Jimmy is a junior developer and jimmy lives in xi'an.
```

你可以将“Jimmy”这个表达式的匹配过程理解成这样：它由“j”、“i”、“m”、“m”、“y”这五固定的单个字符组成，对于每个单个字符来说，只能匹配与它完全相同的字符；而将这五个单个字符组合起来的时候，它就匹配 **字符排列顺序与表达式完全相同 并且 相应位置上的字符也与对应字符相同** 的字符串(实际上也就是与表达式完全相同的字符串)。

由于这种匹配方式的灵活度最小，只能匹配与它完全相同的字符，所以也叫“全字匹配”。

2. 匹配任意单个字符

“.”可以匹配任意的 **单个** 字符、英文字母、数字，以及它本身。我们现在结合上面介绍的全字匹配来学习它：

Text

```
regular.doc
regular1.exe
regular2.dat
expression.doc
```

```
express.dat
```

RegEx

```
regular.
```

Result

```
regular.doc
regular1.exe
regular2.dat
expression.doc
express.dat
```

可以看到，对于表达式“regular.”来说，前面“regular”部分是一个全字匹配，只能固定的匹配“regular”字符串；后面的“.”部分，可以匹配任意单个字符(包含字符“.”本身)。

“.”可以连续使用，比如，我们可以写出“.e..”这样一个正则表达式。它将匹配所有：前面有任意一个字符，紧跟着一个 e，随后又跟着任意两个字符的文本。

Text

```
regular.doc
regular1.exe
regular2.dat
expression.doc
express.dat
```

RegEx

```
.e..
```

Result

```
regular.doc
regular1.exe
regular2.dat
expression.doc
express.dat
```

NOTE：很多情况下，“.”不匹配换行。但在 RegexTester 这个工具中，它匹配换行，所以，这里有个值得注意的匹配，就是 **回车符+exp**，我在上面的代码中用绿色粗体标识出了(回车符难以标识)。

3. 匹配“.”元字符

有的时候，我们不想让“.”去匹配任何的字符，仅仅想让它匹配“.”这一单个字符，也就是仅匹配它本身，此时，可以使用“\.”来对它进行转义。

Text

```
regular.exe
regular1.exe
```

RegEx

```
r\.
```

Result

```
regular.exe
regular1.exe
```

表达式“r\.”仅仅匹配了字符串“r.”，没有匹配下面的 “r1”。

表达式“r.”则会匹配“r1”、“r.”、以及“re”。

NOTE：如果要匹配“\”，可以使用“\\”来对它进行转义。 后面还会介绍更多需要转义的字符。

4. 匹配字符组

4.1 字符组的基本语法

“.”过于灵活了，它可以匹配几乎所有的单个字符。有的时候，我们只希望匹配有限个字符中的某一个。这个时候，可以使用字符组。

假设有这样一种情况，我们希望验证某个单词是不是拼写正确，比如说 “head” 是一个单词，“heat”也是一个单词，但“heay”就不是一个单词，所以，“hea”后面要么出现“d”，要么“t”，而如果我们使用“.”来进行匹配，那么不管是“heay”也好，“heas”也罢，都会被匹配(被认为是正确的单词)。

正则表达式提供 **字符组** 来解决这一问题，对于上例，“hea”后面仅可以匹配“d”或者“t”的情况，它的语法是 “[dt]”。**中括号是特殊标记，用以划定属于组内的字符的界限，它所代表的含义是：“匹配 d 或者 t”。**

Text

```
bread
heay
teas
head
heat
```

RegEx

```
.ea[td]
```

Result

```
bread
heay
teas
head
heat
```

NOTE：字符组 虽然由多个字符构成，但它仍只匹配 **单个** 字符，而字符组能够匹配的单个字符，即是它定义中的字符（“[]”内的字符）。“[]”本身不进行字符匹配，它仅仅划定字符组边界。

4.2 在字符组中使用字符区间

现在假设我们需要匹配一组文件名，它们的名称为 city0.jpg、city1.jpg...city9.jpg。根据前面介绍的内容，我们很容易写出这样的表达式：“city[0123456789].jpg”。没错！这样写法的确可以达到我们要的效果。如果说写 10 个阿拉伯数字你觉得并不困难，那么如果要匹配这样的文件名呢？a_1.jpg、b_1.jpg、c_1.jpg...z_1.jpg。这次，你的表达式变成这样了：“[abcdefghijklmnopqrstuvwxyz]_1\.jpg”，哇！看着就难受。

正则表达式提供了字符区间来简化这一写法，它的语法是：**“起始字符-结束字符”。**对于上面匹配阿拉伯数字的例子，它的写法是[0-9]。

NOTE：不一定非要将匹配写成“[0-9]”，完全可以根据需要写成“[0-3]”，它将仅匹配“0,1,2,3”这个区间。**起始字符 和 结束字符，依据的是它的 ASCⅡ值的大小，即是说，将会匹配其 ASCⅡ码位于 起始字符 和 结束字符 的 ASCⅡ之间的所有字符(包含起始、结束字符)。**另外，如果起始字符的 ASCⅡ值大于结束字符的 ASCⅡ值，例如，如果你写成“[3-0]”，则会出错，导致匹配失败。

Text

```
city.jpg
city0.jpg
city1.jpg
city2.jpg
city3.jpg
city4.jpg
```

RegEx

```
city[1-3]\.jpg
```

Result

```
city.jpg
city0.jpg
city1.jpg
city2.jpg
```

```
city3.jpg
city4.jpg
```

NOTE：如果要在字符组(“[”“]”内)中匹配“-”，需要使用转义符，写法是“\-”；而在“[”“]”以外，“-”变成了一个普通字符，无需再进行转义。

同样的道理，我们可以写出“[a-z]”来匹配所有的小写字母，“[A-Z]”匹配所有的大写字母，这里就不再举例子了。

现在假设需要在 HTML 中匹配所有的 RGB 颜色，我们知道，在 Web 中，颜色通常表示为诸如“#FF00CC”这样的值，那么结合上面讲述的内容，我们可以像这样使用匹配：

Text

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF">
```

RegEx

```
#[0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-F]
```

Result

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF">
```

NOTE：之前说了，我们默认 Ignore Case 是为 True 的，也就是说忽略大小写。因此，本例中的表达式也可以写作：“#[0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f]”

4.3 反义字符组

有的时候，我们需要匹配“除了某些字符以外”的其他字符，这时候，我们可以使用反义字符组，其语法是：**“[^字符集合]”**

我们在以上面的例子为例，现在我们匹配“city1.jpg、city2.jpg、city3.jpg”以外的文件名。

Text

```
city.jpg
city0.jpg
city1.jpg
city2.jpg
city3.jpg
city4.jpg
citys.jpg
cityss.jpg
```

RegEx

```
city[^1-3]\.jpg
```

Result

```
city.jpg
city0.jpg
city1.jpg
city2.jpg
city3.jpg
city4.jpg
citys.jpg
cityss.jpg
```

注意：这里并没有匹配“city.jpg”。因为，**不管是 普通字符组 还是 反义字符组，它总是 确定一定以及肯定 要匹配一个字符的。**换言之，在本例的模式中，“city”和“.”之间是无论如何需要有一个字符的，而“city.jpg”之间没有，所以它不符合此模式。

5.匹配特殊字符

5.1 匹配元字符

我们先对元字符作一个定义：元字符是在正则表达式中具有特殊含义的字符。如果我们之前已经介绍过的，“.”就是一个元字符，它用来匹配任意单个字符。当我们要匹配字符“.”本身的时候，需要使用“\”来对它进行转义：“\.”

很容易就看出“\”也是一个元字符，它叫做转义符。所以，当我们需要匹配字符“\”的时候，就需要对它进行转义：“\\”。

回想一下之前讲述的字符组，我们知道“[”和“]”也是元字符，当我们需要匹配“[”和“]”字符的时候，需要分别写作：“\[”和“\]”。

举个例子，加入我们需要匹配 “City[0].Name”，却写成了下面这样：

Text

