第 1 篇　正则表达式基础

第 1 章　外行看正则表达式

正则表达式在程序设计语言中有着广泛的应用，通常用来处理字符串，如匹配字符串、查找字符串、替换字符串等。可以说，正则表达式是一段文本或一个公式，它是用来描述用某种模式去匹配一类字符串的公式，并且该公式具有一定的模式。本章讲解正则表达式的基本规则。

1.1　什么是正则表达式

正则表达式（Regular Expression）起源于人类神经系统的早期研究。神经生理学家 Warren McCulloch 和 Walter Pitts 研究出一种使用数学方式描述神经网络的方法。1956 年，数学家 Stephen Kleene 发表了一篇标题为“神经网事件的表示法”的论文，并在该论文中引入了“正则表达式”这一概念。该论文称正则表达式是：“正则集的代数”的表达式，因此，采用“正则表达式”这个术语。正则表达式的定义存在多种说法，具体如下。

- 正则表达式就是用某种模式去匹配一类字符串的公式，主要用来描述字符串匹配的工具。
- 正则表达式描述了一种字符串匹配的模式。它可以用来检查字符串中是否含有某种子串、将匹配的子串做替换或者从某个字符串中取出符合某个条件的子串等。
- 正则表达式是由普通字符（如字符 a~z）和特殊字符（称为元字符）组成的文字模式。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。
- 正则表达式就是用于描述某些规则的工具，这些规则通常用于处理字符串中的查找或替换字符串。换句话说，正则表达式就是记录文本规则的代码。
- 正则表达式就是用一个“字符串”来描述一个特征，然后去验证另一个“字符串”是否符合这个特征。

学过《编译原理》的读者可能知道不确定有限自动机（Non-deterministic finite automaton，简称 NFA）和确定有限自动机（Deterministic finite automaton，简称 DFA）。其实，正则表达式是一个不确定有限自动机。NFA 和 DFA 的最大区别在于它们的状态转换函数。NFA 可以对同一个字符串产生多种理解方式，而 DFA 则只有唯一的一种理解方式。也正因为如此，NFA 在匹配过程中可能会回溯，NFA 的效率一般要低于 DFA。因此，在书写正则表达式时应尽量减少回溯来提高正则表达式的效率。

如果你在 Windows 或 DOS 下使用过用于文件查找的通配符*或?，那么就不难理解正则表达式。如果需要查找所有 Word 文档，那么你可能使用表达式*.doc，其中，字符*是一个通配符，它可以代表任意字符串。正则表达式和通配符具有相似性，它可以使用一些字符（如字符.）表示任意字符，并且，它比通配符更具有精确性。

在正则表达式中，匹配是最常用的一个词语，它描述了正则表达式的动作结果。给定一段文

本或字符串，使用正则表达式从文本或字符串中查找出符合正则表达式的字符串。有可能文本或字符存在不止一个部分满足给定的正则表达式，这时每一个这样的部分都被称为一个匹配。匹配分为以下三种类型。

- 形容词性的匹配，即一个字符串匹配一个正则表达式。
- 动词性的匹配，即在文本或字符串里匹配正则表达式。
- 名词性的匹配，即字符串中满足给定的正则表达式的一部分。

正则表达式的应用非常广泛，特别是在字符串处理方面。目前，正则表达式在很多软件中都已得到广泛应用，如 Linux、UNIX、HP 等运算系统，C#、PHP、Java 等程序开发环境，以及在很多的应用软件中，都可以看到正则表达式的不同应用。正则表达式常见的应用如下。

- 验证字符串，即验证给定的字符串或子字符串是否符合指定特征，例如，验证是否是合法的邮件地址、验证是否为合法的 HTTP 地址等。
- 查找字符串，从给定的文本中查找符合指定特征的字符串，比查找固定字符串更加灵活方便。
- 替换字符串，即把给定字符串中的符合指定特征的子字符串替换为其他字符串，比普通的替换更强大。
- 提取字符串，即从给定的字符串中提取符合指定特征的子字符串。

1.2 本书使用的测试工具

创建一个正则表达式之后，需要测试该正则表达式是否正确。本书中，使用正则表达式测试工具“Code Architects Regex Tester”来测试正则表达式。这是一个专门用来测试正则表达式的工具，它的初始界面如图 1.1 所示。此时，该工具包含 4 个区域：Regex、Replace、Source 和 Matches。其中，Regex 区域用来书写正则表达式，Replace 区域用来书写替换相关的表达式，Source 区域用来输入被测试的字符串或文本，Matches 区域则显示测试或匹配的结果。

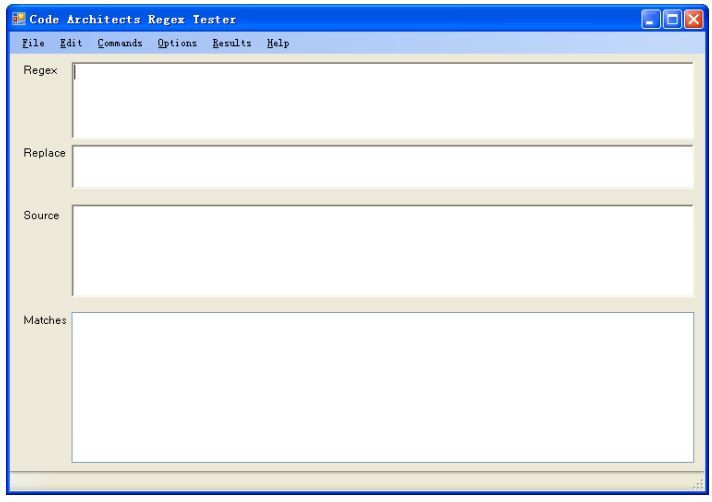

图 1.1 Code Architects Regex Tester 工具初始界面

单击“Results”→“Auto”命令，运算如图 1.2 所示。Code Architects Regex Tester 工具只显示了三个区域：Regex、Source 和 Matches，如图 1.3 所示。

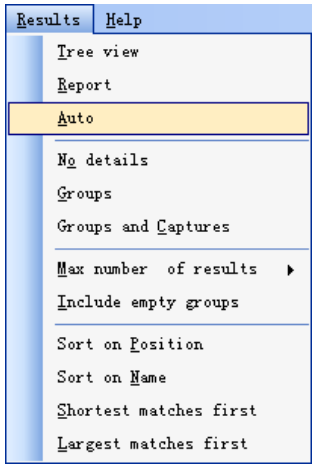

图 1.2　单击“Auto”命令

图 1.3　Code Architects Regex Tester 工具只显示 3 个区域的界面

使用 Code Architects Regex Tester 工具测试正则表达式\d+。其中，被测试的字符串为“0123456789”。测试结果如图 1.4 所示。

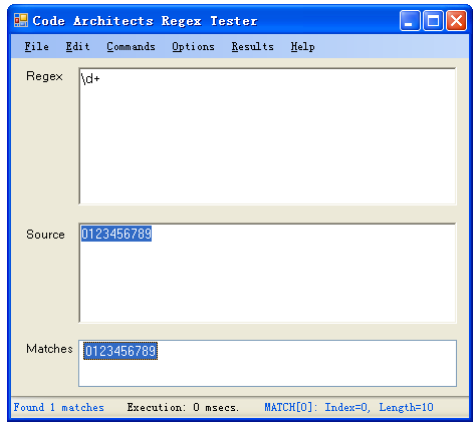

图 1.4　测试正则表达式“\d+”

1.3　理解元字符

在正则表达式中，元字符（Metacharacter）是一类非常特殊的字符，它能够匹配一个位置或字符集合中的一个字符，如.、\w 等。根据功能，元字符可分为两种类型：匹配位置的元字符和匹配字符的元字符。

1.3.1　匹配位置的元字符

匹配位置的元字符包括 3 个字符：^、$和\b。其中，^（脱字符号，通常在文章中插入字时使用）和$（美元符号）只匹配一个位置，它们分别匹配行的开始和结尾。

以下正则表达式匹配以“String”开始的行，即被匹配的行的第一个字符串为“String”。

```
^String                                                    (1)
```

以下正则表达式匹配以“String”结尾的行，即被匹配的行的最后一个字符串为“String”。

```
String$                                                    (2)
```

