第 2 章　量词

2.1　一般形式

根据上一章的介绍，可以用字符组`[0-9]`或者`\d`匹配单个数字字符。现在用正则表达式来验证更复杂的字符串，比如大陆地区的邮政编码。

粗略来看，邮政编码并没有特殊的规定，只是 6 位数字构成的字符串，比如 201203、100858，所以用正则表达式来表示就是`\d\d\d\d\d\d`，如例 2-1 所示，只有同时满足“长度是 6 个字符”和“每个字符都是数字”两个条件，匹配才成功（同样，这里不能忽略`^`和`$`）。

例 2-1　匹配邮政编码

```
re.search(r"^\d\d\d\d\d\d$", "100859") != None       #  => True
re.search(r"^\d\d\d\d\d\d$", "201203") != None       #  => True

re.search(r"^\d\d\d\d\d\d$", "20A203") != None       #  => False
re.search(r"^\d\d\d\d\d\d$", "20103") != None        #  => False
re.search(r"^\d\d\d\d\d\d$", "2012036") != None      #  => False
```

虽然这不难理解，但`\d`重复了 6 次，读写都不方便。为此，正则表达式提供了**量词**（quantifier），比如上面匹配邮政编码的表达式，就可以如例 2-2 那样，简写为`\d{6}`，它使用阿拉伯数字，更简洁也更直观。

例 2-2　使用量词简化字符组

```
re.search(r"^\d{6}$", "100859") != None       #  => True
re.search(r"^\d{6}$", "201203") != None       #  => True

re.search(r"^\d{6}$", "20A203") != None       #  => False
re.search(r"^\d{6}$", "20103") != None        #  => False
re.search(r"^\d{6}$", "2012036") != None      #  => False
```

量词还可以表示不确定的长度，其通用形式是`{m,n}`，其中 *m* 和 *n* 是两个数字（有些人习惯在代码中的逗号之后添加空格，这样更好看，但是量词中的逗号之后绝不能有空格），它限定之前的元素[7]能够出现的次数，*m* 是下限，*n* 是上限（均为闭区间）。比如`\d{4,6}`，就表示这个数字字符串的长度最短是 4 个字符（“单个数字字符”至少出现 4 次），最长是 6 个字符。

如果不确定长度的上限，也可以省略，只指定下限，写成`\d{m,}`，比如`\d{4,}`表示“数字字符串的长度必须在 4 个字符以上”。

量词限定的出现次数一般都有明确下限，如果没有，则默认为 0。有一些语言（比如 Ruby）支持

---

[7] 在上一章提到，字符组是正则表达式的基本“结构”之一，而此处提到之前的“元素”，在此做一点解释。在本书中，“结构”一般指的是正则表达式所提供功能的记法。比如字符组就是一种结构，下一章要提到的括号也是一种结构；而“元素”指的是具体的正则表达式中的某个部分，比如某个具体表达式中的字符组`[a-z]`，可以算作一个元素，“元素”也叫“子表达式”（sub-expression）。

`{,n}`的记法，这时候并不是“不确定长度的下限”，而是省略了“下限为 0”的情况，比如`\d{,6}`表示“数字字符串最多可以有 6 个字符”。不过，这种用法并不是所有语言中都通用的，比如 Java 就不支持这种写法，所以必须写明`{0,n}`。我推荐的做法是：最好使用`{0,n}`的记法，因为它是广泛支持的。表 2-1 集中说明了这几种形式的量词，例 2-3 展示了它们的使用。

表 2-1　量词的一般形式

量词	说明
{n}	之前的元素必须出现 *n* 次
{m,n}	之前的元素最少出现 *m* 次，最多出现 *n* 次
{m,}	之前的元素最少出现 *m* 次，出现次数无上限
{0,n}	之前的元素可以不出现，也可以出现，最多出现 *n* 次（在某些语言中可以写为{,n}）

例 2-3　表示不确定长度的量词

```
re.search(r"^\d{4,6}$", "123") != None         #  => False
re.search(r"^\d{4,6}$", "1234") != None        #  => True
re.search(r"^\d{4,6}$", "123456") != None      #  => True
re.search(r"^\d{4,6}$", "1234567") != None     #  => False

re.search(r"^\d{4,}$", "123") != None          #  => False
re.search(r"^\d{4,}$", "1234") != None         #  => True
re.search(r"^\d{4,}$", "123456") != None       #  => True

re.search(r"^\d{0,6}$", "12345") != None       #  => True
re.search(r"^\d{0,6}$", "123456") != None      #  => True
re.search(r"^\d{0,6}$", "1234567") != None     #  => False
```