```
City[0].Name = "Xian";
City[0]_Name = "Shanghai";
City0.Name = "Hangzhou";
City0_Name = "Beijing";
```

RegEx

```
City[0].Name
```

Result

```
City[0].Name = "Xian";
City[0]_Name = "Shanghai";
```

```
City0.Name = "Hangzhou";
City0_Name = "Beijing";
```

因为没有对“[”、“]”和“.”进行转义，所以“[0]”被当作字符组来解释，因为字符组中只有一个字符“0”，所以，它仅能匹配一个“0”，所以“City[0]”相当于“City0”。又因为没有对“.”进行转义，所以它可以匹配 它本身 以及 “_”。

正确的写法：City\[0\]\.Name

Text

```
City[0].Name = "Xian";
City[0]_Name = "Shanghai";
City0.Name = "Hangzhou";
City0_Name = "Beijing";
```

RegEx

```
City\[0\]\.Name
```

Result

```
City[0].Name = "Xian";
City[0]_Name = "Shanghai";
City0.Name = "Hangzhou";
City0_Name = "Beijing";
```

5.2 匹配空字符

我想先介绍一下回车换行的由来。通常，当我们在键盘上敲击一下回车键时，不管光标此时在哪里，总是会新起一行，然后将光标位于新行的首位置。这在计算机上看起来一气呵成，用一个符号来表示就 OK 了，可在正则表达式中，以及很多语言中(比如 VBScript)，却被表示成了两个动作，一个叫“回车”(Carriage Return)，一个叫“换行”(Line Feed)，在语言，比如 VBScript 中，就表示成了：Chr(13)&Chr(10)。这与打印机的工作原理有关，大家知道，打印机先于计算机键盘很多年，是键盘的雏形，在打印机上换行时，将进行两个动作：1、将打印头换到下一行；2、将打印头返回到新行的行首位置。也就分别对应了现在的“换行”和“回车”。

在正则表达式中，比较常用的三类空白字符如下表所示：

元字符	匹配描述
\r	回车
\n	换行
\t	Tab 键

这种情况使得书写表达式时，变得稍有不便，例如，如果我们想匹配一个换行的效果，我们需要将表达式写成“\r\n”。然后，我们在 IE 和 Firefox 中用 javascript 分别做个测试，却地发现对于 IE6(NOTE：IE7 我没有试过)来说“\r\n”可以匹配一个换行，而在 Firefox 中，只用

一个“\n”就可以了，使用“\r\n”则无法匹配。

对于我们的测试工具 RegexTester 来说，它也是只使用“\n”来匹配一个换行的效果。

NOTE：不常用的有：\f，换页；\v，垂直 Tab。

5.3 匹配特定字符类型

结合“匹配元字符”和“匹配空字符”这两个小节，我们发现这样一个规律：

- 对于“.”和“[”等来说，**它们本身就是 元字符**，而当给它们前面加上转义字符“\”的时候，它们才代表一个普通字符：“\.”匹配字符“.”，“\[”匹配字符“[”。
- 对于“r”和“n”等来说，**它们本身只是 普通字符**，而只有当加上转义字符“\”的时候(变成了“\r”和“\n”)，它们才代表着元字符：“\r”匹配空字符回车，“\n”匹配空字符换行。

下面将讨论的特定字符类型，就属于上面的第二种情况。

5.3.1 匹配数字类型

元字符	匹配描述
\d	所有单个数字，与 [0-9] 相同
\D	所有非数字，与 [^0-9] 相同

Text

```
City[0].Name = "Xian";
City[1].Name = "Shanghai";
City[a].Name = "Beijing";
```

RegEx

```
City\[\d\]\.Name
```

Result

```
City[0].Name = "Xian";
City[1].Name = "Shanghai";
City[a].Name = "Beijing";
```

NOTE：不管 Ignore Case 是否设置为 True，在这种情况下，“\d”与“\D”总是区分大小写的，下面将介绍的也是一样。

5.3.2 匹配字母、数字、下划线

不管是在程序命名中，还是文件命名中，这一类字符集都是最常见的，那就是：所有大小写

字母、数字 以及 下划线，其正则表达式为“[a-zA-Z0-9_]”。

正则表达式中可以使用“\w ”来代表这一匹配；类似于上一节介绍的，使用“\W”来匹配所有不属于这一字符集的其他字符：“[^a-zA-Z0-9_]”。

元字符	匹配描述
\w	所有单个大小写字母、数字、下划线，与 [a-zA-Z0-9_] 相同
\W	所有单个非大小写字母、非数字、非下划线，与 [^a-zA-Z0-9_] 相同

Text

```
abcde
12345
a1b2c
abcd
1234
```

RegEx

```
\w\d\w\d\w
```

Result

```
abcde
12345
a1b2c
abcd
1234
```

5.3.3 匹配空字符

最后一种就是匹配空字符了，其语法如下表所示：

元字符	匹配描述
\s	所有单个空字符，与 [\f\n\r\t\v] 相同
\S	所有单个非空字符，与 [^\f\n\r\t\v] 相同

匹配多个字符

应该了解，上面所介绍的不管简单也好，复杂也好，都只是匹配单个字符，如果需要匹配一个很长的字符串，而组成这个字符串的每个字符都比较复杂(没有诸如\d 这样的简写方式)，那么，可以想象，一个表达式会多么复杂。

回顾一下匹配 Web 中颜色的例子，我们的正则表达式写法是这样的：“#[0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f]”。应该想到，如果有办法确定匹配的单个字符的个数就好了。本章中，将讨论使用正则表达式来进行多个字符的匹配。

1. 匹配一个或多个

正则表达式中，可以在 单个字符(比如“j”)、字符组(比如“[abcde]”)、特定字符类型(比如“\d”)、单个任意字符(即“.”) 后面加“+”，来表示匹配一个或多个(换言之，至少一个)字符组成的字符串。

拿上面的例子来说，对于 “j+” 有：

Text

```
jimmy、jjjimmy、simmy、immy、enjoy、enjjoy
```

RegEx

```
j+
```

Result

```
jimmy、jjjimmy、simmy、immy、enjoy、enjjoy
```

对于 “[abcde]+”有：

Text

```
abandon、abroad、bbbbb、aaa、edcab、bbb、jskal
```

RegEx

```
[abcde]+
```

Result

```
abandon、abroad、bbbbb、aaa、edcab、bbb、jskal
```

对于其他两个就不再进行说明了，可以自行做测试。

我们现在考虑一个实际的例子，我的电子邮件的写法：jimmy_dev@163.com ，如果没有本章的知识，匹配邮件地址是不可能的，因为，我们无法确定的知道邮件地址的长度。现在，如果我们要匹配这个邮件地址，结合之前的知识 和 本小节的新内容，很容易得出这样的表达式：“\w+@\w+\.\w+”。

Text

```
somebody@domain.net
jimmy_dev@163.com
jimmy.dev@hotmail.com
jimmy_dev@sina.com.cn
```

RegEx