以下正则表达式匹配以“String”开始和结尾的行，即被匹配的行的第一个字符串和最后一个字符串都为“String”。实际上，该行只包含字符串“String”。

```
^String$                                                   (3)
```

以下正则表达式匹配一个空行，该行中不包含任何字符串。

```
^$                                                         (4)
```

以下正则表达式匹配任意行。该表达式只匹配行中的开始位置，因为任意行都包括其开始位置，所以该表达式将匹配任意行。

```
^                                                          (5)
```

元字符\b 和^、$具有相似性，它也是匹配一个位置。\b 可以匹配单词的开始或结尾，即单词的分界处。通常情况下，英文单词之间往往由空格符号、标点符号或换行符号来分隔，但是元字符\b 不匹配空格符号、标点符号和换行符号中的任何一个，它仅仅匹配一个位置。

以下正则表达式匹配以“Str”开头的字符串，如“String”、“String Format”等。

```
\bStr                                                      (6)
```

正则表达式\bStr 匹配的字符串必须以“Str”开头，并且“Str”之前是单词的分界处。正则表达式\bStr 不能描述或限定“Str”之后的字符串的形式。以下正则表达式匹配以“ing”结尾的字符串，如“String”、“This is a String”等。

```
ing\b                                                      (7)
```

正则表达式 ing\b 匹配的字符串必须以“ing”结尾，且“ing”之后是单词的分界处。以下正则表达式匹配一个完整的单词“String”。

```
\bString\b                                                 (8)
```

> **注意**：在某些特定环境或语言下，还可以分别采用\<和\>来匹配单词的开始位置和结束位置。它们在效果上和元字符\b 等效，即都匹配单词的边界的两个位置，即开始位置和结束位置。

1.3.2 匹配字符的元字符

匹配字符的元字符包括 7 个字符：.（点号）、\w、\W、\s、\S、\d 和\D。其中，.（点号）匹配除换行符之外的任意字符；\w 匹配单词字符（包括字母、数字、下画线和汉字）；\W 匹配任意的非单词字符；\s 匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等；\S 匹配任意的非空白字符；\d 匹配任意的数字；\D 匹配任意的非数字字符。

以下正则表达式匹配一个非空行，该行中可以包含除换行符之外的任意字符。

```
^.$                                                        (9)
```

以下正则表达式匹配一个非空行，且该行中只能包含字母、数字、下画线和汉字中的任意字符。

```
^\w$                                                       (10)
```

以下正则表达式匹配以字母“a”开头的长度等于 8 的任意单词。

```
\ba\w\w\w\w\w\w\w\b                                        (11)
```

正则表达式\ba\w\w\w\w\w\w\w\b 匹配单词“anterior”的方式如图 1.5 所示。

以下正则表达式匹配以字母“a”开头、后跟随形如“3 个字符”＋“3 个字符”＋“1 个非数字字符”、长度等于 8 的任意单词。

```
\ba\w\w\w\d\d\d\D\b                                        (12)
```

正则表达式\ba\w\w\w\d\d\d\D\b 匹配字符串“ante123_”的方式如图 1.6 所示。

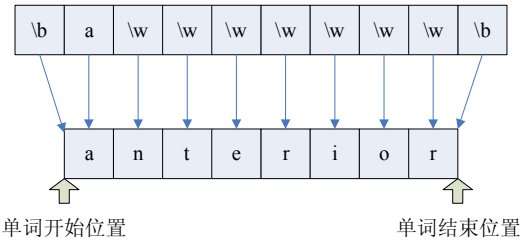

图 1.5　\ba\w\w\w\w\w\w\w\b 匹配单词“anterior”的方式

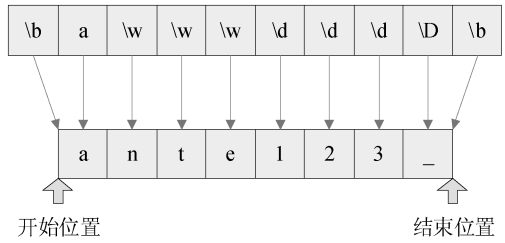

图 1.6　\ba\w\w\w\d\d\d\D\b 匹配字符串“ante123_”的方式

1.3.3　元字符总结

正则表达式的常用元字符有^、$、\b、.、\w、\W、\s、\S、\d 和\D，它们的功能描述说明如表 1.1 所示。

表 1.1　常用元字符

字符	说明
^	匹配行的开始位置
$	匹配行的结束位置
\b	匹配单词的开始或结束位置
.	匹配除换行符之外的任意字符
\w	匹配单词字符（包括字母、数字、下画线和汉字）
\W	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
\s	匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等
\S	匹配任意的非空白字符
\d	匹配任意的数字
\D	匹配任意的非数字字符

元字符.能够匹配除换行符之外的任意字符，如大写字母、小写字母、数字、_（下画线）等。以下正则表达式匹配除换行符之外的以任何字符分割字符串“2007”、“06”、“22”的字符串。

```
2007.06.22                                                    (13)
```

元字符\W 能够匹配除单词字符之外的任意字符。以下正则表达式匹配长度为 2 的字符串，且该字符串不包括单词字符。

```
\W\W                                                          (14)
```

使用工具 Regex Tester 测试正则表达式 2007.06.22，结果如图 1.7 所示。使用工具 Regex Tester 测试正则表达式\W\W，结果如图 1.8 所示。在图 1.8 所示结果中，匹配了 3 个结果：“?”、“*”和“**”。在第一个结果中，正则表达式\W\W 中的第一个\W 匹配字符“?”的上一行的换行符号，第二个\W 才匹配字符“?”。在第三个结果中，正则表达式\W\W 中的每一个\W 都匹配字符“*”。

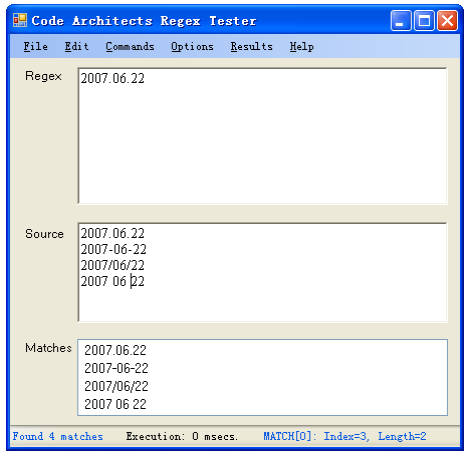

图 1.7 测试正则表达式 2007.06.22

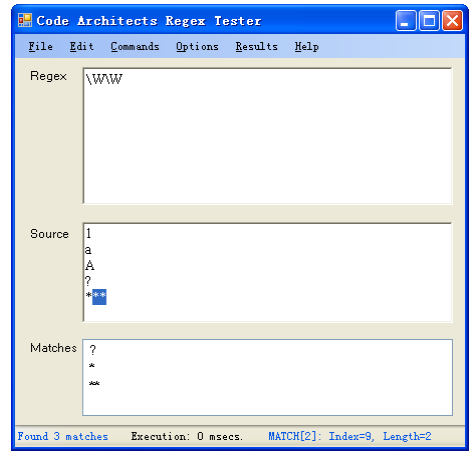

图 1.8 测试正则表达式\W\W

元字符\s 能够匹配空白字符，如空格、制表符、换行符、中文全角空格等。以下正则表达式首先匹配一个单词字符，然后匹配一个空白字符，最后匹配一个单词字符。

```
\w\s\w                                                    (15)
```

元字符\S 能够匹配非空白字符，即除空格、制表符、换行符、中文全角空格等字符之外的字符。以下正则表达式首先匹配一个非空白字符，然后匹配一个空白字符，最后匹配一个非空白字符。

```
\S\s\S                                                    (16)
```

使用工具 Regex Tester 分别测试正则表达式\w\s\w 和\S\s\S，结果分别如图 1.9 和图 1.10 所示。\S 和\w 都能够匹配单词字符，但是，\S 能够匹配除了单词字符之外的字符，如字符“/”和“*”等。