2.2　常用量词

`{m,n}`是通用形式的量词，正则表达式还有三个常用量词，分别是`+`、`?`、`*`。它们的形态虽然不同于`{m,n}`，功能却是相同的（也可以把它们理解为“量词简记法”），具体说明见表 2-2。

表 2-2　常用量词

常用量词	{m,n}等价形式	说明
*	{0,}	可能出现，也可能不出现，出现次数没有上限
+	{1,}	至少出现 1 次，出现次数没有上限
?	{0,1}	至多出现 1 次，也可能不出现

在实际应用中，在很多情况只需要表示这三种意思，所以常用量词的使用频率要高于`{m,n}`，下面分别说明。

大家都知道，美国英语和英国英语有些词的写法是不一样的，比如 traveler 和 traveller，如果希望“通吃”traveler 和 traveller，就要求第 2 个 l 是“至多出现 1 次，也可能不出现”的，正好使用`?`量词：`travell?er`，如例 2-4 所示。

例 2-4　量词?的应用

```
re.search(r"^travell?er$", "traveler") != None     #  => True
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

```
re.search(r"^travell?er$", "traveller") != None    #  => True
```

其实这样的情况还有很多，比如 favor 和 favour、color 和 colour。此外还有很多其他应用场合，比如 http 和 https，虽然是两个概念，但都是协议名，可以用`https?`匹配；再比如表示价格的字符串，有可能是 100 也有可能是￥100，可以用`￥?100`匹配[8]。

量词也广泛应用于解析 HTML 代码。HTML 是一种“标签语言”，它包含各种各样的 tag（标签），比如<head>、<img>、<table>等，这些 tag 的名字各异，形式却相同：从<开始，到>结束，在<和>之间有若干字符，“若干”的意思是长度不确定，但不能为 0（<>并不是合法的 tag），也不能是>字符[9]。如果要用一个正则表达式匹配所有的 tag，需要用`<`匹配开头的<，用`>`匹配结尾的>，用`[^>]+`匹配中间的“若干字符”，所以整个正则表达式就是`<[^>]+>`，程序如例 2-5 所示。

例 2-5　量词+的应用

```
re.search(r"^<[^>]+>$", "<bold>") != None          #  => True
re.search(r"^<[^>]+>$", "</table>") != None        #  => True
re.search(r"^<[^>]+>$", "<>") != None              #  => False
```

类似的，也可以使用正则表达式匹配双引号字符串。不同的是，双引号字符串的两个双引号之间可以没有任何字符，""也是一个完全合法的双引号字符串，应该使用量词`*`，于是整个正则表达式就成了`"[^"]*"`，程序见例 2-6。

例 2-6　量词*的应用