```
\w+@\w+\.\w+
```

Result

```
somebody@domain.net
jimmy_dev@163.com
jimmy.dev.design@hotmail.com
jimmy_dev@sina.com.cn
```

我们发现，这样的匹配并不完整，对于多级域名的匹配不完整。我们回想一下之前的内容，很容易想到，我们需要将“.”也加入到字符组中，最后，我们将“\w+”改写成“[\w\.]+”。

前面说过：**“-”只在字符组“[]”区间内才是一个元字符，在“[]”以外是普通字符。在此处，“.”在“[]”区间以外才是元字符，用以匹配任意单个字符，而在“[]”区间以内，它就变成了一个普通字符。**所以，“[\w\.]”可以简写成“[\w.]”(写成“[\w\.]也不会出错”)。

Text

```
somebody@domain.net
jimmy_dev@163.com
jimmy.dev.design@hotmail.com
jimmy_dev@sina.com.cn
```

RegEx

```
[\w.]+@[\w.]+\.\w+
```

Result

```
somebody@domain.net
jimmy_dev@163.com
jimmy.dev.design@hotmail.com
jimmy_dev@sina.com.cn
```

这次，可以看到能够正确地匹配多级域名了。

2. 匹配零个或多个字符

让我们再次回到刚才匹配城市图片的例子(字符组和反义字符组那一节)，我们知道，不管是使用“city[0-3]\.jpg”还是“city[^0-3]\.jpg”，“city”与“.jpg”之间总是要出现一个字符的，而有的时候，我们允许它们之间可以不出现字符，比如说：现在我要求可以匹配“city.jpg”，那么，该如何完成呢？

正则表达式中，可以在 单个字符(比如“j”)、字符组(比如“[abcde]”)、特定字符类型(比如“\d”)、单个任意字符(即“.”) 后面加“*”，来表示匹配零个或多个字符组成的字符串。可以看出，其使用方法与“+”完全相同。

对于上面的例子，我们可以写出下面的表达式：

Text

```
I like city.jpg、city1.jpg、city365.jpg most.
I suppose cityss.jpg and city_1.jpg are the worst ones.
```

RegEx

```
city\d*\.jpg
```

Result

```
I like city.jpg、city1.jpg、city365.jpg most.
I suppose cityss.jpg and city_1.jpg are the worst ones.
```

我们看到，表达式“city\d*\.jpg”匹配这样的字符串：以“city”开头，后面紧跟零个多个“0-9”的数字字符，然后，以“.jpg”结果。

我们再次回到上面的匹配 Email 的例子中。表达式是这样的：“[\w.]+@[\w.]+\.\w+”，看上去似乎没有问题，但是，我们发现对于 “.jimmy.dev@hotmail.com”这样的地址，它也能够匹配。而我们知道，“.”不应该出现在邮件的第一个字符位置上。

结合本节的知识，我们知道，Email 的起首一定是 **一个或多个** 字母或数字组合，而后面可以是 **零个或多个** 字母数字与“.”的组合。所以，我们修改匹配为如下：

Text

```
.jimmy.dev@hotmail.com is not a valid email format, while jimmy.dev@hotmail.com
is.
```

RegEx

```
\w+[\w.]*@[\w.]+\.\w+
```

Result

```
.jimmy.dev@hotmail.com is not a valid email format, while jimmy.dev@hotmail.com
is.
```

尽管这次对正确的 Email 地址部分进行了匹配，但由于“.jimmy.dev@hotmail.com”是作为一个整体出现，而它并不是一个合法的地址，所以我们通常希望的操作是完全不去匹配它，即认为它并不是一个有效 Email，而不是去匹配它合法的部分。如何解决这个问题，留待后面字符边界说明。

3. 匹配零个或一个字符串

现在加入我们要对单词的正确性进行匹配，对于“flower”这个单词来说，不管是单数“flower”还是复数“flowers”，都是正确的写法，而对于“flowersss”则是不正确的。

使用本章前两章的知识，无法完成这样的匹配：

对于“flowers+”来说，它不能匹配“flower”，因为它只能匹配“flower”后面有**一个或多个“s”**的单词，于是，“flowers”和“flowersss”都可以被匹配。

对于“flowers*”来说，它虽然能够匹配“flower”和“flowers”，但是一样可以匹配“flowersss”。

正则表达式中，使用“?”来匹配零个或一个字符。其使用方式与“+”和“*”相同。

对于本例而言，其使用方法如下：

Text

```
These flowers are very beautiful, hope you like them.
This flower is so beautiful, seems like smiling to you.
```

RegEx

```
flowers?
```

Result

```
These flowers are very beautiful, hope you like them.
This flower is so beautiful, seems like smiling to you.
```

很明显，**本章的所讲述的多字符匹配中，“+”、“*”、“?”都是元字符，如果要对它们进行匹配，需要使用“\”进行转义：“\+”、“*”、“\?”。**

4. 匹配指定数目字符

尽管“?”、“+”、“*”这三个元字符的出现解决了很多问题，但是，它们并不完善：1. 没有办法指定最多匹配多少个字符，比如说，我们要匹配手机号码，那么应该是 11 个数字，而“+”和“*”会匹配尽可能多的数字，不管是 17 个还是 18 个都认为是正确的。2. 没有办法指定最少匹配多少个字符，“+”、“*”、“?”，所能提供的最少字符，要么零个，要么一个。

4.1 匹配固定数目的字符

正则表达式中，可以在 单个字符(比如“j”)、字符组(比如“[abcde]”)、特定字符类型(比如“\d”)、单个任意字符(即“.”) 后面加“{数字}”，来表示匹配零个或多个字符组成的字符串。

例如：使用“\d{3}”，可以匹配从 000 到 999，这 1000 个数。而使用“a{6}”，则可以匹配“aaaaaa”(也只能匹配它，因为“a”是固定字符)。

现在我们考虑一个更复杂的例子，假如我们要匹配手机号码，那么它的规则是：首位为“1”，第二位为“3 或者 5”，后面 9 位任意数字。那么它的匹配应该是这样的：

Text

```
13991381592 is a valid telephone number.
1369188351 is too short.
The second number of 17991381592 is out of range.
The character “d” in 13d91381592 is not allowed.
The first character in 63991381592 should be 1 constantly.
```

RegEx

```
1[35]\d{9}
```

Result

```
13991381592 is a valid telephone number.
1369188351 is too short.
The second number of 17991381592 is out of range.
The character “d” in 13d91381592 is not allowed.
The first character in 63991381592 should be 1 constantly.
```

4.2 匹配区间以内数目的字符

我们再次考虑 000-999 的匹配：“\d{3}”，尽管它没有错，但它只能匹配精确地匹配 3 位：000、001、002 … 100、101 … 999。而通常，我们需要对于 0、10、99 这样的数也能够匹配，这时，就需要指定可以匹配 1 到 3 位的数字。

正则表达式中，使用“{最小数目，最大数目}”的语法来实现，它的使用方式与上一节介绍的匹配固定数目字符的语法相同。

Text

```
0、10、111、001、789、1234 are all matched numbers.
0d、0h、1xx、99. are not matched.
```

RegEx