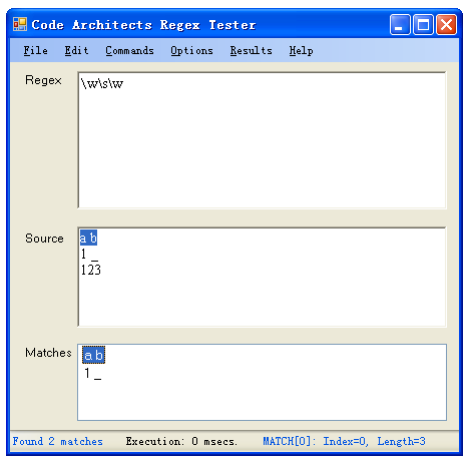

图 1.9 测试正则表达式\w\s\w

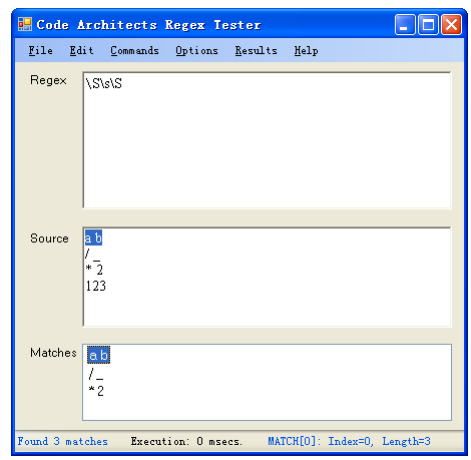

图 1.10 测试正则表达式\S\s\S

元字符\d 能够匹配 0～9 中的任意数字。以下正则表达式匹配 1 位的整数（即小于 10 的整数）。

```
\d                                                        (17)
```

以下正则表达式匹配 3 位的整数（即大于 99 小于 1000 的整数）。

```
\d\d\d                                                    (18)
```

元字符\D 能够匹配除 0～9 之外的任何字符。以下正则表达式匹配以数字开头、非数字字符结尾的字符串，且数字字符是该字符串的第一个字符。

```
\b\d\D                                                    (19)
```

使用工具 Regex Tester 分别测试正则表达式\d\d\d 和\b\d\D，结果分别如图 1.11 和图 1.12 所示。

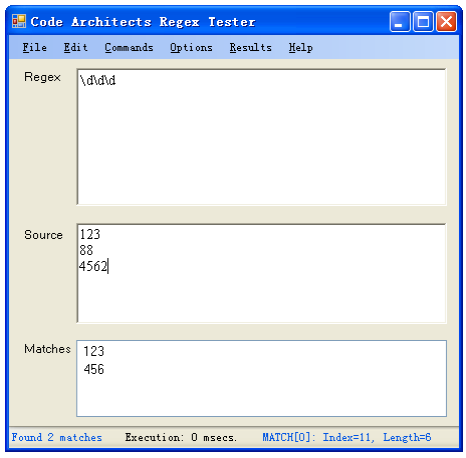

图 1.11　测试正则表达式\d\d\d

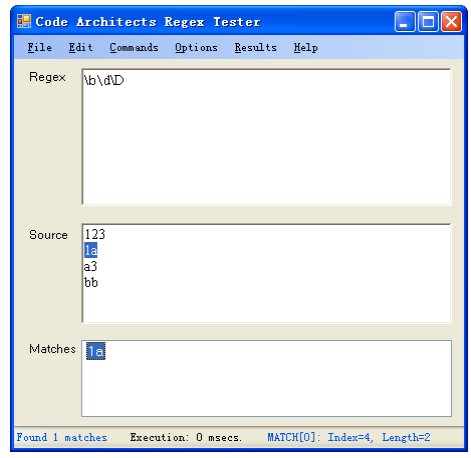

图 1.12　测试正则表达式\b\d\D

1.4　文字匹配

1.4.1　字符类

在正则表达式中，元字符通常一次只能匹配一个位置或字符集合中的一个字符。通常情况下，如果要匹配数字、字母、空白等字符时，可以直接使用与这些集合相对应的元字符。然而，如果要匹配的字符集合（如集合[0,1,2,3,4,5]）没有与之相对应的元字符时，则需要自定义匹配的字符集合。此时，可以使用字符类解决这个问题。字符类是一个字符集合，如果该字符集合中的任何一个字符被匹配，则它就会找到该匹配项。

字符类是正则表达式中的“迷你”语言，可以在方括号“[]”中定义。最简单的字符类由方括号“[]”和一个字母表构成，如元音字符类[aeiou]。以下正则表达式匹配数字 0、1、2、3、4、5、6 中的任何一个。

```
[0123456]                                                    (20)
```

以下正则表达式可以匹配任何数字（即 0、1、2、3、4、5、6、7、8、9）。

```
[0123456789]                                                 (21)
```

以下正则表达式匹配 HTML 标记中的“<H1>”、“<H2>”、“<H3>”、“<H4>”、“<H5>”或“<H6>”。

```
<H[123456]>                                                  (22)
```

以下正则表达式匹配字符串“Jack”或者“jack”。

```
[Jj]ack                                                      (23)
```

然而，正则表达式[0123456789]的书写非常不方便。因此，正则表达式引入了连接符“-”来定义字符的范围。以下正则表达式等价于正则表达式[0123456789]。

```
[0-9]                                                        (24)
```

以下正则表达式可以匹配任意小写字母。

```
[a-z]                                                        (25)
```

以下正则表达式可以匹配任意大写字母。

```
[A-Z]                                                        (26)
```

> **注意**：当且仅当在字符类中的连接符“-”不是第一个字符时，它才具有特殊的含义：它可以指定字符类的最大边界和最小边界之间的任何字符。它的具体含义由具体的字符类决定。因此，字符类的最大边界和最小边界，以及字符在 ASCII 或 Unicode 表中出现的顺序共同确定了连接符“-”指定的字符的范围。
>
> 如果要在字符类中包括连接符“-”，则必须将它作为第一个字符。如正则表达式[-a]匹配字符“-”或者“a”。

在字符类中，若字符“^”是字符类的第一个字符，则表示否定该字符类，即匹配除了该字符类之外的任意字符。以下正则表达式可以匹配任何非元音字符。

```
[^aAeEiIoOuU]                                                    (27)
```

以下正则表达式可以匹配除连接符“-”之外的任何字符。

```
[^-]                                                             (28)
```

以下正则表达式匹配字符 a 之后不是字符 b 的字符串。

```
a[^b]                                                            (29)
```

使用工具 Regex Tester 测试正则表达式 a[^b]的结果如图 1.13 所示。

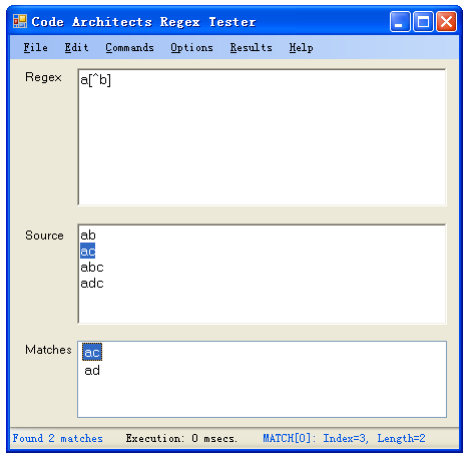

图 1.13　测试正则表达式 a[^b]

> **注意**：正则表达式中的元字符在字符类中不做任何特殊处理，它仅仅表示一个自身的字符。如正则表达式[-.]只能匹配字符“-”和“.”，它不能匹配除字符“-”和“.”之外的任何字符。因此，在字符类中使用元字符时，不需要使用转义运算。

在正则表达式中，常用的字符类如表 1.2 所示。

表 1.2　常用的字符类

字符或表达式	说明
.	匹配除换行符之外的任意字符
\w	匹配单词字符（包括字母、数字、下画线和汉字）
\W	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
\s	匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等
\S	匹配任意的非空白字符
\d	匹配任意的数字
\D	匹配任意的非数字字符
[aeiou]	匹配字符集合中的任何字符

续表

字符或表达式	说明
[^aeiou]	匹配除字符集合中的以外的字符
[0-9a-zA-Z_]	匹配任何数字、字母（大写字母和小写字母）和下画线，等同于\w
[^0-9a-zA-Z_]	匹配除任何数字、字母、下画线之外的任何字符，等同于\W
\p{name}	匹配{name}指定的命名字符类中的任何字符
\P{name}	匹配除{name}指定的命名字符类中之外的任何字符

