

第 1 章 字符组

1.1 普通字符组

字符组 (Character Class)¹ 是正则表达式最基本的结构之一，要理解正则表达式的“灵活”，认识它是第一步。

顾名思义，字符组就是一组字符，在正则表达式中，它表示“在同一个位置可能出现的各种字符”，其写法是在一对方括号[和]之间列出所有可能出现的字符，简单的字符组比如 [ab]、[314]、[#.?] 在解决一些常见问题时，使用字符组可以大大简化操作，下面举“匹配数字字符”的例子来说明。

字符可以分为很多类，比如数字、字母、标点等。有时候要求“只出现一个数字字符”，换句话说，这个位置上的字符只能是 0、1、2、…、8、9 这 10 个字符之一。要进行这种判断，通常的思路是：用 10 个条件分别判断字符是否等于这 10 个字符，对 10 个结果取“或”，只要其中一个条件成立，就返回 True，表示这是一个数字字符，其伪代码如例 1-1 所示。

例 1-1 判断数字字符的伪代码

```
charStr == "0" || charStr == "1" ... || charStr == "9"
```

注：因为正则表达式处理的都是“字符串” (String) 而不是“字符”，所以这里假设变量 charStr (虽然它只包含一个字符) 也是字符串类型，使用了双引号，在有些语言中字符串也用单引号表示。

这种解法的问题在于太烦琐——如果要判断是否是一个小写英文字母，就要用||连接 26 个判断；如果还要兼容大写字母，则要连接 52 个判断，代码长到几乎无法阅读。相反，用字符组解决起来却异常简单，具体思路是：列出可能出现的所有字符（在这个例子里就是 10 个数字字符），只要出现了其中任何一个，就返回 True。例 1-2 给出了使用字符组判断的例子，程序语言使用 Python。

例 1-2 用正则表达式判断数字字符

```
re.search("[0123456789]", charStr) != None
```

re.search() 是 Python 提供的正则表达式操作函数，表示“进行正则表达式匹配”；charStr 仍然是需要判断的字符串，而 [0123456789] 则是以字符串形式给出的正则表达式，它是一个字符组，表示“这里可以是 0、1、2、…、8、9 中的任意一个字符。只要 charStr 与其中任何一个字符相同（或者说“charStr 可以由 [0123456789] 匹配”），就会得到一个 MatchObject 对象（这个对象暂时不必关心，在第 17 页会详细讲解）；否则，返回 None。所以判断结果是否为 None，就可以判断 charStr 是否是数字字符。

¹ 在有的资料中，写作 Character Set，所以也有人翻译为“字符类”或者“字符集”。不过在计算机术语中，“类”是和“对象”相关的，“字符集”常常表示 Character Set (比如 GBK、UTF-8 之类)，所以本书中没有采用这两个名字。

当今流行的编程语言大多支持正则表达式，上面的例子在各种语言中的写法大抵相同，唯一的区别在于如何调用正则表达式的功能，所以用法其实大同小异。例 1-3 列出了常见语言中的表示，如果你现在就希望知道语言的细节，可以参考本书第三部分的具体章节。

例 1-3 用正则表达式判断数字字符在各种语言中的应用²

```
.NET (C#)  
//能匹配则返回 true, 否则返回 false  
Regex.IsMatch(charStr, "[0123456789]");  
  
Java  
//能匹配则返回 true, 否则返回 false  
charStr.matches("[0123456789]");  
  
JavaScript  
//能匹配则返回 true, 否则返回 false  
/[0123456789]/.test(charStr);  
  
PHP  
//能匹配则返回 1, 否则返回 0  
preg_match("/[0123456789]/", charStr);  
  
Python  
#能匹配则返回 RegexObject, 否则返回 None  
re.search("[0123456789]", charStr)  
  
Ruby  
#能匹配则返回 0, 否则返回 nil  
charStr =~ /[0123456789]/
```

可以看到，不同语言使用正则表达式的方法也不相同。如果仔细观察会发现 Java、.NET、Python、PHP 中的正则表达式，都要以字符串形式给出，两端都有双引号²；而 Ruby 和 JavaScript 中的正则表达式则不必如此，只在首尾有两个斜线字符³，这也是不同语言中使用正则表达式的不同之处。不过，这个问题现在不需要太关心，因为本书中大部分例子以 Python 程序来讲解，下面讲解关于 Python 的基础知识，其他语言的细节留到后文会详细介绍。

1.2 关于 Python 的基础知识

本书选择使用 Python 语言来演示实际的匹配结果，因为它能在多种操作系统中运行，安装也很方便；另一方面，Python 是解释型语言，输入代码就能看到结果，方便动手实践。考虑到不是所有人都熟悉 Python，这里专门用一节来介绍。

如果你的机器上没有安装 Python，可以从 <http://python.org/download/> 下载，目前 Python 有 2 和 3 两个版本，本书的例子以 2 版本为准³。请选择自己平台对应的程序下载并安装（目前 MacOS、Linux 的各种发行版一般带有 Python，具体可以在命令行下输入 python，看是否启动对应的程序）。