```
\d{1,3}
```

Result

```
0、10、111、001、789、1234 are all matched numbers.
0d、0h、1xx、99. are not matched.
```

NOTE：我们发现了两个问题：1、我们不希望 “1234” 被匹配，然而，它被分成两部分“123”和“4”进行了匹配，如何不让它被匹配；2、为什么要被分成 123 和 4，而不是 1 和 234？我们留待后面讨论。

注意两个特例：

- 最小数目可以是 0，所以 “{0,1}”，相当于 “?”。

- 如果不限制最大数目，可以将最大数目设为空，所以“\d{1,}”相当于“+”；而“{0,}”相当于“*”。

“{”和“}”也是元字符，当我们需要对它们进行匹配的时候，使用“\”进行转义：“\{”和“\}”。

5.贪婪匹配和惰性匹配

5.1 贪婪匹配、惰性匹配概述

我们首先从字面意思上来理解一下贪婪匹配和惰性匹配：

- **贪婪匹配(greedy)**：它会匹配尽可能多的字符。它首先看整个字符串，如果不匹配，对字符串进行收缩；遇到可能匹配的文本，停止收缩，对文本进行扩展，当发现匹配的文本时，它不着急将该匹配保存到匹配集合中，而是对文本继续扩展，直到扩展完整个字符串，然后将前面最后一个符合匹配的文本(也是最长的)保存起来到匹配集合中。所以说它是贪婪的。
- **惰性匹配(lazy)**：它会匹配尽可能少的字符，它从第一个字符开始找起，一旦符合条件，立刻保存到匹配集合中，然后继续进行查找。所以说它是懒惰的。

光看上面的定义，我们很难有一个生动的认识，现在假设我们要匹配下面 和 之间的文本。为了做演示，尽管不符合 HTML 的定义，我们再加入一段和</c>之间的文本：

Text

```
Jimmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>.
```

RegEx

```
<b>.*</b>
```

Result

```
Jimmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>.
```

这样匹配显然不是我们的初衷，它仅找到了一个匹配，而通常情况下，我们希望得到的是 junior 和 living 两个匹配。

解决的办法，就是上面说到的惰性匹配，它的语法如下表所示：

贪婪匹配	惰性匹配	匹配描述
?	??	匹配 0 个或 1 个
+	+?	匹配 1 个或多个
*	*?	匹配 0 个或多个
{n}	{n}?	匹配 n 个
{n,m}	{n,m}?	匹配 n 个或 m 个
{n,}	{n,}?	匹配 n 个或多个

对于本例，当我们使用惰性匹配时：

Text

```
Jimmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>.
```

RegEx

```
<b>.*?</b>
```

Result

```
Jimmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>.
```

现在，请对照着之前对它们的定义，并结合下面对匹配过程的分析，来理解 贪婪匹配 和 惰性匹配的匹配过程：

5.2 贪婪匹配的匹配过程：

```
Jimmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>.//不匹配，收缩
immy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>. //不匹配，收缩
mmy is a <b>junior</b> developer <b>living</b> in <b>xi'an</c>. //不匹配，收缩
...          //中间略
<b>junior</b> developer <b>living</b> in <b>xi'an</c>. // 找到可能匹配的，扩展
<b>junior</b> developer <b>living</b> in <b>xi'an</c>.// 找到可能匹配的，扩展
<b>junior</b> developer <b>living</b> in <b>xi'an</c>. // 找到可能匹配的，扩展
...          // 中间略
<b>junior</b> developer <b>living</b> in <b>xi'an</c>. //找到一个匹配，但是并不保存到结果集中，而是继续进行扩展
<b>junior</b> developer <b>living</b> in <b>xi'an</c>.
...          //中间略
<b>junior</b> developer <b>living</b> in <b>xi'an</c>. //又找到匹配，继续扩展
...          //中间略
<b>junior</b> developer <b>living</b> in <b>xi'an</c>.  //字符串结束，将前面找到的最后一个匹配 <b>junior</b> developer <b>living</b> 保存到匹配结果集中
```

5.3 惰性匹配的匹配过程

```
J        // 不匹配，继续
Ji       // 不匹配，继续
...      // 中间略
Jimmy is a <            // 找到可能匹配的字符，继续
Jimmy is a <b           // 找到可能匹配的字符，继续
...      // 中间略
Jimmy is a <b>junior</b>    //找到匹配，保存到结果集中，继续进行剩下的文本。

d        // 不匹配，继续
de       // 不匹配，继续
```

```
developer <                    // 找到可能的匹配，继续
developer <b                   // 找到可能的匹配，继续
...      // 中间略
developer <b>living</b>     //找到匹配，保存到结果集中，继续进行剩下的文本。

i        //不匹配，继续
in       //不匹配，继续
in <         // 找到可能的匹配，继续
in <b        // 找到可能的匹配，继续
...      // 中间略
in <b>xi'an</c       // 匹配失败，继续找
in <b>xi'an</c>      // 不匹配，继续
in <b>xi'an</c>.     // 字符串结束，匹配结束。一共找到了两个匹配<b>junior</b> 和
<b>living</b>
```

我们回顾一下上面 “\d{1,3}” 匹配数字的例子，对于“1234”，当我们使用“\d{1,3}”时，进行的是贪婪匹配，它首先找到“123”(因为“1234”不符合)，之后的“4”也符合，所以，找到的匹配是“123”和“4”。

当我们使用“\d{1,3}?”匹配上面的例子，对于“1234”，这次是惰性匹配。首先，发现“1”符合，将“1”保存到匹配集合中；随后，依次发现“2”、“3”、“4”符合，并依次保存到结果集中，最后，我们得到了四个匹配“1”、“2”、“3”、“4”。

5.4 值得注意的两个匹配模式

现在请回顾一下上面贪婪、惰性匹配语法的表，有两个匹配模式比较有意思：

一个是“{n}”，对于这种形式的匹配，由于它精确地要求匹配 n 个字符，所以无所谓贪婪还是惰性，尽管“{n}?”也是正确的匹配写法，但它的匹配结果总是与“{n}”相同。

还有一个就是“??”，它看上去比较古怪且不好理解，因为通常我们使用贪婪匹配的时候都是匹配多个，也就是“*”或者“+”之类的匹配，而这里是 0 个或 1 个，它的贪婪与惰性匹配又是如何呢？我们还是看范例来说明：

Text

```
These flowers are for you, my beloved.
```

RegEx

```
flowers?
```

Result

```
These flowers are for you, my beloved.
```

我们来分析一下，在这个匹配中，匹配是这样的：首先需要匹配“flower”字符串，然后，可以有 0 个或者 1 个“s”，按上面讲述的，贪婪匹配匹配尽可能多的，所以 flowers 被匹配了。

再来看看“??”这种匹配。

Text

```
These flowers are for you, my beloved.
```

RegEx

```
flowers??
```

Result

```
These flowers are for you, my beloved.
```

这次只匹配了“flower”，我想现在已经不用过多解释，你对这个怪异的匹配语法已经明白了。惰性匹配发现“flower”满足匹配 0 个的条件，于是将它保存到匹配结果集，然后重新进行匹配查找，直到字符串结束。

匹配边界

回忆一下刚才邮件地址的匹配，表达式“\w+[\w.]*@[\w.]+\.\w+”匹配了不合法的邮件地址“.jimmy_dev@163.com”中合法的部分，而通常，我们希望它完全不去匹配它。这时，就有一个匹配边界的问题，我们希望：“\w+”必须出现在字符串的首位置，也就是字符串的边界。

邮件的例子稍显复杂，我们再看一个更简单的情况：

Text

```
The cat scattered its food all over the room.
```

RegEx

```
cat
```

Result

```
The cat scattered its food all over the room.
```

可见，通常情况下，我们只希望匹配 cat ，而不希望匹配 scattered 中出现的 cat。

1. 匹配单词边界

正则表达式中，可以在 字符 前加“\b”，来匹配其 后面 的字符位于字符串首位的字符。

NOTE：以后提到 字符，指：单个字符(比如“j”)、字符组(比如“[abcde]”)、特定字符类型(比如“\d”)、转义过的特殊字符“\[” 或者 单个任意字符(即“.”)。

我们来看一个最简单的例子：

Text