```
re.search(r"^\"[^\"]*\"$", "\"some\"") != None    #  => True
re.search(r"^\"[^\"]*\"$", "\"\"") != None        #  => True
```
注：字符串之中表示双引号需要转义写成\"，这并不是正则表达式中的规定，而是为字符串转义考虑。

量词的使用有很多学问，不妨多看几个 tag 匹配的例子：tag 可以粗略分为 open tag 和 close tag，比如<head>就是 open tag，而</html>就是 close tag；另外还有一类标签是 self-closing tag，比如
。现在来看分别匹配这三类 tag 的正则表达式。

open tag 的特点是以<开头，然后是“若干字符”（但不能以/开头），最后是>，所以对应的正则表达式是`<[^/][^>]*>`；注意：因为`[^/]`必须匹配一个字符，所以“若干字符”中其他部分必须写成`[^>]*`，否则它无法匹配名字为单个字符的标签，比如<b>。

close tag 的特点是以<开头，之后是/字符，然后是“若干字符（但不能以/开头）”，最后是>，所以对应的正则表达式是`</[^>]+>`；

self-closing tag 的特点是以<开头，中间是“若干字符”，最后是/>，所以对应的正则表达式是`<[^>]+/>`。注意：这里不是`<[^>/]+/>`，排除型字符组只排除>，而不排除/，因为要确认的只是在结尾的>之前出现/，如果写成`<[^>/]+/>`，则要求 tag 内部不能出现/，就无法匹配这类的 tag 了。

表 2-3 列出了匹配几类 tag 的表达式。

---

[8] 实际上，这个问题比较复杂，因为￥并不是一个 ASCII 字符，所以`￥?`可能会产生问题，具体情况请参考第 7 章。

[9] 如果你对 HTML 代码比较了解，可能会有疑问，假如 tag 内部出现>符号，怎么办？这种情况确实存在，比如<input name=txt value=">">。以目前已经讲解的知识还无法解决这个问题，不过下一章就会给出它的解法。

专业图书网上第一专营店
www.china-pub.com
独家提供样章

表 2-3　各类 tag 的匹配

匹配所有 tag 的表达式	tag 分类	匹配分类 tag 的表达式
	open tag	<[^/>][^>]*>
<[^>]+>	close tag	</[^>]+>
	self-closing tag	<[^>/]+/>

对比表格中“匹配所有 tag 的表达式”和“匹配分类 tag 的表达式”，可以发现它们的模式是相近的，只是细节上有差异。也就是说，通过变换字符组和量词，可以准确控制正则表达式能匹配的字符串的范围，达到不同的目的。这其实是使用正则表达式时的一条根本规律：使用合适的结构（包括字符组和量词），精确表达自己的意图，界定能匹配的文本。

再仔细观察，你或许会发现，匹配 open tag 的表达式，也可以匹配 self-closing tag：<[^/][^>]*>能够匹配
，因为[^>]*并不排除对/的匹配。那么将表达式改为<[^/][^>]*[^/]>，就保证匹配的 open tag 不会以/>结尾了。

不过这会产生新的问题：<[^/][^>]*[^/]>能匹配的 tag，在<和>之间出现了两个[^/]，上一章已经讲过，排除型字符组表示“在当前位置，匹配一个没有列出的字符”，所以 tag 里的字符串必须至少包含两个字符，这样就无法匹配<u>了。

仔细想想，真正要表达的意思是，在 tag 内部的字符串不能以/开头，也不能以/结尾，如果这个字符串只包含一个字符，那么它既是开头，又是结尾，使用两个排除型字符组显然是不合适的，看起来没办法解决了。实际上，只是现有的知识还不足够解决这个问题而已，在第**错误！未定义书签。**页有这个问题的详细解法。

2.3　数据提取

正则表达式的功能很多，除去之前介绍的验证（字符串能否由正则表达式匹配），还可以从某个字符串中提取出某个字符串能匹配的所有文本。

上一章提到，re.search()如果匹配成功，返回一个 MatchObject 对象。这个对象包含了匹配的信息，比如表达式匹配的结果，可以像例 2-7 那样，通过调用 *MatchObject*.group(0)来获得。这个方法以后详细介绍，现在只需要了解一点：调用它可以得到表达式匹配的文本。

例 2-7　通过 MatchObject 获得匹配的文本

```
#注意这里使用链式编程
print re.search(r"\d{6}", "ab123456cd").group(0)
123456
print re.search(r"^<[^>]+>$", "<bold>").group(0)
<bold>
```

这里再介绍一个方法：re.findall(*pattern*, *string*)。其中 *pattern* 是正则表达式，*string* 是字符串。这个方法会返回一个数组，其中的元素是在 *string* 中依次寻找 *pattern* 能匹配的文本。

以邮政编码的匹配为例，假设某个字符串中包含两个邮政编码：zipcode1:201203, zipcode2:100859，仍然使用之前匹配邮政编码的正则表达式\d{6}，调用 re.*findall*()可以将这两个邮政编码提取出来，如例 2-8。注意，这次要去掉表达式首尾的^和$，因为要使用正则表达式在字

符串中寻找匹配，而不是验证整个字符串能否由正则表达式匹配。

例 2-8　使用 re.findall()提取数据

```
print re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859")
['201203', '100859']

#也可以逐个输出
for zipcode in re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859"):
    print zipcode
201203
100859
```

借助之前的匹配各种 tag 的正则表达式，还可以通过 `re.findall()`将某个 HTML 页面中所有的 tag 提取出来，下面以 Yahoo 首页为例。

首先要读入 http://www.yahoo.com/的 HTML 源代码，在 Python 中先获得 URL 对应页面的源代码，保存到 *htmlSource* 变量中，然后针对匹配各类 tag 的正则表达式，分别调用 `re.findall()`，获得各类 tag 的列表（因为这个页面中包含的 tag 太多，每类 tag 只显示前 3 个）。

因为这段程序的输出很多，在交互式界面下不方便操作和观察，建议将这些代码单独保存为一个.py 文件，比如 `findtags.py`，然后输入 `python findtags.py` 运行。如果输入 `python` 没有结果（一般在 Windows 下会出现这种情况），需要准确设定 `PATH` 变量，比如 `d:\Python\python`。之后，就会看到例 2-9 显示的结果。

例 2-9　使用 re.findall()提取 tag