> **注意：**在表 1.2 中，表达式\p{name}和\P{name}为.NET Framework 所支持。

1.4.2　字符转义

正则表达式定义了一些特殊的元字符，如^、$、.等。由于这些字符在正则表达式中被解释成其他的指定的意义，如果需要匹配这些字符，则需要使用字符转义来解决这一问题。转义字符为“\”（反斜杠），它可以取消这些字符（如^、$、.等）在表达式中具有的特殊意义。

以下正则表达式匹配字符“.”。

```
\.                                                    (30)
```

以下正则表达式匹配字符“*”。

```
\*                                                    (31)
```

以下正则表达式匹配字符“\”。

```
\\                                                    (32)
```

以下正则表达式匹配字符串“www.myweburl.com”。

```
www\.myweburl\.com                                    (33)
```

正则表达式的常用转义字符的说明如表 1.3 所示。其中，除.、$、^、{、[、(、|、)、*、+、?、\之外的字符不需要进行转义，它们都表示字符本身。

表 1.3　常用字符转义

字符或表达式	说明
\a	响铃（警报）\u0007
\b	在正则表达式中，表示单词的边界；如果在字符类中，则表示退格符\u0008
\t	制表符\u0009
\r	回车符\u000D
\v	垂直制表符\u000B
\f	换页符\u000C
\n	换行符\u000A
\e	回退（Esc）符\u001B
\040	将ASCII字符匹配为八进制数（最多三位）
\x20	使用十六进制表示，形式与ASCII字符匹配
\cC	ASCII控制字符，如Ctrl-C
\u0020	使用十六进制表示形式（恰好四位）与Unicode字符匹配

1.4.3　反义

在使用正则表达式时，如果需要匹配不在字符类指定范围内的字符时，可以使用反义规则。

以下正则表达式匹配字符 a 之后不是字符 b 的字符串。

```
a[^b]                                                 (34)
```

以下正则表达式匹配被尖括号括起来的、以字符串“asp”开头的、倒数第二个字符不能为字符“>”的、长度为 6 的任意字符串。

```
<asp[^>]>                                                    (35)
```

其实，在前面的小节中已经使用了反义的表达式，如\W、\S、\D、[^aeiou]等。常用的反义表达式如表 1.4 所示。

表 1.4　常用的反义表达式

字符或表达式	说明
\W	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
\S	匹配任意的非空白字符
\D	匹配任意的非数字字符
\B	匹配不是单词开头和结束的任何位置
[^a]	匹配除字符a之外的任意字符
[^aeiou]	匹配除字符集合（aeiou）中的字符之外的任意字符

1.4.4　限定符

正则表达式的元字符一次一般只能匹配一个位置或一个字符，如果想要匹配零个、一个或多个字符时，则需要使用限定符。限定符用于指定允许特定字符或字符集自身重复出现的次数。如{n}表示重复 *n* 次、{n,}表示重复至少 *n* 次、{n,m}表示重复至少 *n* 次，最多 *m* 次。常用限定符的说明如表 1.5 所示。

表 1.5　常用限定符

字符或表达式	说明
{n}	重复*n*次
{n,}	重复至少*n*次
{n,m}	重复至少*n*次，最多*m*次
*	重复至少0次，等同于{0,}
+	重复至少1次，等同于{1,}
?	重复0次或1次，等同于{0,1}
*?	尽可能少地使用重复的第一个匹配
+?	尽可能少地使用重复但至少使用一次
??	使用零次重复（如有可能）或一次重复
{n}?	等同于{n}
{n,}?	尽可能少地使用重复，但至少使用*n*次
{n,m}?	介于*n*次和*m*次之间、尽可能少地使用重复

以下正则表达式可以匹配字符串“color”或者“colour”。表达式 u?表示字母“u”可以出现 1 次或者不出现。

```
colou?r                                                      (36)
```

以下正则表达式可以匹配字符串“four”或者“for”。表达式 u?表示字母“u”可以出现 1 次或者不出现。

```
fou?r                                                        (37)
```

以下正则表达式匹配以字符串“name”开头的、以数字字符串结尾的字符串。其中，表达式\d+可以匹配长度至少为 1 的数字字符串。

```
\bname\d+\b                                                  (38)
```

以下正则表达式匹配被尖括号括起来的、以字符串“asp:TextBox ”（最后一个字符是空格）

开头的字符串。正则表达式<asp:TextBox [^>]+>中的字符类[^>]匹配除了尖括号“>”之外的任何字符。

```
<asp:TextBox [^>]+>                                                    (39)
```

以下正则表达式匹配以字母 a 开头的单词。\w 匹配一个单词字符，\w*表示该单词字符可以重复零次或多次。

```
\ba\w*\b                                                               (40)
```

正则表达式\ba\w*\b 匹配单词“anterior”的过程如图 1.14 所示，匹配的具体步骤如下。

1. 匹配单词的开始位置；
2. 在步骤①时匹配字符 a；
3. 匹配字符 a 之后，该表达式可以直接通过步骤②匹配到单词的结束位置，并完成整个匹配过程，从而匹配单词“a”；
4. 该表达式在匹配字符 a 之后，可以通过步骤③来匹配任意单词字符；
5. 重复步骤④一次或多次，即匹配一个或多个任意单词字符；
6. 匹配到单词的结束位置，并完成整个匹配过程，从而匹配到以字符 a 开头的长度大于 1 的单词。

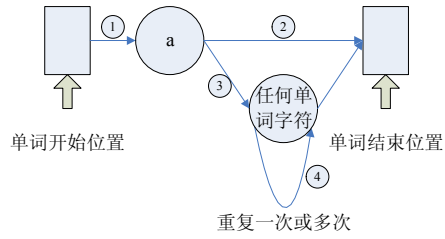

图 1.14　\ba\w*\b 匹配过程

限定符除了使用*、+和?之外，还可以使用数字直接指定重复的次数。{n}表示重复 *n* 次；{n,}表示至少重复 *n* 次，而最多重复的次数不限制；{n,m}（其中，*n*<=*m*）表示至少重复 *n* 次、最多重复 *m* 次。以下正则表达式匹配 3 位的整数，该正则表达式和正则表达式\d\d\d 等价。

```
\d{3}                                                                  (41)
```

以下正则表达式匹配 3 位以上（包括 3 位）的整数，该正则表达式和正则表达式\d\d\d\d*或\d\d\d+等价。

```
\d{3,}                                                                 (42)
```

以下正则表达式匹配 3 位或 4 位的整数，该正则表达式和正则表达式\d\d\d\d?等价。

```
\d{3,4}                                                                (43)
```

以下正则表达式匹配当前国内以“13”开头的手机号码：即以字符串“13”开头的、后接连续 9 个数字的字符串。

```
13\d{9}                                                                (44)
```

以下正则表达式匹配当前国内部分地区的固定电话号码。其中，号码的前 3 位为区号，后 8 位为本地号码，区号和本地号码使用连接符号“-”进行连接。

```
0\d{2}-\d{8}                                                           (45)
```

以下正则表达式匹配当前国内部分地区的固定电话号码。其中，号码的前 4 位为区号，后 7 位为本地号码，区号和本地号码使用连接符号“-”进行连接。

```
0\d{3}-\d{7}                                                           (46)
```

以下正则表达式能够匹配满足以下特征的单词。

- 字符串只包含字符 a 和 b，且字符 b 必须在字符 a 之后；
- 字符 a 连续的个数最多为 4，最小为 1；
- 字符 b 连续的个数最多为 3，最小为 1。

```
\ba{1,4}b{1,3}\b                                                    (47)
```