```
Attention: Zhang is the first name of JimmyZhang.
```

RegEx

```
\bz
```

Result

```
Attention: Zhang is the first name of JimmyZhang.
```

如同定义的那样：**“\bz”匹配所有位于字符串首位的“z”，而对位于字符串中间的“z”则不进行匹配（“JimmyZhang”中的“Z”）。**不过，我们很少如此简单地使用边界匹配，而是将它加在一个表达式前面。

现在回到本章开头的例子，修改表达式，让它只匹配单词 cat：

Text

```
The cat scattered its food all over the room.
```

RegEx

```
\bcat
```

Result

```
The cat scattered its food all over the room.
```

这一次，匹配正确，有了刚才单个字符的例子，现在这个表达式很好理解了：**“\b”只规定了“c”这个字符必须出现在字符串首位，接下来需要出现字符“a”、“t”。**这两个字符的匹配与“\bc”无关，它们属于固定字符匹配的范畴。关于固定字符匹配，可见本文第一章。

但是这个匹配还存在问题，如果我们将上面的例子稍微做下修改，将“scattered”单词分成“s”和“cattered”：

Text

```
The cat s cattered its food all over the room.
```

RegEx

```
\bcat
```

Result

```
The cat s cattered its food all over the room.
```

可以看到，我们仅仅将“scattered”拆写成“s”和“cattered”，就又错误地匹配了“cattered”，因为它符合我们前面所说的匹配规则。

可见，如果需要仅匹配“cat”这个单词，我们还需要规定“t”必须出现在单词的末尾。

正则表达式中，可以在 字符 后加“\b”，来匹配其 前面 的字符位于字符串末位的字符。

Text

```
The cat s cattered its food all over the room.
The cat scat tered its food all over the room.
```

RegEx

```
cat\b
```

Result

```
The cat s cattered its food all over the room.
The cat scat tered its food all over the room.
```

NOTE：注意上面两个句子中“scattered”单词拆分位置不同。

可以看到，“cat\b” 不能匹配“cattered”之中的“cat”，却匹配了 “scat” 中的“cat”。这个匹配的过程就不再说明了，与上面类似。

显然，为了精确地匹配“cat”，我们需要在前后都加上字符边界，“\bcat\b”。

Text

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
```

RegEx

```
\bcat\b
```

Result

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
```

如果我简单地说：“\bcat\b”匹配完整地一个“cat”单词，那相当于什么也没说。我们综合上面讲述的，来分析一下它是如何匹配的：首先，“c”必须出现在字符串首位；然后，紧跟一个“a”字符；最后，它必须以“t”结束。

2. 边界及其相对性

2.1 边界的定义

讲了这么多，还漏掉了一个重要的内容：究竟什么才算边界？

通常情况下，以 空格、段落首行、段落末尾、逗号、句号 等符号作为边界，值得注意的是，分隔符“-”也可以作为边界。

Text

```
The cat s-cat-tered its food all over the room.
```

RegEx

```
\bcat\b
```

Result

```
The cat s-cat-tered its food all over the room.
```

这是什么原因呢？其实很好理解，从“-”的字面意思：分隔符，大致就可以想到了。实际上，在英语中，它是用来做单词分隔的。

NOTE：这里有个重要的搜索引擎优化常识，大家注意到本文档的命名，我采用的是：Regular-Expression-Tutorial.pdf，为什么不用下划线分隔，命名成 Regular_Expression_Tutorial.pdf 呢？ 因为当搜索引擎看到“-”的时候，会把它视为一个空格“ ”，而看到下划线“_”的时候，会把它视为空字符“”，实际上，下划线的正确叫法是“连字符”。于是，当我命名为 Regular-Expression-Tutorial.pdf 时，搜索引擎看到的是：Regular Expression Tutorial.pdf，而当我命名成 Regular_Expression_Tutorial.pdf 时，搜索引擎看作 RegularExpressionTutorial.pdf 。

可以看出，正则表达式在字符边界问题上 对“-”的处理方式 与 搜索引擎相同。

2.2 边界的相对性

请牢牢记住边界的这个特点：

- 当你对一个普通字母，比如“s”，设定边界的时候，它的边界是诸如空格、分隔符、逗号、句号等。
- 当你对一个边界，比如分隔符“-”或者“,”等，设定边界的时候，它的边界是普通字母。

我们先看第一种情况：

Text

```
aaaaxaaaa
aaa-x-aaa
```

RegEx

```
\bx\b
```

Result

```
aaaaxaaaa
aaa-x-aaa
```

这个和我们上面将的例子相似，和我们预想的一样，下面 x 被匹配了，因为“-”可以作为边界。

我们再看另一个例子：

Text

```
aaaa,aaaa
aaa-,-aaa
```

RegEx

```
\b,\b
```

Result

```
aaaa,aaaa\
aaa-,-aaa
```

与上面唯一不同的是：这次我们匹配逗号“,”，而它本身也是一个边界，结果与上面完全相反。可见，对于“,”而言，它的边界是一个普通字母。

边界的相对性是很重要的，因为我们很多时候需要匹配诸如“<”这样的字符。

3. 匹配非单词边界

和上面 匹配特定类型字符有些相似，有了“\b”，自然有“\B”，它用来匹配不在边界的字符。

我们继续拿上面的例子做示范，来看看“\Bcat”匹配的效果：

Text

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
```

RegEx

```
\Bcat
```

Result

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
```

它的匹配规则是这样的：字符“c”必须出现，但是不能位于字符串首位；随后跟两个固定字符“a”和“t”。

我们再对上面例子进行扩展：

Text

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
The cat scattered its food all over the room.
```

RegEx

```
\Bcat\B
```

Result

```
The cat scat tered its food all over the room.
The cat s cattered its food all over the room.
The cat scattered its food all over the room.
```

看了这么多例子，现在不用我讲你也应该明白了：首先，必须出现字符“c”，且**不能**位于字符串首位；接着，“c”后面必须出现字符“a”；最后，必须出现字符“t”，且它**不能**位于字符串的末尾。

3. 匹配文本边界

有的时候，我们想要匹配的字符串必须位于全部文本的首位，比如说 XML 文件的声明“<?xml version="1.0" encoding="UTF-8" ?>”；有的时候，需要匹配的字符串位于全部文本的末尾，比如</html>。对于这种匹配，上面介绍的单词边界匹配就无能为力了。

3.1 匹配文本首

在正则表达式中，可以在 匹配模式 的第一个字符前添加 “^”，以匹配 满足模式且位于全部文本之首的字符串。可以将它的匹配方式理解成这样：1、假设不存在“^”，进行一个正常匹配，将所有匹配的文本保存到匹配集合中；2、在匹配集合中寻找位于 所搜索的文本 首位的匹配；3、从匹配集合中删除其他匹配，仅保留该匹配。

我们依然是从简单的例子看起：“^city\d?\.jpg”。

Text

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

RegEx

```
^city\d?\.jpg
```

Result

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

按照之前说的，它的匹配过程是这样：

1. 假设匹配模式是 city\d?\.jpg，对文本进行匹配。
2. 一共找到 6 个符合模式的文本：第一行 和 第二行的 city.jpg, city1,jpg 及 city9.jpg
3. 从所有匹配的文本中筛选出位于文本首位的匹配文本：即第一行的 city.jpg，删除所有其他匹配。

这里有个值得注意的地方，如果我们在第一行的 city.jpg 中添第几个空格，就破坏了这个匹配。

Text

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

RegEx

```
^city\d?\.jpg
```

Result

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

可见，没有找到任何匹配，所以，我们进行文本边界匹配时，通常还需要添加对空字符的处理：

Text

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

RegEx

```
^\s*city\d?\.jpg
```

Result

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

3.2 匹配文本末

有匹配文本首的语法，自然有匹配文本末的语法。

在正则表达式中，可以在 匹配模式 的最后一个字符后添加 “$”，以匹配 满足模式且位于全部文本之末的字符串。

它的匹配方式 与 匹配文本首“ˆ”相似，这里就不再详细说明了，只给出一个例子：

Text

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

RegEx

```
city\d?\.jpg\s*$
```

Result

```
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
city.jpg、city1.jpg are all beautiful pictures except city9.jpg
```