然后可以启动 Python，在 MacOS 和 Linux 下是输入 python，会显示出 Python 提示符，进入交互

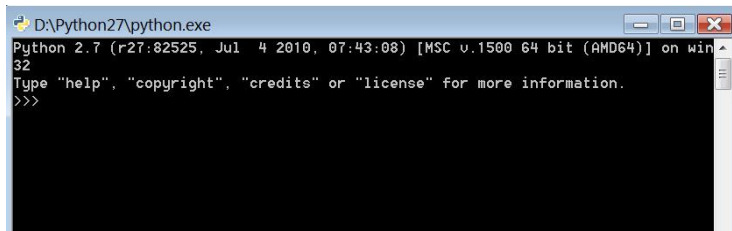
² 客观地说，Perl 是正则表达式处理最方便的编程语言，考虑到今天使用 Perl 的人数，以及 Perl 程序员一般都熟练掌握正则表达式的现实，本书没有给出 Perl 语言的例子。

³ 本书写作时，2.x 最新的版本为 2.7，示例以此为准。Python 3 虽然已经正式发行，但相对 2.x 变化较大，而 2.x 较为流行，所以采用了 2.x 版本。关于 2.x 和 3.x 的差别，在 Python 一章有详细介绍。

模式，如图 1-1（Linux 下的提示符与 MacOS 下的差不多，所以此处不列出）；而在 Windows 下，需要在“开始”菜单的“程序”中，选择 Python 目录下的 Python(command line)，如图 1-2 所示。

```
Python 2.7.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1-1 MacOS 下的 Python 提示符



```
D:\Python27\python.exe
Python 2.7 (r27:82525, Jul 4 2010, 07:43:08) [MSC v.1500 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1-2 Windows 下的 Python 提示符

Python 中常用的关于正则表达式的函数是 `re.search()`，使用它必须首先导入正则表达式对应的包（package），也就是输入下面的代码。

```
#导入正则表达式对应的包
import re
```

通常的用法是提供两个参数：`re.search(pattern, string)`，其中 `pattern` 是字符串形式提供的正则表达式，`string` 是需要匹配的字符串；如果能匹配，则返回一个 `MatchObject`（详细介绍请参考第 **错误！未定义书签。** 页，暂时可以不必关心），这时提示符会显示类似 `<_sre.SRE_Match object at 0x000000001D8E578>` 之类的结果；如果不能匹配，结果是 `None`（这是 Python 中的一个特殊值，类似其他某些语言中的 `Null`），不会有任何显示。图 1-3 演示了运行 Python 语句的结果。

```
>>> import re
>>> re.search("[0123456789]", "4")
<_sre.SRE_Match object at 0x000000001D8E578>
>>> re.search("[0123456789]", "a")
>>>
```

图 1-3 观察 `re.search()` 匹配的返回值

注：`>>>`是等待输入的提示符，以`>>>`开头的行，之后文本是用户输入的语句；其他行是系统生成的，比如打印出语句的结果（在交互模式下，匹配结果会自动输出，便于观察；真正程序运行时不会如此）。

为讲解清楚、形象、方便，本书中的程序部分需要做两点修改。

第一，因为暂时还不需要关心匹配结果的细节，只关心有没有结果，所以在 `re.search()` 之后添加判断返回值是否为 `None`，如果为 `True`，则表示匹配成功，否则返回 `False` 表示匹配失败。为节省版面，尽可能用注释表示这类匹配结果，如 `# => True` 或者 `# => False`，附在语句之后。

第二，目前我们关心的是整个字符串是否能由正则表达式匹配。但是，在默认情况下 `re.search(pattern, string)` 只判断 `string` 的某个子串能否由 `pattern` 匹配，即便 `pattern` 只能匹配 `string` 的一部分，也不会返回 `None`。为了测试整个 `string` 能否由 `pattern` 匹配，在 `pattern` 两端加上 `^` 和 `$`。`^` 和 `$` 是正则表达式中的特殊字符，它们并不匹配任何字符，只是表示“定位到字符串的起始位置”和“定位到字符串的结束位置”（原理如图 1-4 所示，如果你现在就希望详细了解这两个特殊字符，可

以参考第**错误! 未定义书签**.页), 这样就保证: 只有在整个 *string* 都可以由 *pattern* 匹配时, 才算匹配成功, 不返回 None, 如例 1-4 所示。

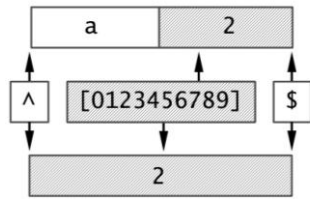


图 1-4 `^[0123456789]$`的匹配

例 1-4 使用^和\$测试 string 由 pattern 完整匹配

```
# 只要字符串中包含数字字符, 就可以匹配
re.search("[0123456789]", "2") != None           # => True
re.search("^[0123456789]$", "12") != None       # => False
re.search("[0123456789]", "a2") != None         # => True

# 整个字符串就是一个数字字符, 才可以匹配
re.search("[0123456789]", "2") != None           # => True
re.search("^[0123456789]$", "12") != None       # => False
re.search("^[0123456789]$", "a2") != None       # => False
```