正则表达式\ba{1,4}b{1,3}\b 能够匹配的单词如下：

ab、aab、aaab、aaaab、abb、abbb、aabb、aabbb、aaabb、aaabbb、aaaabb、aaaabbb。

使用工具 Regex Tester 测试正则表达式\ba{1,4}b{1,3}\b 的结果如图 1.15 所示。

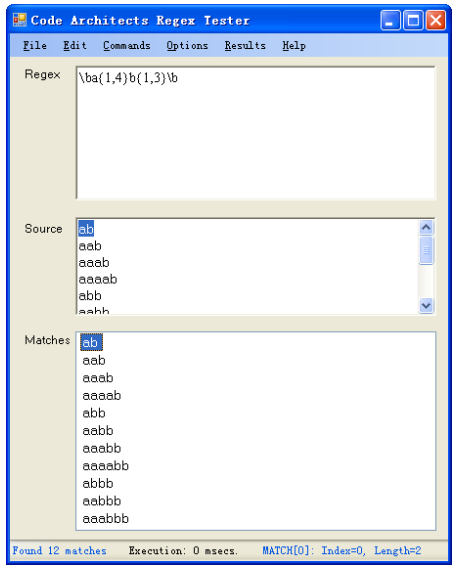

图 1.15 测试正则表达式\ba{1,4}b{1,3}\b

如果在限定符*、+、?、{n}、{n,}和{n,m}之后再添加一个字符“?”，则表示尽可能少地重复字符“?”之前的限定符号的重复次数，这种方式匹配被称为懒惰匹配。与之相对应的是贪婪匹配，即仅仅使用单个限定符*、+、?、{n}、{n,}和{n,m}的匹配。常用的懒惰限定符如表 1.6 所示。

表 1.6 常用的懒惰限定符

字符或表达式	说明
*?	尽可能少地使用重复的第一个匹配
+?	尽可能少地使用重复，但至少使用一次
??	使用零次重复（如有可能）或一次重复
{n}?	等同于{n}
{n,}?	尽可能少地使用重复但至少使用*n*次
{n,m}?	介于*n*次和*m*次之间、尽可能少地使用重复

以下正则表达式匹配以字母 a 开头、以字母 b 结束的最长字符串。此时，这是一种贪婪匹配。

```
a.*b                                                                (48)
```

以下正则表达式匹配以字母 a 开头、以字母 b 结束的最短字符串。此时，这是一种懒惰匹配。

```
a.*?b                                                               (49)
```

如果将正则表达式 a.*b 应用于字符串“aabab”，则匹配字符串“aabab”。如果将正则表达式 a.*?b 应用于字符串“aabab”，则匹配字符串“aab”和字符串“ab”，而不会匹配字符串“aabab”。

使用工具 Regex Tester 分别测试正则表达式 a.*b 和 a.*?b，结果分别如图 1.16 和图 1.17 所示。

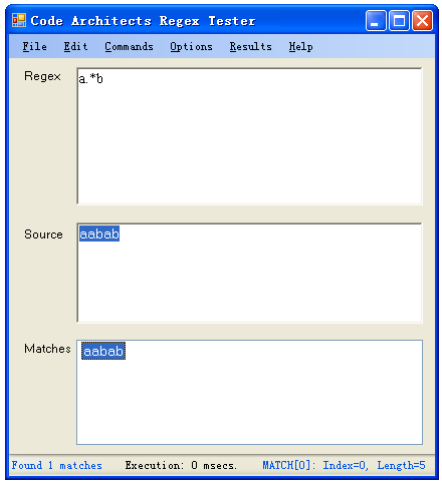

图 1.16　测试正则表达式 a.*b

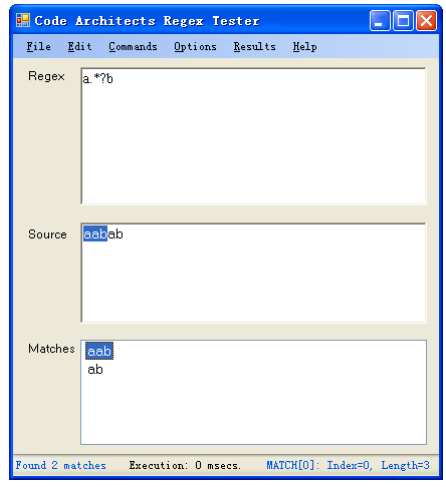

图 1.17　测试正则表达式 a.*?b

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度至少为 3 的字符串。此时，这是一种贪婪匹配。

```
a.+b                                                    (50)
```

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度至少为 3 的字符串。此时，这是一种懒惰匹配。

```
a.+?b                                                   (51)
```

正则表达式 a.+?b 在匹配过程中，字母 a 和字母 b 之间的字符串实际上只重复了 1 次。如果将正则表达式 a.+b 应用于字符串“avbab”，则匹配字符串“avbab”。如果将正则表达式 a.+?b 应用于字符串“avbab”，则匹配字符串“avb”，而不会匹配字符串“avbab”。使用工具 Regex Tester 分别测试正则表达式 a.+b 和 a.+?b，结果分别如图 1.18 和图 1.19 所示。

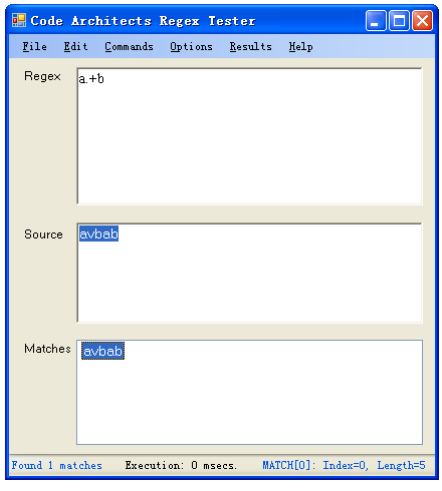

图 1.18　测试正则表达式 a.+b

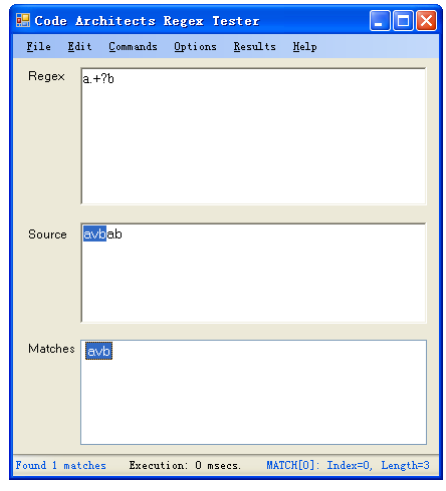

图 1.19　测试正则表达式 a.+?b

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度为 2 或者 3 的字符串。此时，这是一种贪婪匹配。

```
a.?b                                                    (52)
```

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度为 3 的字符串。此时，这是一种懒

惰匹配。

```
a.??b                                                    (53)
```

正则表达式 a.??b 在匹配过程中，字母 a 和字母 b 之间的字符要么出现，要么最多出现 1 次。使用工具 Regex Tester 分别测试正则表达式 a.?b 和 a.??b，结果分别如图 1.20 和图 1.21 所示。

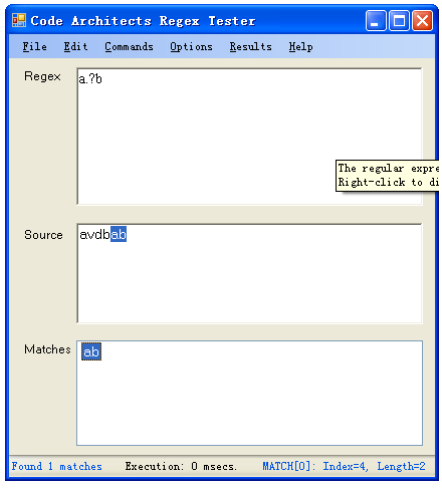

图 1.20　测试正则表达式 a.?b

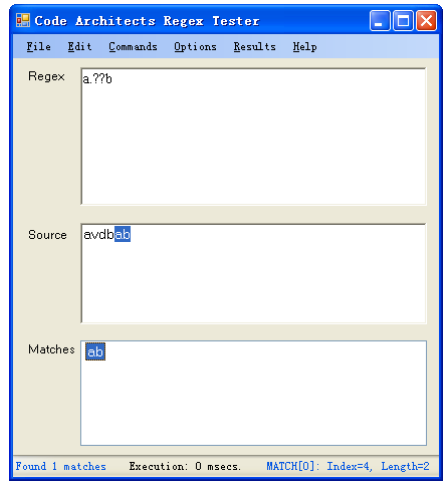

图 1.21　测试正则表达式 a.??b

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度至少为 3 的字符串。此时，这是一种贪婪匹配。

```
a.{1,}b                                                  (54)
```