```
#导入需要的 package
import urllib
import re
#读入 HTML 源代码
sock = urllib.urlopen("http://yahoo.org/")
htmlSource = sock.read()
sock.close()
#匹配，输出结果（[0:3]表示取前 3 个）
print "open tags:"
print re.findall(r"<[^/>][^>]*[^/>]>", htmlSource)[0:3]
print "close tags:"
print re.findall(r"</[^>]+>", htmlSource) [0:3]
print "self-closing tags:"
print re.findall(r"<[^>/]+/>", htmlSource) [0:3]

open tags:
['<!DOCTYPE html>', '<html lang="en-US" class="y-fp-bg y-fp-pg-grad  bkt701">',
'<!-- m2 template 0 -->']
close tags:
['</title>', '</script>', '</script>']
self-closing tags:
['<br/>', '<br/>', '<br/>']
```

2.4　点号

上一章讲到了各种字符组，与它相关的还有一个特殊的元字符：点号`.`。一般文档都说，点号可

以匹配“任意字符”，点号确实可以匹配“任意字符”，常见的数字、字母、各种符号都可以匹配，如例 2-10 所示。

例 2-10　点号.的匹配

```
re.search(r"^.$", "a") != None       #  => True
re.search(r"^.$", "0") != None       #  => True
re.search(r"^.$", "*") != None       #  => True
```

有一个字符不能由点号匹配，就是换行符\n。这个字符平时看不见，却存在，而且在处理时并不能忽略（下一章会给出具体的例子）。

如果非要匹配“任意字符”，有两种办法：可以指定使用单行匹配模式，在这种模式下，点号可以匹配换行符（☞**错误！未定义书签。**）；或者使用上一章的介绍“自制”通配字符组[\s\S]（也可以使用[\d\D]或[\w\W]），正好涵盖了所有字符。例 2-11 清楚地说明，这两个办法都可以匹配换行符。

例 2-11　换行符的匹配

```
re.search(r"^.$", "\n") != None          #  => False
#单行模式
re.search(r"(?s)^.$", "\n") != None      #  => True
#自制“通配字符组”
re.search(r"^[\s\S]$", "\n") != None     #  => True
```

2.5　滥用点号的问题

因为点号能匹配几乎所有的字符，所以实际应用中许多人图省事，随意使用.*或.+，结果却事与愿违，下面以双引号字符串为例来说明。

之前我们使用表达式"[^"]*"匹配双引号字符串，而“图省事”的做法是".*"。通常这么用是没有问题的，但也可能有意外，例 2-12 就说明了一种如此。

例 2-12　“图省事”的意外结果

