**3　字符类**：

|  |  |
| --- | --- |
| **字符类** | |
| [abc] | a, b 或 c（简单类） |
| [^abc] | 除 a, b 或 c 之外的任意字符（取反） |
| [a-zA-Z] | a 到 z，或 A 到 Z，包括（范围） |
| [a-d[m-p]] | a 到 d，或 m 到 p：[a-dm-p]（并集） |
| [a-z&&[def]] | d，e 或 f（交集） |
| [a-z&&[^bc]] | 除 b 和 c 之外的 a 到 z 字符：[ad-z]（差集） |
| [a-z&&[^m-p]] | a 到 z，并且不包括 m 到 p：[a-lq-z]（差集） |

　　左边列指定正则表达式构造，右边列描述每个构造的匹配的条件。

注意：“字符类（character class）”这个词中的“类（class）”指的并不是一个 .class 文件。在正则表达式的语义中，字符类是放在方括号里的字符集，指定了一些字符中的一个能被给定的字符串所匹配。

字符类最基本的格式是把一些字符放在一对方括号内。例如：正则表达式[bcr]at会匹配“bat”、“cat”或者“rat”，这是由于其定义了一个字符类（接受“b”、“c”或“r”中的一个字符）作为它的首字符。

**4　预定义字符类**

　　Pattern 的 API 包有许多有用的预定义字符类（predefined character classes），提供了常用正则表达式的简写形式。

|  |  |
| --- | --- |
| **预定义字符类** | |
| . | 任何字符（匹配或者不匹配行结束符） |
| \d | 数字字符：[0-9] |
| \D | 非数字字符：[^0-9] |
| \s | 空白字符：[\t\n\x0B\f\r] |
| \S | 非空白字符：[^\s] |
| \w | 单词字符：[a-zA-Z\_0-9] |
| \W | 非单词字符：[^\w] |

　　\d 匹配数字字符  
　　\s 匹配空白字符  
　　\w 匹配单词字符  
　　也可以使用意思正好相反的大写字母：  
　　\D 匹配非数字字符  
　　\S 匹配非空白字符  
　　\W 匹配非单词字符

**5　量词**[返回目录](http://www.java3z.com/cwbwebhome/article/article8/Regex/Java.Regex.Tutorial.html" \l "contents)

　　这一节我们来看一下贪婪（greedy）、勉强（reluctant）和侵占（possessive）量词，来匹配指定表达式X的次数。  
　　量词（quantifiers）允许指定匹配出现的次数，方便起见，当前 Pattern API 规范下，描述了贪婪、勉强和侵占三种量词。

| **量　词　种　类** | | | **意　　义** |
| --- | --- | --- | --- |
| **贪婪** | **勉强** | **侵占** |
| X? | X?? | X?+ | 匹配 X 零次或一次 |
| X\* | X\*? | X\*+ | 匹配 X 零次或多次 |
| X+ | X+? | X++ | 匹配 X 一次或多次 |
| X{n} | X{n}? | X{n}+ | 匹配 X n 次 |
| X{n,} | X{n,}? | X{n,}+ | 匹配 X 至少 n 次 |
| X{n,m} | X{n,m}? | X{n,m}+ | 匹配 X 至少 n 次，但不多于 m 次 |

Enter your regex: [abc]{3}

Enter input string to search: abccabaaaccbbbc

I found the text "abc" starting at index 0 and ending at index 3.

I found the text "cab" starting at index 3 and ending at index 6.

I found the text "aaa" starting at index 6 and ending at index 9.

I found the text "ccb" starting at index 9 and ending at index 12.

I found the text "bbc" starting at index 12 and ending at index 15.

Enter your regex: abc{3}

Enter input string to search: abccabaaaccbbbc

No match found.

为了说明一下，看看输入的字符串是 xfooxxxxxxfoo 时。

Enter your regex: .\*foo // 贪婪量词

Enter input string to search: xfooxxxxxxfoo

I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .\*?foo // 勉强量词

Enter input string to search: xfooxxxxxxfoo

I found the text "xfoo" starting at index 0 and ending at index 4.

I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .\*+foo // 侵占量词

Enter input string to search: xfooxxxxxxfoo

No match found.