回顾下之前介绍的，可以看出：“\b”和“\B”是对 匹配模式(表达式) 中某个字符出现的进行位置(单词首位还是末位)进行限制。“^”和“$” 是对 整个待搜索文本 的 匹配模式(表达式) 出现位置(文本首位还是文本末位)进行限制。它们的关系是一小一大。

匹配子模式

可以看出，我们之前介绍的所有匹配模式(例如“+”、“*”、“{n,m}”)，都是针对于某种 **单个字符** 的。

考虑这样一个例子：我们需要将 HTML 中两个或以上的“
”、“<br/>”、“<br>”全部替换成一个“
”。按照之前的例子，我们只能写出这样的表达式：

Text

```
This is the first line.<br>
This is the second line.<br><br/><br />
This is the third line.<br>>>>>>
```

RegEx

```
<br\s*/?>{2,}
```

Result

```
This is the first line.<br>
This is the second line.<br><br/><br />
This is the third line.<br>>>>>>
```

可以看到，匹配结果并不是我们想要的，这是因为 “{2,}”限制的是它之前的单个字符，在本例中也就是“>”的出现次数，而我们希望的，是整个“
”、“<br/>”或“
”。

1.子模式

在正则表达式中，可以使用“(”和“)”将模式中的子字符串括起来，以形成一个子模式。将子模式视为一个整体时，那么它就相当于一个单个字符。

就本例而言，我们希望子模式就是 “<br\s*/>”，按上面的定义，我们重新写模式：

Text

```
This is the first line.<br>
This is the second line.<br><br/><br />
This is the third line.<br>>>>>
```

RegEx

```
(<br\s*/?>){2,}
```

Result

```
This is the first line.<br>
This is the second line.<br><br/><br />
This is the third line.<br>>>>>
```

这次匹配了正确的文本，我们可以将匹配过程理解成这样：子模式“(<br\s*/?>)”首先匹配所有“
”、“<br/>”或“
”；然后，将每一个匹配结果视为一个整体(相当于单个字符)；接着，匹配这个整体连续出现两次或以上的文本。

2. “或”匹配

有的时候，我们要取消某段文字中的加粗、斜体等效果，我们想匹配所有的“”、“”或者 “”、“”，然后把它们替换成空，仅利用之前的知识，我们只能进行两次匹配和替换，一次是“</?b>”，一次是“</?i>”。

在正则表达式中，可以使用“|”将一个表达式拆分成两部分“reg1|reg2”，它的意思是：匹配所有符合表达式 reg1 的文本 或者 符合表达式 reg2 的文本。

对于本节提出的问题，可以这样进行解决：

Text

```
The <b>text of</b> this row is bold.
The <i>text of</i> this row is italic.
```

RegEx

```
</?i>|</?b>
```

Result

```
The <b>text of</b> this row is bold.
The <i>text of</i> this row is italic.
```

3. 在子模式中使用“或”匹配

从上面的定义应该可以看出，“|”分隔的是整个表达式，而有的时候，我们希望分隔的是一个表达式的一部分，比如说，我们想要匹配 “1900”到“2099”的所有年份。

Text

```
1932 is supposed to be matched as a whole, but it is matched only part of it.
2055 is mathced in the right way.
3019 is out of range, but it's still matched partly.
```

RegEx

```
19|20\d{2}
```

Result

```
1932 is supposed to be matched as a whole, but it is matched only part of it.
2055 is mathced in the right way.
3019 is out of range, but it's still matched partly.
```

可以看到，表达式“19|20\d{2}”要么匹配“19”，要么匹配“20\d{2}”。而我们希望的是匹配“19\d{2}”或者“20\d{2}”，当然，我们可以改写上面的表达式为“19\d{2}|20\d{2}”来完成，但是，利用本章所讲述的子模式，可以更加简洁地完成这个过程。

Text

```
1932 is supposed to be matched as a whole, but it is matched only part of it.
2055 is mathced in the right way.
3019 is out of range, but it's still matched partly.
```

RegEx

```
(19|20)\d{2}
```

Result

```
1932 is supposed to be matched as a whole, but it is matched only part of it.
2055 is mathced in the right way.
3019 is out of range, but it's still matched partly.
```

这次，我们得到了想要的结果，使用子模式可以简化匹配表达式。

NOTE：有点类似于数学中的提取公因式：2*3 + 7*3 = (2+7)*3 --> 19\d{2}|20\d{2} = (19|20)\d{2}

4. 嵌套子模式

子模式可以再继续嵌套子模式，产生更加功能强大的匹配能力。比如，我们要匹配 1900 年 1 月 1 日 到 2000 年 1 月 1 日 除过闰年外的所有正确日期。

我们先对这个匹配模式做一个分析：

1. 首位可以是 19 也可以是 20； `Reg: 19|20`
2. 当是 19 的时候，后面可以是 00 到 99 中任意数字； `RegEx: 19\d{2}|20`
3. 当是 20 的时候，只能匹配 00； `Reg: 19\d{2}|2000`

4. 月份可以是 1 到 9，或者 10 到 12；`Reg：(19\d{2}|2000)-([1-9]|1[0-2])`

因为天数与月份相关，所以将（[1-9]|1[0-2]）拆分为下面三个子模式：

5. 当月份是 2 的时候，天数是 28； `Reg：2-([1-9]\b|1\d|2[0-8])`
6. 1、3、5、7、8、10、12 月，天数是 31；`Reg：([13578]|1[02])-([1-9]\b|[12]\d|3[01])`
7. 4、6、9、11 月，天数是 30； `Reg：([469]|11)-([1-9]\b|[12]\d|30)`

NOTE：注意上面日期部分的匹配，分成了两部分，月和日；对于月来说，如果我们要匹配大月(31 天的月)，写法是：[13578]|1[0-2]；而日期部分，比如说要匹配 31 天，它又由三部分组成：[1-9]表示 1 号到 9 号；[12]\d 表示 10 号到 29 号；3[01]表示 30 号到 31 号。

还有个地方需要注意：单词边界问题，如果你这样写表达式：`2-([1-9]|1\d|2[0-8])`，对于 2-29 这样不应该匹配的日期，会匹配它合法的部分 `2-2`9，因为`2-2`满足`2-[1-9]`。回顾下我之前讲述的内容，我们还必须规定，当天数是个位数时，它必须处于单词边界`[1-9]\b`。

组合一下，得到的月份和天数的模式是：

```
2-([1-9]\b|1\d|2[0-8])|([13578]|1[02])-([1-9]\b|[12]\d|3[01])|([469]|11)-([1
-9]\b|[12]\d|30)
```

再结合上面年的部分，得到最终的结果：

```
(19\d{2}|2000)-(2-([1-9]\b|1\d|2[0-8])|([13578]|1[02])-([1-9]\b|[12]\d|3[01]
)|([469]|11)-([1-9]\b|[12]\d|30))
```

Text

```
These dates are matched: 1900-1-1、1928-2-28、1931-11-30、2000-1-1、1999-10-30
These dates are not matched: 1900-1-32、1928-2-29、2000-01-1、1982-12-08
```

RegEx

```
(19\d{2}|2000)-(2-([1-9]\b|1\d|2[0-8])|([13578]|1[02])-([1-9]\b|[12]\d|3[01]
)|([469]|11)-([1-9]\b|[12]\d|30))
```