以下正则表达式匹配以字母 a 开头、以字母 b 结束、长度至少为 3 的字符串。此时，这是一种懒惰匹配。

```
a.{1,}?b                                                 (55)
```

正则表达式 a.{1,}?b 在匹配过程中，字母 a 和字母 b 之间的字符串实际上只重复了 1 次。如果将正则表达式 a.{1,}b 应用于字符串“avbab”，则匹配字符串“avbab”。如果将正则表达式 a.{1,}?b 应用于字符串“avbab”，则匹配字符串“avb”，而不会匹配字符串“avbab”。使用工具 Regex Tester 分别测试正则表达式 a.{1,}b 和 a.{1,}?b，结果分别如图 1.22 和图 1.23 所示。

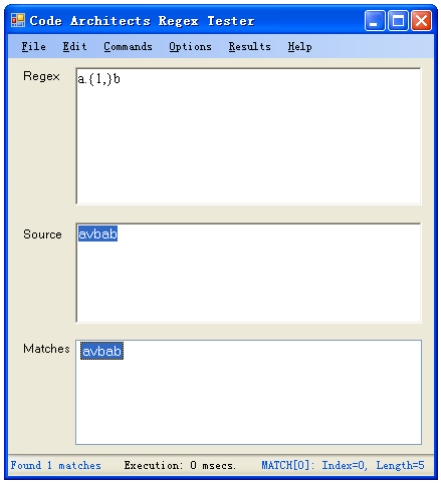

图 1.22　测试正则表达式 a.{1,}b

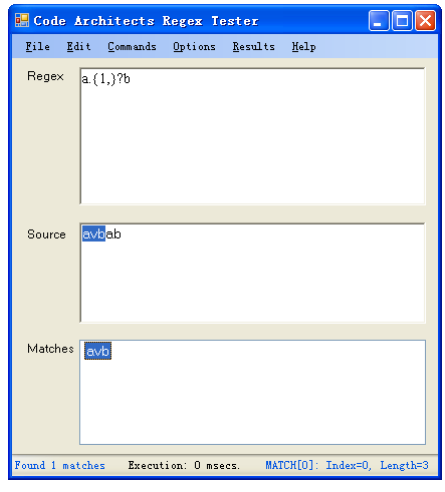

图 1.23　测试正则表达式 a.{1,}?b

1.5　字符的运算

1.5.1　替换

正则表达式 0\d{2}-\d{8}和 0\d{3}-\d{7}分别匹配区号为 3 位和 4 位的固定电话号码，如果需要同时匹配区号为 3 位和 4 位的固定电话号码时，使用替换可以满足这一需求。最简单的替换是使用字符“|”表示，它表示如果某一个字符串匹配了正则表达式中的字符“|”的左边或者右边的规则，那么该字符串也匹配了该正则表达式。

以下正则表达式匹配了当前国内部分地区的两种固定电话号码：一种是号码的前 4 位为区号，后 7 位为本地号码；另一种是号码的前 3 位为区号，后 8 位为本地号码。其中，区号和本地号码都使用连字符“-”进行连接。

```
0\d{2}-\d{8}|0\d{3}-\d{7}                                    (56)
```

以下正则表达式匹配了当前国内部分地区的三种固定电话号码：一种是号码的前 3 位为区号，后 8 位为本地号码；另一种是号码的前 4 位为区号，后 7 位为本地号码；最后一种是号码的前 4 位为区号，后 8 位为本地号码。其中，区号和本地号码都使用连字符“-”进行连接。

```
0\d{2}-\d{8}|0\d{3}-\d{7}|0\d{3}-\d{8}                       (57)
```

以下正则表达式匹配当前国内部分地区的区号为 4 位的固定电话号码。其中，区号和本地号码可以使用连字符“-”进行连接，也可以不使用连接符号“-”。

```
0\d{3}-\d{7}|0\d{3}[-]?\d{7}                                 (58)
```

使用工具 Regex Tester 测试正则表达式 0\d{3}-\d{7}|0\d{3}[-]?\d{7}，测试结果如图 1.24 所示。其中，电话号码 0731-1234567 和 07311234567 均被匹配。

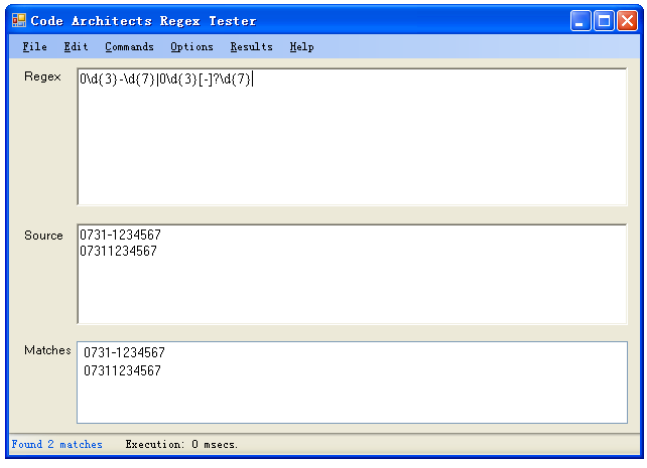

图 1.24　测试 0\d{3}-\d{7}|0\d{3}[-]?\d{7}匹配电话号码

正则表达式[Jj]ack 可以匹配字符串“Jack”或者“jack”。该正则表达式还可以使用替换来实现同样的匹配效果。以下正则表达式等效于正则表达式[Jj]ack。

```
Jack|jack                                                    (59)
```

以下正则表达式等效于正则表达式 Jack|jack。因此，正则表达式[Jj]ack、Jack|jack 和 Jack|jack 能够匹配的所有字符串都是相同的。

```
(J|j)ack                                                     (60)
```

使用工具 Regex Tester 分别测试正则表达式[Jj]ack 和(J|j)ack，结果分别如图 1.25 和图 1.26 所示。从图中可以看到，两个正则表达式匹配的结果是相同的。

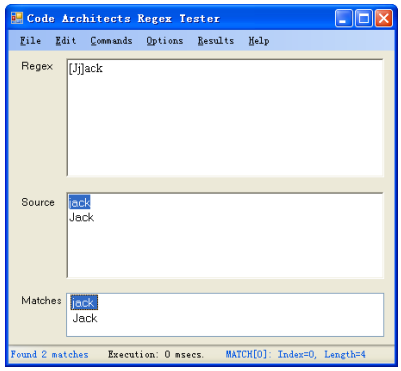

图 1.25 测试正则表达式[Jj]ack

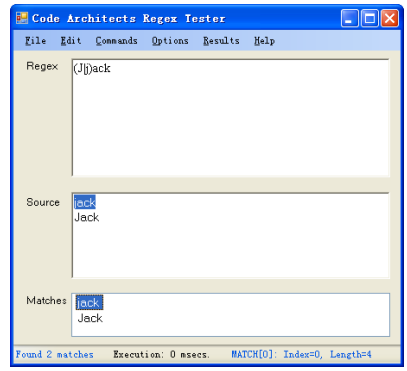

图 1.26 测试正则表达式(J|j)ack

正则表达式的常用替换说明如表 1.7 所示。

表 1.7 正则表达式的常用替换

字符或表达式	说明
|	匹配|（竖线）字符的左侧或右侧
(?(表达式)yes|no)	表达式要么与“yes”部分匹配；要么与“no”部分匹配。其中，“no”部分可省略
(?(name)yes|no)	以name命名的字符串要么与“yes”部分匹配，要么与“no”部分匹配。其中，“no”部分可省略

> **注意：**字符|在匹配表达式时，首先匹配|字符的左侧部分，当左侧部分不匹配时，它才尝试匹配|字符的右侧部分。

有以下两个正则表达式：

```
\d{5}-\d{3}|\d{5}                    (61)
\d{5}|\d{5}-\d{3}                    (62)
```

根据字符|的匹配原则（优先匹配左侧表达式），正则表达式\d{5}|\d{5}-\d{3}只能匹配 5 位的数字字符串，而不会匹配用连接符号连接的 8 位数字字符串。然而，正则表达式\d{5}-\d{3}|\d{5}能够匹配用连接符号连接的 8 位数字字符串或者 5 位的数字字符串。因为，该表达式首先尝试匹配用连接符号连接的 8 位数字字符串，只有当未匹配时，才匹配 5 位的数字字符串。