1.3 普通字符组 (续)

介绍完关于 Python 的基础知识, 继续讲解字符组。字符组中的字符排列顺序并不影响字符组的功能, 出现重复字符也不会影响, 所以 `[0123456789]` 完全等价于 `[9876543210]`、`[1029384756]`、`[9988876543210]`。

不过, 代码总是要容易编写, 方便阅读, 正则表达式也是一样, 所以一般并不推荐在字符组中出现重复字符。而且, 还应该让字符组中的字符排列更符合认知习惯, 比如 `[0123456789]` 就好过 `[0192837465]`。为此, 正则表达式提供了**-范围表示法** (range), 它更直观, 能进一步简化字符组。

所谓“-范围表示法”, 就是用 `[x-y]` 的形式表示 *x* 到 *y* 整个范围内的字符, 省去一一列出的麻烦, 这样 `[0123456789]` 就可以表示为 `[0-9]`。如果你觉得这不算什么, 那么确实比 `[abcdefghijklmnopqrstuvwxyz]` 简单太多了。

你可能会问, “-范围表示法”的范围是如何确定的? 为什么要写作 `[0-9]`, 而不写作 `[9-0]`?

要回答这个问题, 必须了解范围表示法的实质。在字符组中, `-`表示的范围, 一般是根据字符对应的**码值** (Code Point, 也就是字符在对应编码表中的编码的数值) 来确定的, 码值小的字符在前, 码值大的字符在后。在 ASCII 编码中 (包括各种兼容 ASCII 的编码中), 字符 `0` 的码值是 48 (十进制), 字符 `9` 的码值是 57 (十进制), 所以 `[0-9]` 等价于 `[0123456789]`; 而 `[9-0]` 则是错误的范围, 因为 `9` 的码值大于 `0`, 所以会报错。程序代码见例 1-5。

例 1-5 `[0-9]`是合法的, `[9-0]`会报错

```
re.search("^[0-9]$", "2") != None           # => True

re.search("^[9-0]$", "2") != None
Traceback (most recent call last):
```

error: bad character range

如果知道 0~9 的码值是 48~57, a~z 的码值是 97~122, A~Z 的码值是 65~90, 能不能用 [0-z] 统一表示数字字符、小写字母、大写字母呢?

答案是: 勉强可以, 但不推荐这么做。根据惯例, 字符组的范围表示法都表示一类字符 (数字字符是一类, 字母字符也是一类), 所以虽然 [0-9]、[a-z] 都是很好理解的, 但 [0-z] 却很难理解, 不熟悉 ASCII 编码表的人甚至不知道这个字符组还能匹配大写字母, 更何况, 在码值 48 到 122 之间, 除去数字字符 (码值 48~57)、小写字母 (码值 97~122)、大写字母 (码值 65~90), 还有不少标点符号 (参见表 1-1), 从字符组 [0-z] 中却很难看出来, 使用时就容易引起误会, 例 1-6 所示的程序就可能让人莫名其妙。

表 1-1 ASCII 编码表 (片段)

码值	字符	码值	字符	码值	字符	码值	字符	码值	字符
48	0	63	?	78	N	93]	108	l
49	1	64	@	79	O	94	^	109	m
50	2	65	A	80	P	95	_	110	n
51	3	66	B	81	Q	96	`	111	o
52	4	67	C	82	R	97	a	112	p
53	5	68	D	83	S	98	b	113	q
54	6	69	E	84	T	99	c	114	r
55	7	70	F	85	U	100	d	115	s
56	8	71	G	86	V	101	e	116	t
57	9	72	H	87	W	102	f	117	u
58	:	73	I	88	X	103	g	118	v
59	;	74	J	89	Y	104	h	119	w
60	<	75	K	90	Z	105	i	120	x
61	=	76	L	91	[106	j	121	y
62	>	77	M	92	\	107	k	122	z

例 1-6 [0-z] 的奇怪匹配

```
re.search("[^0-z]$", "A") != None      # => True
re.search("[^0-z]$", ":") != None     # => True
```

在字符组中可以同时并列多个“-范围表示法”, 字符组 [0-9a-zA-Z] 可以匹配数字、大写字母或小写字母; 字符组 [0-9a-fA-F] 可以匹配数字, 大、小写形式的 a~f, 它可以用来验证十六进制字符, 代码见例 1-7。

例 1-7 [0-9a-fA-F] 准确判断十六进制字符

```
re.search("[^0-9a-fA-F]$", "0") != None # => True
re.search("[^0-9a-fA-F]$", "c") != None # => True
re.search("[^0-9a-fA-F]$", "i") != None # => False
re.search("[^0-9a-fA-F]$", "C") != None # => True
re.search("[^0-9a-fA-F]$", "G") != None # => False
```