Result

```
These dates are matched: 1900-1-1、1928-2-28、1931-11-30、2000-1-1、1999-10-30
These dates are not matched: 1900-1-32、1928-2-29、2000-01-1、1982-12-08
```

后向引用

1. 理解后向引用

我们还是一如既往地从最简单的开始。假设我们要进行这样一个匹配：找出下面文本中所有重复的单词，以便日后进行替换。

```
Is the cost of of gasline going up?
```

我们看到：“of of”重复了，我们需要找出它：

Text

```
Is the cost of of gasline going up?
```

RegEx

```
of of
```

Result

```
Is the cost of of gasline going up?
```

很显然，匹配结果满足了我们的要求，在这里使用全字匹配是为了后面好说明。

现在，如果 up 也重复出现了，句子变成这样：

```
Is the cost of of gasline going up up?
```

我们就需要改写表达式成这样：

Text

```
Is the cost of of gasline going up up?
```

RegEx

```
(of|up) (of|up)
```

Result

```
Is the cost of of gasline going up up?
```

NOTE：我们可以使用更简洁的表达式：`((of|up)\b ??){2}`，但是为了后面好说明，这里我们还是使用全字匹配。

关于这个表达式，首先记住，\b 只是对边界进行限制，不匹配任何字符。

如果写做`((of|up)\b ){2}`，则无法匹配“up up”，因为它要求up后面必须出现一个空格“ ”，而本句中，up后面紧跟了一个问号；

如果写成`((of|up)\b ?){2}`，因为是贪婪匹配，如果of后出现空格，就会匹配之。两个of的匹配将变成“`of of `”。通常，我们会替换它成一个“of”，这样，就会出现“ofgasline”的情况。

最终，我们把贪婪匹配改成惰性匹配：((of|up)\b ??){2}

可以看到，对这个句子来说，这样匹配没有问题。但是我们匹配的文本往往比这个复杂，让我们给上面的文本再添一句话。

Text

```
Is the cost of of gasline going up up?
```

```
Look up of the TV, your mobile phone is there.
```

RegEx

```
(of|up) (of|up)
```

Result

```
Is the cost of of gasline going up up?
Look up of the TV, your mobile phone is there.
```

不幸的是，对于下面不应该匹配的“up of”也进行了匹配。而我们需要的是这样一种匹配：当前面是“of”的时候，后面跟的也是“of”；当前面是“up”时，后面跟的也是“up”，只有这样的情况才去匹配它。换言之，后面所需要匹配的内容是前面的一个引用。

正则表达式中，使用“\数字”来进行后向引用，数字表示这里引用的是前面的第几个子模式。

我们回头在看本节第一个例子，我们也可以这样写：

Text

```
Is the cost of of gasline going up?
```

RegEx

```
(of) \1
```

Result

```
Is the cost of of gasline going up?
```

这里，表达式“(of) \1”使用了后向引用，意思是：“\1”代表前面第一个匹配了的子模式的内容，对于本例，因为前面子模式只可能匹配“of”，那么“\1”等价于“of”，整个表达式相当于“of of”。

现在，我们使用后向引用，对刚才出现“up of”问题的表达式加以纠正：

Text

```
Is the cost of of gasline going up up?
Look up of the TV, your mobile phone is there.
```

RegEx

```
(of|up) \1
```

Result

```
Is the cost of of gasline going up up?
Look up of the TV, your mobile phone is there.
```

这一次，我们获得了预期的效果，“(of|up) \1”的含义是：如果前面 子模式 1 匹配了“of”，那么“\1”就代表“of”；如果 子模式 1 匹配了 “up”，那么“\1”就代表“up”，整个表达式相当于“of of|up up”。

在我们所讨论的简单情况下，已知的重复单词是确定的，我们可以找到后向引用的等价表达式。但通常情况下，我们不知道哪些单词重复，这时候，就只能使用后向引用来完成：

Text

```
Is the cost of of gasline going up up?
Look up of the TV, your mobile phone is there.
You are the best of the the best.
```

RegEx

```
(w+) \1
```

Result

```
Is the cost of of gasline going up up?
Look up of the TV, your mobile phone is there.
You are the best of the the best.
```

这次，我想你应该可以想出“(w+) \1”所代表的含义，就不再赘述了。

2. 后向引用的一个常见应用

匹配重复单词是后向引用的一个较为常见的应用，还有一个应用是匹配有效的 HTML 标记：

Text

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

RegEx

```
<h([1-6])>.*?</h\1>
```

Result

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

可以看到，它只匹配了符合 HTML 语法的文本，对于前后不一致的文本没有进行匹配。

3. .Net 中的后向引用

经常写正则表达式的朋友们应该都有体会：对于复杂的匹配，尤其是子模式嵌套子模式的情况，常常判断不清楚后向引用匹配的到底是前面哪个子模式，比如：我们写的“\4”，期望它去匹配第 4 个子模式，但往往它匹配了第 3 个或者第 5 个。

如果能有名字命名子模式就好了，这样，在后向引用中，我们可以使用这个名字，而不用根据数字去判断。

在 .Net 中使用正则表达式进行查找时，给子模式命名的语法是：?<name>，后向引用的语法是：\k<name>。

我们改写前面的范例：

Text

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

RegEx

```
<h(^<sub>[1-6])>.*?</h\k<sub>>
```

Result

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

可以看到，我们给子模式([1-6])起了个名字 sub，然后在后向引用中使用了它“\k<sub>”，“\k<sub>”就相当于“\1”。

文本替换

1. 使用后向引用进行文本替换

迄今为止，我们所了解的正则表达式都是用于匹配文本，而如果我们对匹配了的文本不能进行替代，那它也就没有什么用了。

所以，正则表达式的三部曲应该是：1、查找；2、引用匹配了的文本(后向引用)；3、有选择地替换文本。

需要注意的是：大部分语言的正则表达式实现，在查找中，使用后向引用来代表一个子模式，其语法是“\数字”；而在替换中，其语法是“$数字”。

1.1 高亮显示文本

假设我需要高亮显示所有 h1 中的文本，我们就可以使用后向引用来完成：

Text

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

RegEx

```
<h1>(.*?)</h1>
```

Replace

```
<h1 style="background:#ff0">$1</h1>
```

Result

```
<h1 style="background:#ff0">This is a valid header</h1>
<h2>This is not valid.</h3>
```

在这个例子中，“$1”代表了<h1></h1>之间的文本。

1.2 替换电话号码格式

我们的电话格式通常都是：(区号)电话，比如说：(029)8401132；现在假设我们要求把文本中所有的电话格式都改为：029-8401132，我们可以这样做：

Text

```
(020)82514769
(021)83281314
(029)88401132
```

RegEx

```
\((\d{3})\)(\d{8})
```

Replace

```
$1-$2
```

Result

```
020-82514769
021-83281314
029-88401132
```

如果这篇文章你是从头看到这里，相信这个表达式对你来说没有任何难度，需要留意的是这里对元字符“(”和“)”进行了转义，并且，在替换结果中，我们要求它不出现。

2. .Net 中的文本替换

同后向引用在查找时的情况类似，在 .Net 中，在替换时也可以对后向引用进行命名。

在 .Net 中使用正则表达式进行替换时，给子模式命名的语法是：?<name>，后向引用的语法是：${name}。

我们修改高亮显示 h1 文本的范例：

Text

```
<h1>This is a valid header</h1>
<h2>This is not valid.</h3>
```

RegEx

```
<h1>(?<sub>.*?)</h1>
```

Replace

```
<h1 style="background:#ff0">${sub}</h1>
```

Result

```
<h1 style="background:#ff0">This is a valid header</h1>
<h2>This is not valid.</h3>
```