```
#字符串的值是"quoted string"
print re.search(r"\".*\"", "\"quoted string\"").group(0)
"quoted string"
#字符串的值是"quoted string" and another"
print re.search(r"\".*\"", "\"quoted string\" and another\"").group(0)
"quoted string" and another"
```

用".*"匹配双引号字符串，不但可以匹配正常的双引号字符串"quoted string"，还可以匹配格式错误的字符串"quoted string" and another"。这是为什么呢？

这个问题比较复杂，现在只简要介绍，以说明图省事导致错误的原因，更深入的原因涉及正则表达式的匹配原理，在第 8 章详细介绍。

在正则表达式".*"中，点号.可以匹配任何字符，*表示可以匹配的字符串长度没有限制，所以.*在匹配过程结束以前，每遇到一个字符（除去无法匹配的\n），.*都可以匹配，但是到底是匹配这个字符，还是忽略它，将其交给之后的"来匹配呢？

答案是，具体选择取决于所使用的量词。在正则表达式中的量词分为几类，之前介绍的量词都可

专业图书网上第一专营店 www.china-pub.com 独家提供样章

以归到一类，叫做**匹配优先量词**（greedy quantifier，也有人翻译为**贪婪量词**[10]）。匹配优先量词，顾名思义，就是在拿不准是否要匹配的时候，优先尝试匹配，并且记下这个状态，以备将来“反悔”。

来看表达式".*"对字符串"quoted string"的匹配过程。

一开始，"匹配"，然后轮到字符 q，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 q，并且记录下这个状态【q 也可能是.*不应该匹配的】;

接下来是字符 u，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 u，并且记录下这个状态【u 也可能是.*不应该匹配的】;

……

现在轮到字符 g，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 g，并且记录下这个状态【g 也可能是.*不应该匹配的】;

最后是末尾的"，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配"，并且记录下这个状态【"也可能是.*不应该匹配的】。

这时候，字符串之后已经没有字符了，但正则表达式中还有"没有匹配，所以只能查询之前保存备用的状态，看看能不能退回几步，照顾"的匹配。查询到最近保存的状态是:【"也可能是.*不应该匹配的】。于是让.*“反悔”对"的匹配，把"交给"，测试发现正好能匹配，所以整个匹配宣告成功。这个“反悔”的过程，专业术语叫做**回溯**（backtracking），具体的过程如图 2-1 所示。

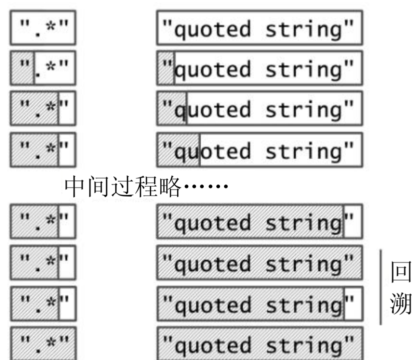

图 2-1 表达式".*"对字符串"quoted string"的匹配过程

".*" "quoted string" and another"

中间过程略……

如果把字符串换成"quoted string" and another"，.*会首先匹配第一个双引号之后的所有字符，再进行回溯，表达式中的"匹配了字符串结尾的字符"，整个匹配宣告完成，过程如图 2-2 所示。

图 2-2 表达式".*"的匹配过程

[10] 许多文档都翻译为“贪婪量词”，单独来看这是没问题的，但考虑到正则表达式中还有其他类型的量词，其英文名字的形式较为统一，所以我在翻译《精通正则表达式》时采用了“匹配优先/忽略优先/占有优先”的名字，也未见读者反对，故此处延用此译法。

互动出版网 China-Pub.com
专业图书网上第一专营店
www.china-pub.com 独家提供样章

```
"[^"]*"    "quoted string" and another"
"[^"]*"    "quoted string" and another"
"[^"]*"    "quoted string" and another"
"[^"]*"    "quoted string" and another"
中间过程略……
"[^"]*"    "quoted string" and another"
"[^"]*"    "quoted string" and another"
```

图 2-3　表达式"[^"]*"的匹配过程

如果要准确匹配双引号字符串，就不能图省事使用`".*"`，而要使用`"[^"]*"`，过程如图 2-3 所示。

中间过程略……

2.6　忽略优先量词

也有些时候，确实需要用到`.*`（或者`[\s\S]*`），比如匹配 HTML 代码中的 JavaScript 示例就是如此。

```
<script type="text/javascript">…</script>
```

匹配的模式仍然是：匹配 open tag 和 close tag，以及它们之间的内容。open tag 是<script type="text/javascript">，close tag 是</script>，这两段的内容是固定的，非常容易写出对应的表达式，但之间的内容怎么匹配呢？在 JavaScript 代码中，各种字符都可能出现，所以不能用排除型字符组，只能用`.*`。比如，用一个正则表达式匹配下面这段 HTML 源代码：

```
<script type="text/javascript">
alert("some punctuation <>/");
</script>
```

开头和结尾的 tag 都容易匹配，中间的代码要比较麻烦，因为点号`.`不能匹配换行符，所以必须使用`[\s\S]`（或者`[\d\D]`、`[\w\W]`）。

```
<script type="text/javascript">[\s\S]*</script>
```

这个表达式确实可以匹配上面的 JavaScript 代码。但是如果遇到更复杂的情况就会出错，比如针对下面这段 HTML 代码，程序运行结果如例 2-13。

```
<script type="text/javascript">
alert("1");
</script>
<br />
<script type="text/javascript">
alert("2");
</script>
```

例 2-13　匹配 JavaScript 代码的错误

```
#假设上面的 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*</script>"
print re.search(jsRegex, htmlSource).group(0)
```

<script type="text/javascript">