在不少语言中，还可以用转义序列 `\xhex` 来表示一个字符，其中 `\x` 是固定前缀，表示转义序列的开头，`num` 是字符对应的码值（Code Point，详见第 127 页，下文用 $\S 127$ 表示），是一个两位的十六进制数值。比如字符 `A` 的码值是 41（十进制则为 65），所以也可以用 `\x41` 表示。

字符组中有时会出现这种表示法，它可以表现一些难以输入或者难以显示的字符，比如 `\x7F`；也可以用来方便地表示某个范围，比如所有 ASCII 字符对应的字符组就是 `[\x00-\x7F]`，代码见例 1-8。这种表示法很重要，在第 **错误！未定义书签**。页还会讲到它，依靠这种表示法可以很方便地匹配所有的中文字符。

例 1-8 `[\x00-\x7F]` 准确判断 ASCII 字符

```
re.search("[\x00-\x7F]", "c") != None # => True
re.search("[\x00-\x7F]", "I") != None # => True
re.search("[\x00-\x7F]", "0") != None # => True
re.search("[\x00-\x7F]", "<") != None # => True
```

1.4 元字符与转义

在上面的例子里，字符组中的横线 `-` 并不能匹配横线字符，而是用来表示范围，这类字符叫做元字符（meta-character）。字符组的开方括号 `[`、闭方括号 `]` 和之前出现的 `^`、`$` 都算元字符。在匹配中，它们有着特殊的意义。但是，有时候并不需要表示这些特殊意义，只需要表示普通字符（比如“我就想表示横线字符 `-`”），此时就必须做特殊处理。

先来看字符组中的 `-`，如果它紧邻着字符组中的开方括号 `[`，那么它就是普通字符，其他情况下都是元字符；而对于其他元字符，取消特殊含义的做法都是转义，也就是在正则表达式中的元字符之前加上反斜线字符 `\`。

如果要在字符组内部使用横线 `-`，最好的办法是将其排列在字符组的最开头。`[-09]` 就是包含三个字符 `-`、`0`、`9` 的字符组；`[0-9]` 是包含 `0-9` 这 10 个字符的字符组，`[-0-9]` 则是由“-范围表示法”`0-9` 和横线 `-` 共同组成的字符组，它可以匹配 11 个字符，例 1-9 说明了使用横线 `-` 的各种情况。

例 1-9 `-` 出现在不同位置，含义不同

```
#作为普通字符
re.search("[[-09]", "3") != None # => False
re.search("[[-09]", "-") != None # => True
#作为元字符
re.search("[0-9]", "3") != None # => True
re.search("[0-9]", "-") != None # => False
#转义之后作为普通字符
re.search("[0\\-9]", "3") != None # => False
re.search("[0\\-9]", "-") != None # => True
```

仔细观察会发现，在正文里说“在正则表达式中的元字符之前加上反斜线字符 `\`”，而在代码里写的却不是 `[0\\-9]`，而是 `[0\\-9]`。这并不是输入错误。

因为在这段程序里，正则表达式是以字符串（String）的方式⁴提供的，而字符串本身也有关于转义的规定（你或许记得，在字符串中有 `\n`、`\t` 之类的转义序列）。上面说的“正则表达式”，其实是经

⁴ 具体来说，在 Java、PHP、Python、.NET 等语言中，正则表达式都是以字符串的形式给出的，在 Ruby 和 JavaScript 中则不是这样。详细的说明，请参考第 96 页。

过“字符串转义处理”之后的字符串的值，正则表达式 `[0\ -9]` 包含 6 个字符：`[、0、\、-、9、]`，在字符串中表达这 6 个字符；但是在源代码里，必须使用 7 个字符：`\` 需要转义成 `\\`，因为处理字符串时，反斜线和之后的字符会被认为是转义序列（Escape Sequence），比如 `\n`、`\t` 都是合法的转义序列，然而 `\-` 不是。

这个问题确实有点麻烦。正则表达式是用来处理字符串的，但它又不完全等于字符串，正则表达式中的每个反斜线字符 `\`，在字符串中（也就是正则表达式之外）还必须转义为 `\\`。所以之前所说的是“正则表达式 `[0\ -9]`”，程序里写的却是 `[0\\ -9]`，这确实有点麻烦。

不过，Python 提供了原生字符串（Raw String），它非常适合于正则表达式：正则表达式是怎样，原生字符串就是怎样，完全不需要考虑正则表达式之外的转义（只有双引号字符是例外，原生字符串内的双引号字符必须转义写成 `\"`）。原生字符串的形式是 `r"string"`，也就是在普通字符串之前添加 `r`，示例代码如例 1-10。

例 1-10 原生字符串的使用

```
#原生字符串和字符串的等价
r"^[0\ -9]$" == "[0\\ -9]$"          # => True
#原生字符串的转义要简单许多
re.search(r"^[0\ -9]$", "3") != None # => False
re.search(r"^[0\ -9]$", "-") != None # => True
```