第一个例子使用贪婪量词.\*，寻找紧跟着字母“f”“o”“o”的“任何东西”零次或者多次。由于量词是贪婪的，表达式的.\*部分第一次“吃掉”整个输入的字符串。在这一点，全部表达式不能成功地进行匹配，这是由于最后三个字母（“f”“o”“o”）已经被消耗掉了。那么匹配器会慢慢地每次回退一个字母，直到返还的“foo”在最右边出现，这时匹配成功并且搜索终止。  
　　然而，第二个例子采用勉强量词，因此通过首次消耗“什么也没有”作为开始。由于“foo”并没有出现在字符串的开始，它被强迫吞掉第一个字母（“x”），在 0 和 4 处触发了第一个匹配。测试用具会继续处理，直到输入的字符串耗尽为止。在 4 和 13 找到了另外一个匹配。  
　　第三个例子的量词是侵占，所以在寻找匹配时失败了。在这种情况下，整个输入的字符串被.\*+消耗了，什么都没有剩下来满足表达式末尾的“foo”。  
　　你可以在想抓取所有的东西，且决不回退的情况下使用侵占量词，在这种匹配不是立即被发现的情况下，它将会优于等价的贪婪量词。

**6.1　编号方式**

　　在 Pattern 的 API 描述中，捕获组通过从左至右计算开始的圆括号进行编号。例如，在表达式((A)(B(C)))中，有下面的四组：  
　　1. ((A)(B(C)))  
　　2. (A)  
　　3. (B(C))  
　　4. (C)

**6.2　反向引用**

　　匹配输入字符串的捕获组部分会存放在内存中，通过反向引用（backreferences）稍后再调用。在正则表达式中，反向引用使用反斜线（\）后跟一个表示需要再调用组号的数字来表示。例如，表达式(\d\d)定义了匹配一行中的两个数字的捕获组，通过反向引用\1，表达式稍候会被再次调用。  
　　匹配两个数字，且后面跟着两个完全相同的数字时，就可以使用(\d\d)\1作为正则表达式：

Enter your regex: (\d\d)\1

Enter input string to search: 1212

I found the text "1212" starting at index 0 and ending at index 4.

　　如果更改最后的两个数字，这时匹配就会失败：

Enter your regex: (\d\d)\1

Enter input string to search: 1234

No match found.

　　对于嵌套的捕获组而言，反向引用采用完全相同的方式进行工作，即指定一个反斜线加上需要被再次调用的组号。

**7　边界匹配器**

|  |  |
| --- | --- |
| **边界匹配器** | |
| ^ | 行首 |
| $ | 行尾 |
| \b | 单词边界 |
| \B | 非单词边界 |
| \A | 输入的开头 |
| \G | 上一个匹配的结尾 |
| \Z | 输入的结尾，仅用于最后的结束符（如果有的话） |
| \z | 输入的结尾 |

　　接下来的例子中，说明了^和$边界匹配器的用法。注意上表中，^匹配行首，$匹配行尾。

Enter your regex: ^dog$

Enter input string to search: dog

I found the text "dog" starting at index 0 and ending at index 3.

Enter your regex: ^dog$

Enter input string to search: dog

No match found.

Enter your regex: \s\*dog$

Enter input string to search: dog

I found the text " dog" starting at index 0 and ending at index 15.

Enter your regex: ^dog\w\*

Enter input string to search: dogblahblah

I found the text "dogblahblah" starting at index 0 and ending at index 11.

　　第一个例子的匹配是成功的，这是因为模式占据了整个输入的字符串。第二个例子失败了，是由于输入的字符串在开始部分包含了额外的空格。第三个例子指定的表达式是不限的空格，后跟着在行尾的 dog。第四个例子，需要 dog 放在行首，后面跟的是不限数量的单词字符。

对于检查一个单词开始和结束的边界模式（用于长字符串里子字符串），这时可以在两边使用\b，例如\bdog\b。

Enter your regex: \bdog\b

Enter input string to search: The dog plays in the yard.

I found the text "dog" starting at index 4 and ending at index 7.

Enter your regex: \bdog\b

Enter input string to search: The doggie plays in the yard.

No match found.

　　对于匹配非单词边界的表达式，可以使用\B来代替：

Enter your regex: \bdog\B

Enter input string to search: The dog plays in the yard.

No match found.

Enter your regex: \bdog\B

Enter input string to search: The doggie plays in the yard.

I found the text "dog" starting at index 4 and ending at index 7.

　　对于需要匹配仅出现在前一个匹配的结尾，可以使用\G：

Enter your regex: dog

Enter input string to search: dog dog

I found the text "dog" starting at index 0 and ending at index 3.

I found the text "dog" starting at index 4 and ending at index 7.

Enter your regex: \Gdog

Enter input string to search: dog dog

I found the text "dog" starting at index 0 and ending at index 3.

　　这里的第二个例子仅找到了一个匹配，这是由于第二次出现的“dog”不是在前一个匹配结尾的开始。

3. 问：普通字符和元字符有什么不同？各给出它们的一个例子。

答：正则表达式中的普通字符匹配其本身。元字符是一个特殊的字符，会影响被匹配模式的方式。字母A是一个普通字符。标点符号.是一个元字符，其匹配任意的单字符。

4. 问：如何把元字符表现成像普通字符那样？

答：有两种方法：

* 在元字符前加上反斜线（\）；
* 把元字符置于\Q（开始）\E（结束）的引用表达式中。

<http://www.java3z.com/cwbwebhome/article/article8/Regex/Java.Regex.Tutorial.html>