使用工具 Regex Tester 分别测试了正则表达式\d{5}|\d{5}-\d{3}和\d{5}-\d{3}|\d{5}，测试结果分别如图 1.27 和图 1.28 所示。正则表达式\d{5}|\d{5}-\d{3}只匹配了字符串“12345”。而正则表达式\d{5}-\d{3}|\d{5}可以匹配字符串“12345-678”和字符串“12345”。

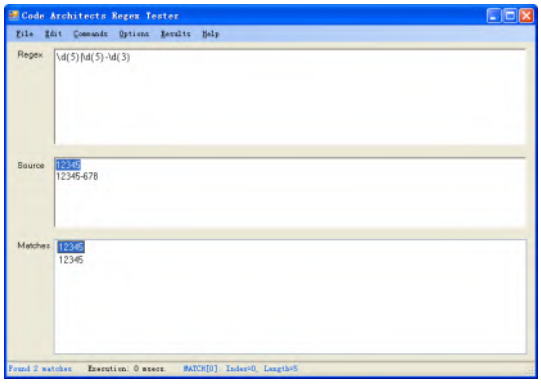

图 1.27 测试正则表达式\d{5}|\d{5}-\d{3}

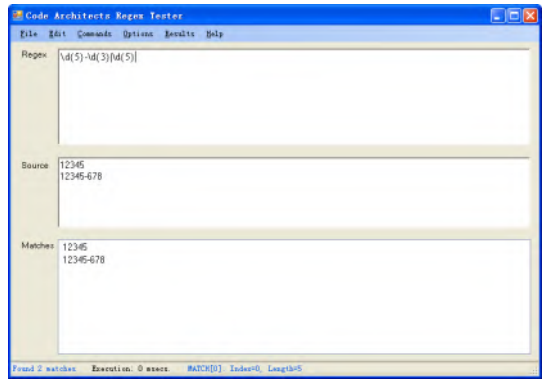

图 1.28 测试正则表达式\d{5}-\d{3}|\d{5}

1.5.2　分组

分组又称为子表达式，即把一个正则表达式的全部或部分分成一个或多个组。其中，分组使用的字符为“(”和“)”，即左圆括号和右圆括号。分组之后，可以将在“(”和“)”之间的表达式看成一个整体来处理。以下正则表达式可以匹配重复出现字符串“abc”一次或两次的字符串。此时，表达式将“abc”看成一个整体来进行重复匹配。

```
(abc){1,2}                                                    (63)
```

以下正则表达式可以匹配简单的 IP 地址。

```
(\d{1,3}\.){3}\d{1,3}                                         (64)
```

正则表达式解释如下。

- 表达式\d{1,3}\.先匹配 1~3 位的整数，然后匹配一个字符“.”（点号），如“1.”、“12.”、“123.”、“888.”等。
- 表达式(\d{1,3}\.){3}将子表达式\d{1,3}\.匹配的字符串重复 3 次，如“1.2.3.”、“12.34.56.”、“123.456.789.”、“888.899.569.”等。
- 表达式\d{1,3}将匹配 1~3 位的整数。

综合以上分析，正则表达式(\d{1,3}\.){3}\d{1,3}能够匹配简单的 IP 地址，如“10.0.0.1”、“123.123.235.235”等。使用工具 Regex Tester 测试正则表达式(\d{1,3}\.){3}\d{1,3}，结果如图 1.29 所示。

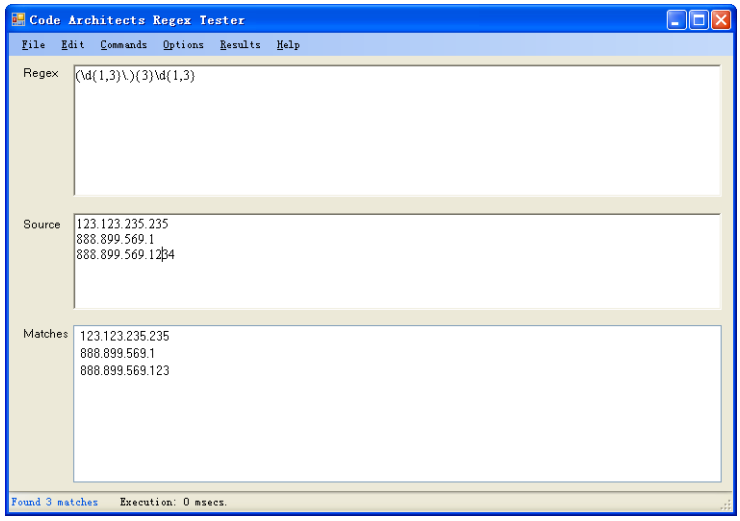

图 1.29　测试正则表达式(\d{1,3}\.){3}\d{1,3}

1.5.3　反向引用

当一个正则表达式被分组之后，每一个组将自动被赋予一个组号，该组号可以代表该组的表达式。其中，组号的编制规则为：从左到右、以分组的左括号“(”为标志，第一个分组的组号为 1，第二个分组的组号为 2，依此类推。

反向引用提供了查找重复字符组的简便方法。它们可以被认为是再次匹配同一个字符串的快捷指令。反向引用可以使用数字命名（即默认名称）的组号，也可以使用指定命名的组号。具体说明如表 1.8 所示。

表 1.8 反向引用

表达式	说明
\数字	使用数字命名的反向引用
\k<name>	使用指定命名的反向引用

> **注意**：在表 1.8 中，表达式\k<name>为.NET Framework 所支持。

以下正则表达式匹配具有两个重复字符的单词。

```
\b(\w)\1\b                                                    (65)
```

以下正则表达式首先匹配单词的开头处，然后匹配一个字符和数字，再重复该字符和数字，最后是单词的结尾处。

```
\b(\w)(\d)\1\2\b                                              (66)
```

> **注意**：正则表达式\b(\w)\1\b 和\b(\w)\w\b 并不等效，第一个表达式只匹配含有两个相同字符的单词，而第二个表达式则匹配具有两个字符（可以相同，也可以不相同）的单词。

使用工具 Regex Tester 分别测试正则表达式\b(\w)\w\b 和\b(\w)\1\b，结果分别如图 1.30 和图 1.31 所示。在测试结果中，正则表达式\b(\w)\1\b 只匹配单词“aa”，而正则表达式\b(\w)\w\b 可以匹配单词“aa”和“ab”。

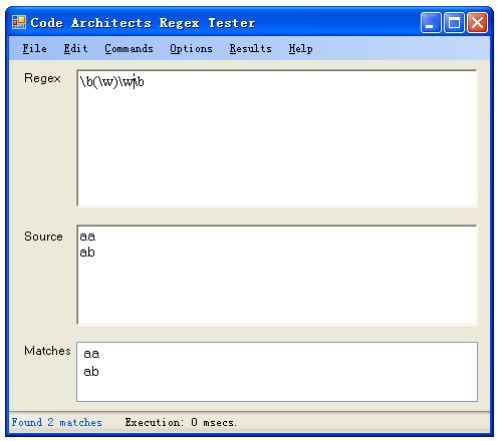

图 1.30 测试正则表达式\b(\w)\1\b

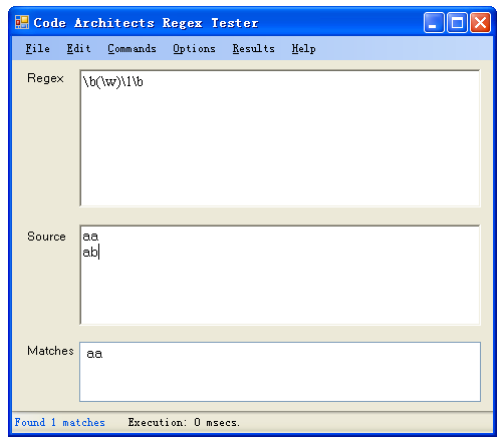

图 1.31 测试正则表达式\b(\w)\w\b

以下正则表达式匹配以两个重复字符结尾的单词。

```
\b\w*(\w+)\1\b                                                (67)
```