原生字符串清晰易懂，省去了烦琐的转义，所以从现在开始，本书中的 Python 示范代码都会使用原生字符串来表示正则表达式。另外，.NET 和 Ruby 中也有原生字符串，也有一些语言并没有提供原生字符串（比如 Java），所以在第 6 章（[错误！未找到引用源。错误！未定义书签。](#)）会专门讲解转义问题。不过，现在只需要知道 Python 示范代码中使用了原生字符串即可。

继续看转义，如果希望在字符组中列出闭方括号 `]`，比如 `[012]345]`，就必须在它之前使用反斜线转义，写成 `[012\]345]`；否则，结果就如例 1-11 所示，正则表达式将 `]` 与最近的 `[` 匹配，这个表达式就成了“字符组 `[012]` 加上 4 个字符 `345]`”，它能匹配的是字符串 `0345]` 或 `1345]` 或 `2345]`，却不能匹配 `]`。

例 1-11 `]` 出现在不同位置，含义不同

```
#未转义的]
re.search(r"^[012]345]$", "2345") != None # => True
re.search(r"^[012]345]$", "5") != None    # => False
re.search(r"^[012]345]$", "]") != None    # => False
#转义的]
re.search(r"^[012\ ]345]$", "2345") != None # => False
re.search(r"^[012\ ]345]$", "5") != None    # => True
re.search(r"^[012\ ]345]$", "]") != None    # => True
```

除去字符组内部的 `-`，其他元字符的转义都必须在字符之前添加反斜线，`]` 的转义也是如此。如果只希望匹配字符串 `[012]`，直接使用正则表达式 `[012]` 是不行的，因为这会被识别为一个字符组，它只能匹配 `0`、`1`、`2` 这三个字符中的任意一个；而必须转义，把正则表达式写作 `\[012]`，请注意，只有开方括号 `[` 需要转义，闭方括号 `]` 不需要转义，如例 1-12 所示。

例 1-12 取消其他元字符的特殊含义

```
re.search(r"^[012]345]$", "3") != None # => False
re.search(r"^[012\ ]345]$", "3") != None # => True
```

```
re.search(r"^[012]$", "[012]") != None      # => False
re.search(r"^\[012]$", "[012]") != None     # => True
```

1.5 排除型字符组

在方括号`[...]`中列出希望匹配的所有字符，这种字符组叫做“普通字符组”，它的确非常方便。不过，也有些问题是普通字符组不能解决的。

给定一个由两个字符构成的字符串 *str*，要判断这两个字符是否都是数字字符，可以用`[0-9][0-9]`来匹配。但是，如果要求判断的是这样的字符串——第一个字符不是数字字符，第二个字符才是数字字符（比如 `A8`、`x6`）⁵——应当如何办？数字字符的匹配很好处理，用`[0-9]`即可；“不是数字”则很难办——不是数字的字符太多了，全部列出几乎不可能，这时就应当使用排除型字符组。

排除型字符组（Negated Character Class）非常类似普通字符组`[...]`，只是在开方括号`[`之后紧跟一个脱字符`^`，写作`[^...]`，表示“在当前位置，匹配一个没有列出的字符”。所以`[^0-9]`就表示“`0~9`之外的字符”，也就是“非数字字符”。那么，`[^0-9][0-9]`就可以解决问题了，如例 1-13 所示。

例 1-13 使用排除型字符组

```
re.search(r"^[^0-9][0-9]$", "A8") != None    # => True
re.search(r"^[^0-9][0-9]$", "x6") != None    # => True
```

排除型字符组看起来很简单，不过新手常常会犯一个错误，就是把“在当前位置，匹配一个没有列出的字符”理解成“在当前位置不要匹配列出的字符”两者其实是不同的，后者暗示“这里不出现任何字符也可以”。例 1-14 很清楚地说明：**排除型字符组必须匹配一个字符**，这点一定要记住。

例 1-14 排除型字符组必须匹配一个字符

```
re.search(r"^[^0-9][0-9]$", "8") != None     # => False
re.search(r"^[^0-9][0-9]$", "A8") != None    # => True
```

除了开方括号`[`之后的`^`，排除型字符组的用法与普通字符组几乎完全相同，唯一需要改动的是：在排除型字符组中，如果需要表示横线字符`-`（而不是用于“-范围表示法”），那么`-`应该紧跟在`^`之后；而在普通字符组中，作为普通字符的横线`-`应该紧跟在开方括号之后，如例 1-15 所示。

例 1-15 在排除型字符组中，紧跟在`^`之后的`-`不是元字符

```
#匹配一个-、0、9之外的字符
re.search(r"^[^-09]$", "-") != None          # => False
re.search(r"^[^-09]$", "8") != None         # => True

#匹配一个0~9之外的字符
re.search(r"^[^0-9]$", "-") != None         # => True
re.search(r"^[^0-9]$", "8") != None         # => False
```

在排除型字符组中，`^`是一个元字符，但只有它紧跟在`[`之后时才是元字符，如果想表示“这

⁵ 一般来说，计算机中的偏移值都是从 0 开始的。此处考虑到叙述自然，使用了“第一个字符”和“第二个字符”的说法，其中“第一个字符”指最左端，也就是偏移值为 0 的字符；“第二个字符”指紧跟在它右侧，也就是偏移值为 1 的字符。