```
alert("1");
</script>
<br />
<script type="text/javascript">
alert("2");
</script>
```

用`<script type="text/javascript">[\s\S]*</script>`来匹配，会一次性匹配两段 JavaScript 代码，甚至包含之间的非 JavaScript 代码。

按照匹配原理，`[\s\S]*`先匹配所有的文本，回溯时交还最后的`</script>`，整个表达式的匹配就成功了，逻辑就是如此，无可改进。而且，这个问题也不能模仿之前双引号字符串匹配，用`[^"]*`匹配`<script...>`和`</script>`之间的代码，因为排除型字符组只能排除单个字符，`[^</script>]`不能表示“不是</script>的字符串”。

换个角度来看，通过改变`[\s\S]*`的匹配策略解决问题：在不确定是否要匹配的场合，先尝试不匹配的选择，测试正则表达式中后面的元素，如果失败，再退回来尝试`.*`匹配，如此就没问题了。

循着这个思路，正则表达式中还提供了**忽略优先量词**（lazy quantifier 或 reluctant quantifier，也有人翻译为**懒惰量词**），如果不确定是否要匹配，忽略优先量词会选择“不匹配”的状态，再尝试表达式中之后的元素，如果尝试失败，再回溯，选择之前保存的“匹配”的状态。

对`[\s\S]*`来说，把`*`改为`*?`就是使用了忽略优先量词，`*?`限定的元素出现次数范围与`*`完全一样，都表示“可能出现，也可能不出现，出现次数没有上限”。区别在于，在实际匹配过程中，遇到`[\s\S]`能匹配的字符，先尝试“忽略”，如果后面的元素（具体到这个表达式中，是`</script>`）不能匹配，再尝试“匹配”，这样就保证了结果的正确性，代码见例 2-14。

例 2-14　准确匹配 JavaScript 代码

```
#仍然假设 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
print re.search(jsRegex, htmlSource) .group(0)

<script type="text/javascript">
alert("1");
</script>

#甚至也可以逐次提取出两段 JavaScript 代码
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
for jsCode in re.findall(jsRegex, htmlSource) :
 print jsCode + "\n"

<script type="text/javascript">
alert("1");
</script>

<script type="text/javascript">
alert("2");
</script>
```

从表 2-4 可以看到，匹配优先量词与忽略优先量词逐一对应，只是在对应的匹配优先量词之后添加`?`，两者限定的元素能出现的次数也一样，遇到不能匹配的情况同样需要回溯；唯一的区别在于，忽略优先量词会优先选择“忽略”，而匹配优先量词会优先选择“匹配”。

表 2-4　匹配优先量词与忽略优先量词

匹配优先量词	忽略优先量词	限定次数
*	*?	可能不出现，也可能出现，出现次数没有上限
+	+?	至少出现 1 次，出现次数没有上限
?	??	至多出现 1 次，也可能不出现
{m,n}	{m,n}?	出现次数最少为 *m* 次，最多为 *n* 次
{m,}	{m,}?	出现次数最少为 *m* 次，没有上限
{,n}	{,n}?	可能不出现，也可能出现，最多出现 *n* 次

忽略优先量词还可以完成许多其他功能，典型的例子就是提取代码中的 C 语言注释。

C 语言的注释有两种：一种是在行末，以//开头；另一种可以跨多行，以/*开头，以*/结束。第一种注释很好匹配，使用`//.*`即可，因为点号`.`不能匹配换行符，所以`//.*`匹配的就是从//直到行末的文本，注意这里使用了量词`*`，因为//可能就是该行最后两个字符；第二种注释稍微复杂一点，因为/*…*/的注释和 JavaScript 一样，可能分成许多段，所以必须用到忽略优先量词；同时因为注释可能横跨多行，所以必须使用`[\s\S]`。因此，整个表达式就是`/*[\s\S]*?*/`（别忘了`*`的转义）。

另一个典型的例子是提取出 HTML 代码中的超链接。常见的超链接形似text。它以<a 开头，以结束，*href* 属性是超链接的地址。我们无法预先判断<a>和之间到底会出现哪些字符，不会出现哪些字符，只知道其中的内容一直到结束[11]，程序代码见例 2-15。

例 2-15 提取网页中所有的超链接 tag