预查和非获取匹配

1. 理解非获取匹配

假设我们有下面这样一段文本，而我们想要获取的 Windows 的所有版本，我们可以这样写：

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows [\w.]+\b
```

Result

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
```

```
Windows Vista is the Latest version of the family.
```

这时，如果我们想将所有的 Windows，全部换成简写 Win，并去掉 Windows 与 版本号 之间的空格，我们则需要使用后向引用：

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows ([\w.]+\b)
```

Replace

```
Win$1
```

Result

```
Win1.03 and Win2.0 fisrt Released in 1985 and 1987 respectively.
Win95 and Win98 are the successor.
Then Win2000 and WinXp appeared.
WinVista is the Latest version of the family.
```

我们首先查看一下表达式的区别，**为了要使用后向引用，我们用“(”和“)”把“[\w.]+\b”包起来，使它成为一个子模式。**我们知道，只有这样，才可以用 $1 去引用它，这里，我们发现使用子模式的一个作用：**系统会在幕后将所有的子模式保存起来，以供后向引用使用(包含查找时的后向引用 和 替换时的后向引用)。**

而很多时候，我们添加一个子模式，并不是为了在后向引用中获取它，我们或许是出于匹配需要，或许简单的只是为了使表达式更清晰。

正则表达式中，可以在子模式内部前面加“?:”来表示这个子模式是一个 非获取匹配，非获取匹配不会被保存，不能在后向引用中获取。

对于本例，我们来测试一下：

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows (?:[\w.]+\b)
```

Replace

```
Win$1
```

Result

```
Win$1 and Win$1 fisrt Released in 1985 and 1987 respectively.
Win$1 and Win$1 are the successor.
Then Win$1 and Win$1 appeared.
Win$1 is the Latest version of the family.
```

我们看到，由于子模式没有被保存，所以“$1”被当作一个普通字符进行了处理。

而如果我们只是进行查找匹配，不进行替换，那么它们返回的效果相同：

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows (?:[\w.]+\b)
```

Result

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

可见，所谓 非获取匹配，意思就是说它只进行匹配，并不保存结果供以后引用。

2. 正向预查

看到这里，你可能觉得 非获取匹配 没有什么实际用途。

现在，我们假设需要仅匹配 Windows，不匹配后面的版本号，并且要求 Windows 后面的版本号只能是 数字类型，换言之，XP 和 Vista 不能被匹配，该如何做？

按照现有的知识，我们大概只能写出这样的表达式：

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows [\d.]+\b
```

Result

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

的确，它是匹配了 数字类型 版本的 Windows，但是，它不符合我们的要求。因为，我们要求不能匹配版本号。

在正则表达式中，可以使用 **正向预查** 来解决这个问题。本例中，写法是：“Windows(?=[\d.]+\b)”。

它的语法是在 子模式内部 前面加“?=”，表示的意思是：首先，要匹配的文本必须满足此子模式 前面 的表达式(本例，“Windows ”)；其次，此子模式不参与匹配。

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows( ?=[\d.]+\b)
```

Result

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

这次，你大概了解了 “非获取匹配” 这五个汉字的含义，它们仅仅起一个限制作用，不参与匹配。**你可以将 正向预查 理解成为自定义的边界(\b)，这个边界位于 表达式末。**

反言之，你可以将位于表达式末的 \b 理解成非获取匹配的一个特例：(?=[,.\r\n<>;\-])。注意，这里我没有写全边界符号。

Text

```
aaaax-aaa
aaaaxaaaa
```

RegEx

```
x(?=[,.\r\n<>;\-])
```

Result

```
aaaax-aaa
aaaaxaaaa
```

你也可以这样理解上面的匹配过程：

1. 先进行普通匹配：Windows ([\d.]+\b)
2. 然后从匹配文本中将 子模式 内的文本排除掉。

3. 反向预查

在上面的例子中，我们知道 正向预查 类似于自定义的 位于文本末 的字符边界。那么自然应该有位于文本首的情况，比如说，我们要匹配下面文本中属于 CNY 的金额：

Text

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
CNY: 23.78
```

RegEx

```
CNY: \d+\.\d+
```

Result

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
CNY: 23.78
```

与上面类似，我们现在要求仅匹配金额，而不匹配前面的 “CNY:”

正则表达式中，可以使用 **反向预查** 来解决这个问题。本例中，写法是：(?<=CNY:)\d+\.\d+

反向预查 的语法是在子模式内部前面加“?<=”，表示的意思是：首先，要匹配的文本必须满足此子模式 后面 的表达式(本例，“\d+.\d+”)；其次，此子模式不参与匹配。

Text

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
CNY: 23.78
```

RegEx

```
(?<=CNY: )\d+\.\d+
```

Result

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
CNY: 23.78
```

与前面类似：**你可以将 反向预查 理解成为自定义的边界(\b)，这个边界位于 表达式首。**

反言之，你可以将位于 表达式首 的 \b 理解成一个非获取匹配的一个特例：(?<=[,.\r\n<>;\-])。注意，我没有写全所有边界。

Text

```
aaa-xaaaa
aaaxaaaaa
```

RegEx

```
(?<=[,.\r\n<>;\-])x
```

Result

```
aaa-xaaaa
aaaxaaaaa
```

你也可以这样理解上面的匹配过程：

1. 先进行普通匹配：(CNY:)\d+\.\d+
2. 然后从匹配文本中将 子模式 内的文本排除掉。

4. 正向、反向预查组合

我们可以将正向预查 和 反向预查 组合起来使用，比如我们想获取所有 head 之间的文本，就可以这么写：

Text

```
<h1>This is header.</h2>
<h2>This is header,too.</h2>
<span>This is not a header.</span>
```

RegEx

```
(?<=<h(?<number>[1-6])>).*?(?=</h\k<number>>)
```

Result

```
<h1>This is header.</h2>
<h2>This is header,too.</h2>
<span>This is not a header.</span>
```

注意，这里我综合应用了前面的知识，将首尾不一致的“<h1></h2>”做了过滤。

5. 负正向预查、负反向预查

5.1 负正向预查

如同 \b 有与之相对的 \B 一样，正向预查 也有它的 逆过程，称之为 负正向预查。

在正则表达式中，可以在子模式内部前面加 “?!” 来形成一个 负正向预查，它的效果与 “?=” 相反。

Text

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

RegEx

```
Windows(?! [\d.]+\b)
```

Result

```
Windows 1.03 and Windows 2.0 fisrt Released in 1985 and 1987 respectively.
Windows 95 and Windows 98 are the successor.
Then Windows 2000 and Windows Xp appeared.
Windows Vista is the Latest version of the family.
```

从结果我们看到，它匹配了后面不是数字版本的 Windows。

5.2 负反向预查

在正则表达式中，可以在子模式内部前面加 “?<!” 来形成一个 负反向预查，它的效果与 “?<=” 相反。

Text

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
```

```
CNY: 23.78
```

RegEx

```
(?<!CNY: )\b\d+\.\d+
```

Result

```
CNY: 128.04
USD: 22.5
USD: 23.5
HKD: 1533.5
CNY: 23.78
```

这次，它匹配所有前面不是CNY的金额，也就是 HKD 和 USD 后面的数字。

总结

在这篇文章中，我通过大量的范例，向大家详细的介绍了正则表达式。

我们从简单开始，了解了如何匹配单个 及 多个字符。

接着，我们回顾了 边界匹配 和 子模式，

最后，我们讨论了正则表达式中相对高级，但用途依旧广泛的 后向引用、替换 以及 正向、反向预查 等主题。

希望这篇文章能给你带来帮助。