个字符组中可以出现`^`字符”，不要让它紧挨着`[`即可，否则就要转义。例 1-16 给出了三个正则表达式，后两个表达式实质是一样的，但第三种写法很麻烦，理解起来也麻烦，不推荐使用。

例 1-16 排除型字符组的转义

```
#匹配一个 0、1、2 之外的字符
re.search(r"^[^012]$", "") != None      # => True
#匹配 4 个字符之一：0、^、1、2
re.search(r"^[0^12]$", "") != None      # => True
#^紧跟在[之后，但经过转义变为普通字符，等于上一个表达式，不推荐
re.search(r"^[^\012]$", "") != None     # => True
```

1.6 字符组简记法

用`[0-9]`、`[a-z]`等字符组，可以很方便地表示数字字符和小写字母字符。对于这类常用的字符组，正则表达式提供了更简单的记法，这就是**字符组简记法**（shorthands）。

常见的字符组简记法有`\d`、`\w`、`\s`。从表面上看，它们与`[...]`完全没联系，其实是一致的。其中`\d`等价于`[0-9]`，其中的 d 代表“数字（digit）”；`\w`等价于`[0-9a-zA-Z_]`，其中的 w 代表“单词字符（word）”；`\s`等价于`[\t\r\n\v\f]`（第一个字符是空格），s 表示“空白字符（space）”。

例 1-17 说明了这几个字符组简记法的典型匹配。

例 1-17 字符组简记法`\d`、`\w`、`\s`

```
#如果没有原生字符串，\d 就必须写作\\d
re.search(r"^\d$", "8") != None          # => True
re.search(r"^\d$", "a") != None          # => False

re.search(r"^\w$", "8") != None          # => True
re.search(r"^\w$", "a") != None          # => True
re.search(r"^\w$", "_") != None          # => True

re.search(r"^\s$", " ") != None          # => True
re.search(r"^\s$", "\t") != None         # => True
re.search(r"^\s$", "\n") != None         # => True
```

一般印象中，单词字符似乎只包含大小写字母，但是字符组简记法中的“单词字符”不只有大小写单词，还包括数字字符和下画线`_`，其中的下画线`_`尤其值得注意：在进行数据验证时，有可能只容许输入“数字和字母”，有人会偷懒用`\w`验证，而忽略了`\w`能匹配下画线，所以这种匹配并不严格，`[0-9a-zA-Z_]`才是准确的选择。

“空白字符”并不难定义，它可以是空格字符、制表符`\t`，回车符`\r`，换行符`\n`等各种“空白”字符，只是不方便展现（因为显示和印刷出来都是空白）。不过这也提醒我们注意，匹配时看到的“空白”可能不是空格字符，因此，`\s`才是准确的选择。

字符组简记法可以单独出现，也可以使用在字符组中，比如`[0-9a-zA-Z_]`也可以写作`[\da-zA-Z_]`，所以匹配十六进制字符的字符组可以写成`[\da-fA-F]`。字符组简记法也可以用在排除型字符组中，比如`[^0-9]`就可以写成`[^\d]`，`[^0-9a-zA-Z_]`就可以写成`[^\w]`，代码如例 1-18。

例 1-18 字符组简记法与普通字符组混用

```
#用在普通字符组内部
re.search(r"^\[da-zA-Z]$", "8") != None # => True
re.search(r"^\[da-zA-Z]$", "a") != None # => True
re.search(r"^\[da-zA-Z]$", "C") != None # => True
#用在排除型字符组内部
re.search(r"^[^\w]$", "8") != None # => False
re.search(r"^[^\w]$", "_") != None # => False
re.search(r"^[^\w]$", ",") != None # => True
```

相对于 `\d`、`\w` 和 `\s` 这三个普通字符组简记法，正则表达式也提供了对应排除型字符组的简记法：`\D`、`\W` 和 `\S`——字母完全一样，只是改为大写。这些简记法匹配的字符互补：`\s` 能匹配的字符，`\S` 一定不能匹配；`\w` 能匹配的字符，`\W` 一定不能匹配；`\d` 能匹配的字符，`\D` 一定不能匹配。例 1-19 示范了这几个字符组简记法的应用。

例 1-19 `\D`、`\W`、`\S` 的使用

```
#\d 和 \D
re.search(r"^\d$", "8") != None # => True
re.search(r"^\d$", "a") != None # => False
re.search(r"^\D$", "8") != None # => False
re.search(r"^\D$", "a") != None # => True
#\w 和 \W
re.search(r"^\w$", "c") != None # => True
re.search(r"^\w$", "!") != None # => False
re.search(r"^\W$", "c") != None # => False
re.search(r"^\W$", "!") != None # => True
#\s 和 \S
re.search(r"^\s$", "\t") != None # => True
re.search(r"^\s$", "0") != None # => False
re.search(r"^\S$", "\t") != None # => False
re.search(r"^\S$", "0") != None # => True
```