```
#仍然获得 yahoo 网站的源代码，存放在 htmlSource 中
for hyperlink in re.findall(r"<a\s[\s\S]+?</a>", htmlSource):
 print hyperlink
#更多结果未列出
<a href="http://search.yahoo.com/">Web</a>
<a href="http://images.search.yahoo.com/images">Images</a>
<a href="http://video.search.yahoo.com/video">Video</a>
```

值得注意的是，在这个表达式中的`<a `之后并没有使用普通空格，而是使用字符组简记法`\s`。HTML 语法并没有规定此处的空白只能使用空格字符，也没有规定必须使用一个空白字符，所以我们用`\s `保证“至少出现一个空白字符”（但是不能没有这个空白字符，否则就不能保证匹配 tag name 是 a）。

之前匹配 JavaScript 的表达式是`<script language="text/javascript">[\s\S]*?</script>`，它能应对的情况实在太少了：在<script 之后可能不是空格，而是空白字符；再之后可能是 type="text/javascript"，也可能是 type="application/javascript"，也可能用 language 取代 type（实际上 language 是以前的写法，现在大都用 type），甚至可能没有属性，直接是<script>[12]。

所以必须改造这个表达式，将条件放宽：在 script 之后，可能出现空白字符，也可能直接是>，这部分可以用一个字符组`[\s>]`来匹配，之后的内容统一用`[\s\S]+?`匹配，忽略优先量词保证了匹配进行到到最近的</script>为止。最终得到的表达式就是`<script[\s>] [\s\S]+?</script>`。

[11] 根据 HTML 规范，<a>这个 tag 可用来表示超链接，也可以用作书签，或兼作两种用途，考虑到书签的情况很少见，这里没有做特殊处理。

[12] 严格说起来，如果只出现<script>，无法保证这里出现的就是 JavaScript 代码，也可能是 VBScript 代码，但考虑到真实世界中的情况，基本可以认为<script 标识的“就是”JavaScript 代码，所以这里不作区分。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

对这个表达式稍加改造，就可以写出匹配类似 tag 的表达式。在解析页面时，常见的需求是提取表格中各行、各单元（cell）的内容。表格的 tag 是<tag>，行的 tag 是<tr>，单元的 tag 是<td>，所以，它们可以分别用下面的表达式匹配，请注意其中的[\s>]，它兼顾了可能存在的其他属性（比如<table border="1">），同时排除了可能的错误（比如<tablet>）。

匹配 table	`<table[\s>][\s\S]+?</table>`
匹配 tr	`<tr[\s>][\s\S]+?</tr>`
匹配 td	`<td[\s>][\s\S]+?</td>`

在实际的 HTML 代码中，table、tr、td 这三个元素经常是嵌套的，它们之间存在着包含关系。但是，仅仅使用正则表达式匹配，并不能得到“某个 table 包含哪些 tr”、“某个 td 属于哪个 tr”这种信息。此时需要像例 2-16 的那样，用程序整理出来。

例 2-16　用正则表达式解析表格

```
# 这里用到了 Python 中的三重引号字符串，以便字符串跨越多行，细节可参考第 14 章
htmlSource = """<table>
<tr><td>1-1</td></tr>
<tr><td>2-1</td><td>2-2</td></tr>
</table>"""

for table in re.findall(r"<table[\s>][\s\S]+?</table>", htmlSource):
 for tr in re.findall(r"<tr[\s>][\s\S]+?</tr>", table):
      for td in re.findall(r"<td[\s>][\s\S]+?</td>", tr):
           print td,
      #输出一个换行符，以便显示不同的行
      print ""
<td>1-1</td>

<td>2-1</td> <td>2-2</td>
```

注：因为 tag 是不区分大小写的，所以如果还希望匹配大写的情况，则必须使用字符组，table 写成[tT][aA][bB][lL][eE]，tr 写成[tT][rR]，td 写成[tT][dD]。

这个例子说明，正则表达式只能进行纯粹的文本处理，单纯依靠它不能整理出层次结构；如果希望解析文本的同时构建层次结构信息，则必须将正则表达式配合程序代码一起使用。

回过头想想双引号字符串的匹配，之前使用的正则表达式是"[^"]*"，其实也可以使用忽略优先量词解决".*?"（如果双引号字符串中包含换行符，则使用"[\s\S]*?"）。两种办法相比，哪个更好呢？

一般来说，"[^"]*"更好。首先，[^"]本身能够匹配换行符，涵盖了点号.可能无法应付的情况，出于习惯，很多人更愿意使用点号.而不是[\s\S]；其次，匹配优先量词只需要考虑自己限定的元素能否匹配即可，而忽略优先量词必须兼顾它所限定的元素与之后的元素，效率自然大大降低，如果字符串很长，两者的速度可能有明显的差异。

而且，有些情况下确实必须用到匹配优先量词，比如文件名的解析就是如此。UNIX/Linux 下的文件名类似这样/usr/local/bin/python，它包含两个部分：路径是/usr/local/bin/；真正的文件名是 python。为了在/usr/local/bin/python 中解析出两个部分，使用匹配优先量词是非常方便的。从字符串的起始位置开始，用.*/匹配路径，根据之前介绍的知识，它会回溯到最后（最右）的斜线字符/，也就是文件名之前；在字符串的结尾部分，[^/]*能匹配的就是真正的文件名。前一章介绍过^和$，它们分别表示“定位到字符串的开头”和“定位到字符串的结尾”，所以应该把^加在匹配路径

的表达式之前，得到^.*/，而把$加在匹配真正文件名的表达式之后，得到[^/]*$，代码见例 2-17。

例 2-17　用正则表达式拆解 Linux/UNIX 的路径