妥善利用这种互补的属性，可以得到一些非常巧妙的效果，最简单的应用就是字符组 `[\s\S]`。初看起来，在同一个字符组中并列两个互补的简记法，这种做法有点奇怪，不过仔细想想就会明白，`\s` 和 `\S` 组合在一起，匹配的就是“所有的字符”（或者叫“任意字符”）。许多语言中的正则表达式并没有直接提供“任意字符”的表示法，所以 `[\s\S]`、`[\w\W]`、`[\d\D]` 虽然看起来有点古怪，但确实可以匹配任意字符⁶。

关于字符组简记法，最后需要补充两点：第一，如果字符组中出现了字符组简记法，最好不要出现单独的 `_`，否则可能引起错误，比如 `[\d-a]` 就让人很迷惑，在有些语言中，`_` 会被作为普通字符，而在有些语言中，这样写会报错；第二，以上说的 `\d`、`\w`、`\s` 的匹配规则，都是针对 ASCII 编码而言的，也叫 **ASCII 匹配规则**。但是，目前一些语言中的正则表达式已经支持了 Unicode 字符，那么数字字符、单词字符、空白字符的范围，已经不仅仅限于 ASCII 编码中的字符。关于这个问题，具体细节在后文有详细的介绍，如果你现在就想知道，可以翻到第 **错误！未定义书签。** 页。

⁶ 许多关于正则表达式的文档说：点号 `.` 能匹配“任意字符”。但在默认情况下，点号其实不能匹配换行符，具体请参考第 84 页。

1.7 字符组运算

以上介绍了字符组的基本功能，它们在常用的语言中都有提供；还有些语言中为字符组提供了更强大的功能，比如 Java 和 .NET 就提供了字符组运算的功能，可以在字符组内进行集合运算，在某些情况下这种功能非常实用。

如果要匹配所有的元音字母（为讲解简单考虑，暂时只考虑小写字母的情况），可以用 `[aeiou]`，但是要匹配所有的辅音字母却没有什么方便的办法，最直接的写法是 `[b-df-hj-np-tv-z]`，不但烦琐，而且难理解。其实，从 26 个字母中“减去”元音字母，剩下的就是辅音字母，如果有办法做这个“减法”，就方便多了。

Java 语言中提供了这样的字符组：`[[a-z]&&[^aeiou]]`，虽然初看有点古怪，但仔细看看，也不难理解。`[a-z]` 表示 26 个英文字母，`[^aeiou]` 表示除元音字母之外的所有字符（还包括大写字母、数字和各种符号），两者取交集，就得到“26 个英文字母中，除去 5 个元音字母，剩下的 21 个辅音字母”。

.NET 中也有这样的功能，只是写法不一样。同样是匹配辅音字母的字符组，.NET 中写作 `[a-z-[aeiou]]`，其逻辑是：从 `[a-z]` 能匹配的 26 个字符中，“减去” `[aeiou]` 能匹配的元音字母。相对于 Java，这种逻辑更符合直觉，但写法却有点古怪——不是 `[[a-z]-[aeiou]]`，而是 `[a-z-[aeiou]]`。例 1-20 集中演示了 Java 和 .NET 中的字符组运算。

例 1-20 字符组运算

```
Java
"a".matches("^[[a-z]&&[^aeiou]]$"); // => True
"b".matches("^[[a-z]&&[^aeiou]]$"); // => False

.NET
Regex.IsMatch("^[a-z-[aeiou]]$", "a"); // => True
Regex.IsMatch("^[a-z-[aeiou]]$", "b"); // => False
```

1.8 POSIX 字符组

前面介绍了常用的字符组，但是在某些文档中，你可能会发现类似 `[:digit:]`、`[:lower:]` 之类的字符组，看起来不难理解（`digit` 就是“数字”，`lower` 就是“小写”），但又很奇怪，它们就是 **POSIX 字符组**（POSIX Character Class）。因为某些语言的文档中出现了这些字符组，为避免困惑，这里有必要做个简要介绍。如果只使用常用的编程语言，可以忽略文档中的 POSIX 字符组，也可以忽略本节；如果了解 POSIX 字符组，或者需要在 Linux/UNIX 下的各种工具（`sed`、`awk`、`grep` 等）中使用正则表达式，最好阅读本节。

之前介绍的字符组，都属于 Perl 衍生出来的正则表达式流派（Flavor），这个流派叫做 PCRE（Per Compatible Regular Expression）。在此之外，正则表达式还有其他流派，比如 POSIX（Portable Operating System Interface for uniX），它是一系列规范，定义了 UNIX 操作系统应当支持的功能，其中也包括关于正则表达式的规范，`[:digit:]` 之类的字符组就是遵循 POSIX 规范的字符组。

常见的 `[a-z]` 形式的字符组，在 POSIX 规范中仍然获得支持，它的准确名称是 **POSIX 方括号表**

达式 (POSIX bracket expression)，主要用在 UNIX/Linux 系统中。POSIX 方括号表达式与之前所说的字符组最主要的差别在于：在 POSIX 字符组中，反斜线\不是用来转义的。所以 POSIX 方括号表达式 `[\d]` 只能匹配 `\` 和 `d` 两个字符，而不是 `[0-9]` 对应的数字字符。