```
print re.search(r"^.*/", "/usr/local/bin/python").group(0)
/usr/local/bin

print re.search(r"[^/]*$", "/usr/local/bin/python").group(0)
python
```

Windows 下的路径分隔符是\，比如 C:\Program Files\Python 2.7.1\python.exe，所以在正则表达式中，应该把斜线字符/换成反斜线字符\。因为在正则表达式中反斜线字符\是用来转义其他字符的，为了表示反斜线字符本身，必须连写两个反斜线，所以两个表达式分别改为^.*\\和[^\\]*$，代码见例 2-18。

例 2-18　用正则表达式拆解 Windows 的路径

```
#反斜线\必须转义写成\\
print re.search(r"^.*\\", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
C:\Program Files\Python 2.7.1\

print re.search(r"[^\\]*$", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
python.exe
```

2.7　转义

前面讲解了匹配优先量词和忽略优先量词，现在介绍量词的转义[13]。

在正则表达式中，*、+、?等作为量词的字符具有特殊意义，但有些情况下只希望表示这些字符本身，此时就必须使用转义，也就是在它们之前添加反斜线\。

对常用量词所使用的字符+、*、?来说，如果希望表示这三个字符本身，直接添加反斜线，变为\+、*、\?即可。但是在一般形式的量词{m,n}中，虽然具有特殊含义的字符不止一个，转义时却只需要给第一个{添加反斜线即可，也就是说，如果希望匹配字符串{m,n}，正则表达式必须写成\{m,n}。

另外值得一提的是忽略优先量词的转义，虽然忽略优先量词也包含不只一个字符，但是在转义时却不像一般形式的量词那样，只转义第一个字符即可，而需要将两个量词全部转义。举例来说，如果要匹配字符串*?，正则表达式就必须写作*\?，而不是*?，因为后者的意思是“*这个字符可能出现，也可能不出现”。

表 2-5 列出了常用量词的转义形式。

表 2-5　各种量词的转义

量词	转义形式
{n}	\{n}
{m,n}	\{m,n}
{m,}	\{m,}

[13] Java 等语言还支持“占有优先量词（possessive quantifier）”，但这种量词较复杂，使用也不多，所以本书中不介绍占有优先量词。

专业图书网上第一专营店
www.china-pub.com
独家提供样章

（续表）

量词	转义形式
{,n}	\{,n}
*	*
+	\+
?	\?
*?	*\?
+?	\+\?
??	\?\?

之前还介绍了点号`.`，所以还必须讲解点号的转义：点号`.`是一个元字符，它可以匹配除换行符之外的任何字符，所以如果只想匹配点号本身，必须将它转义为`\.`。

因为未转义的点号可以匹配任何字符，其中也可以包含点号，所以经常有人忽略了对点号的转义。如果真的这样做了，在确实需要严格匹配点号时就可能出错，比如匹配小数（如`3.14`）、IP 地址（如`192.168.1.1`）、E-mail 地址（如`someone@somehost.com`）。所以，如果要匹配的文本包含点号，一定不要忘记转义正则表达式中的点号，否则就有可能出现例 2-19 那样的错误。

例 2-19　忽略转义点号可能导致错误

```
#错误判断浮点数
print re.search(r"^\d+.\d+$", "3.14") != None      #  => True
print re.search(r"^\d+.\d+$", "3a14") != None      #  => True
#准确判断浮点数
print re.search(r"^\d+\.\d+$", "3.14") != None     #  => True
print re.search(r"^\d+\.\d+$", "3a14") != None     #  => False
```