为了解决字符组中特殊意义字符的转义问题，POSIX 方括号表达式规定：如果要在字符组中表达字符] (而不是作为字符组的结束标记)，应当让它紧跟在字符组的开方括号之后，所以 `[]a` 能匹配的字符就是] 或 a；如果要在字符组中标识字符- (而不是“-范围表示法”)，就必须将它放在字符组的闭方括号] 之前，所以 `[a-]` 能匹配的字符就是 a 或-。

另一方面，POSIX 规范还定义了 POSIX 字符组 (POSIX character class)，它大致等于之前介绍的字符组简记法，都是使用类似 `[:digit:]`、`[:lower:]` 之类有明确意义的记号表示某类字符。

表 1-2 简要介绍了 POSIX 字符组，注意表格中与其对应的是 ASCII 字符组，也就是能匹配的 ASCII 字符 (ASCII 编码表中码值在 0~127 之间的字符)。因为 POSIX 规范中有一个重要概念：locale (通常翻译为“语言环境”)，它是一组与语言和文化相关的设定，包括日期格式、货币币值、字符编码等。POSIX 字符组的意义会根据 locale 的变化而变化，表 1-2 介绍的只是这些 POSIX 字符组在 ASCII 编码中的意义；如果换用其他的 locale (比如使用 Unicode 字符集)，它们的意义可能会发生变化，具体请参考第 **错误！未定义书签。** 页。

表 1-2 POSIX 字符组

POSIX 字符组	说明	ASCII 字符组	等价的 PCRE 简记法
<code>[:alnum:]*</code>	字母字符和数字字符	<code>[a-zA-Z0-9]</code>	
<code>[:alpha:]</code>	字母	<code>[a-zA-Z]</code>	

(续表)

POSIX 字符组	说明	ASCII 字符组	等价的 PCRE 简记法
<code>[:ASCII:]</code>	ASCII 字符	<code>[\x00-\x7F]</code>	
<code>[:blank:]</code>	空格字符和制表符	<code>[\t]</code>	
<code>[:cntrl:]</code>	控制字符	<code>[\x00-\x1F\x7F]</code>	
<code>[:digit:]</code>	数字字符	<code>[0-9]</code>	<code>\d</code>
<code>[:graph:]</code>	空白字符之外的字符	<code>[\x21-\x7E]</code>	
<code>[:lower:]</code>	小写字母字符	<code>[a-z]</code>	
<code>[:print:]</code>	类似 <code>[:graph:]</code> ，但包括空白字符	<code>[\x20-\x7E]</code>	
<code>[:punct:]</code>	标点符号	<code>[] [! " # \$ % & ' () * + , . / : ; < = > ? @ ^ _ ` { } ~ -]</code>	
<code>[:space:]</code>	空白字符	<code>[\t\r\n\v\f]</code>	<code>\s</code>
<code>[:upper:]</code>	大写字母字符	<code>[A-Z]</code>	
<code>[:word:]*</code>	字母字符	<code>[A-Za-z0-9_]</code>	<code>\w</code>
<code>[:xdigit:]</code>	十六进制字符	<code>[A-Fa-f0-9]</code>	

注：标记*的字符组简记法并不是 POSIX 规范中的，但使用很多，一般语言中都提供，文档中也会出现。

POSIX 字符组的使用也与 PCRE 字符组简记法的使用有所不同，主要区别在于，PCRE 字符组简记法可以脱离方括号直接出现，而 POSIX 字符组必须出现在方括号内。所以同样是匹配数字字符，PCRE 中可以直接写 `\d`，而 POSIX 字符组必须写成 `[[:digit:]]`。

在本书介绍的 6 种语言中，Java、PHP、Ruby 支持使用 POSIX 字符组。

在 PHP 中可以直接使用 POSIX 字符组，但是 PHP 中的 POSIX 字符组只识别 ASCII 字符，也就是说，任何非 ASCII 字符（比如中文字符）都不能由任何一个 POSIX 字符组匹配。

Ruby 的情况稍微复杂一点。Ruby 1.8 中的 POSIX 字符组只能匹配 ASCII 字符，而且不支持 `[:word:]` 和 `[:ASCII:]`；Ruby 1.9 中的 POSIX 字符组可以匹配 Unicode 字符，而且支持 `[:word:]` 和 `[:ASCII:]`。

Java 中的情况更加复杂。POSIX 字符组 `[[:name:]]` 必须使用 `\p{name}` 的形式，其中 *name* 为 POSIX 字符组对应的名字，比如 `[:space:]` 就应当写作 `\p{Space}`，请注意第一个字母要大写，其他 POSIX 字符组都是这样，只有 `[:xdigit:]` 要写作 `\p{XDigit}`。并且 Java 中的 POSIX 字符组，只能匹配 ASCII 字符。