以下正则表达式匹配重复出现的单词。

```
\b(\w+)\b\s+\1\b                                              (68)
```

上述正则表达式\b(\w+)\b\s+\1\b 匹配的具体过程如下。

1. 表达式\b(\w+)\b 匹配一个单词，且单词的长度至少为 1。
2. 表达式\s+匹配一个或多个空白字符。
3. 表达式\1 将重复子表达（\w+）匹配的内容，即重复匹配的单词。
4. 匹配单词的结束位置。

分组不但可以使用数字作为组号，而且还可以使用自定义名称作为组号。以下两个正则表达式都是将分组后的子表达式\w+命名为“word”。

```
(?<word>\w+)                                                    (69)
(?'word'\w+)                                                    (70)
```

因此，正则表达式\b(\w+)\b\s+\1\b 和以下正则表达式等价，它们都匹配重复出现的单词。

```
\b(?<word>\w+)\b\s+\k<word>\b                                   (71)
```

以下正则表达式和正则表达式\b\w*(\w+)\1\b 等价，也是匹配以两个重复字符结尾的单词。

```
\b\w*(?<char>\w+)\k<char>\b                                     (72)
```

分组子表达式（?<name>）将元字符括在其中，并强制正则表达式引擎记住该子表达式匹配，同时使用“name”将该匹配进行命名。反向引用\k<name>使引擎对当前字符和以名称“name”存储的先前匹配字符进行比较，从而匹配具有重复字符的字符串。正则表达式中的常用分组说明如表 1.9 所示。

表 1.9　常用分组说明

字符	说明
(experssion)	匹配字符串experssion，并将匹配的文本保存到自动命名的组里
(?<name>experssion)	匹配字符串experssion，并将匹配的文本以name进行命名。该名称不能包含标点符号，不能以数字开头
(?:experssion)	匹配字符串experssion，不保存匹配的文本，也不给此组分配组号
(?=experssion)	匹配字符串experssion前面的位置
(?!experssion)	匹配后面不是字符串experssion的位置
(?<=experssion)	匹配字符串experssion后面的位置
(?<!experssion)	匹配前面不是字符串experssion的位置
(?>experssion)	只匹配字符串experssion一次

1.6　正则的其他运算

1.6.1　零宽度断言

在前面的小节中，元字符\b、^和$都匹配一个位置，且这个位置满足一定的条件。这里，把满足的这一个条件称为断言或零宽度断言。正则表达式中的常用零宽度断言如表 1.10 所示。

表 1.10　零宽度断言

字符（断言）	说明
^	匹配行的开始位置
$	匹配行的结束位置
\A	匹配必须出现在字符串的开头
\Z	匹配必须出现在字符串的结尾或字符串结尾处的\换行符号n之前
\z	匹配必须出现在字符串的结尾
\G	匹配必须出现在上一个匹配结束的地方
\b	匹配字符的开始或结束位置
\B	匹配不是在字符的开始或结束位置

在表 1.9 中，表达式(?=experssion)、(?!experssion)、(?<=experssion)和(?<!experssion)都是匹配一个位置。下面将详细介绍表达式(?=experssion)和(?<=experssion)。

(?=experssion)又称为零宽度正预测先行断言，它断言自身位置的后面能够匹配表达式experssion。以下正则表达式匹配以字符串“ed”结尾的单词的前面部分，即匹配单词的除字符串“ed”之外的部分。

```
\b\w+(?=ed\b)                                                    (73)
```

(?<=experssion)又称为零宽度正回顾后发断言，它断言自身位置的前面能够匹配表达式experssion。以下正则表达式匹配以字符串“an”开头的单词的后面部分，即匹配单词的除字符串“an”之外的部分。

```
(?<=\ban)\w+\b                                                   (74)
```

使用工具 Regex Tester 分别测试正则表达式\b\w+(?=ed\b)和(?<=\ban)\w+\b，结果分别如图 1.32 和图 1.33 所示。

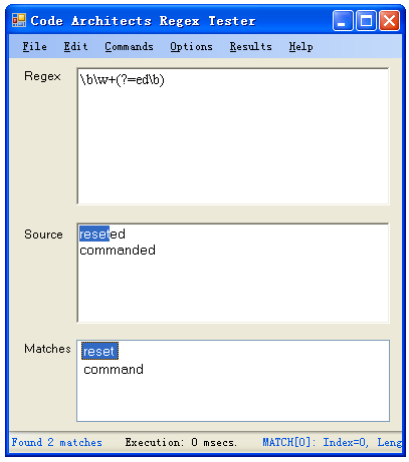

图 1.32 测试正则表达式\b\w+(?=ed\b)

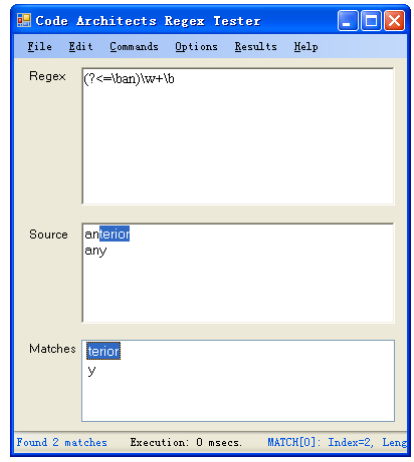

图 1.33 测试正则表达式(?<=\ban)\w+\b

1.6.2 负向零宽度断言

零宽度断言只能指定或匹配一个位置。而负向零宽度断言与零宽度断言恰恰相反，它能够指定或匹配不止一个位置，即所说的“反义”。特别是在匹配字符串中不包含指定的字符时，负向零宽度断言特别有用。以下正则表达式中的表达式 a(?!b)将断言字符“a”之后不能为字符“b”。

```
\b\w*a(?!b)\w*\b                                                 (75)
```

因此，正则表达式\b\w*a(?!b)\w*\b 匹配单词字符串，且该字符串中的字符“a”之后不能为字符“b”。

表达式(?!experssion)又称为负向零宽度断言或者零宽度负预测先行断言，它断言自身位置的后面不能匹配字符串 experssion。以下正则表达式首先匹配长度为 3 的单词字符串，该字符串之后不能是数字字符串。

```
\b\w{3}(?!\d+)                                                   (76)
```

使用工具 Regex Tester 测试正则表达式\b\w{3}(?!\d+)，结果如图 1.34 所示。

表达式(?<!experssion)又称为零宽度负回顾后发断言，它断言自身位置的前面不能匹配字符串experssion。

以下正则表达式匹配不以数字开头的字符串，该字符串的开头只能包括大写字母、小写字母或下画线。

```
(?<!\d+)[a-z_A-Z]+                                               (77)
```

使用工具 Regex Tester 测试正则表达式(?<!\d+)[a-z_A-Z]+，结果如图 1.35 所示。

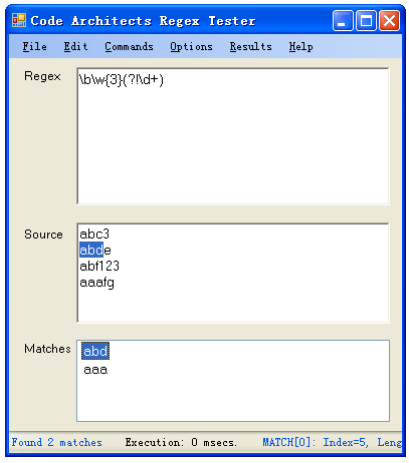

图 1.34　测试正则表达式\b\w{3}(?!\d+)

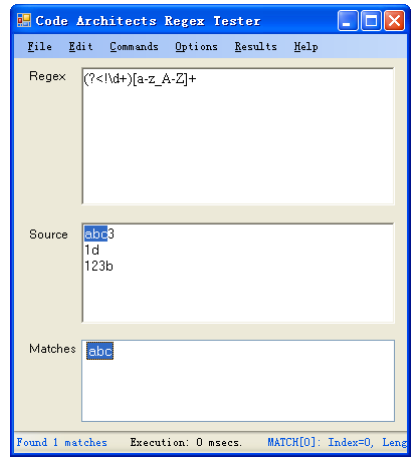

图 1.35　测试正则表达式(?<!\d+)[a-z_A-Z]+

1.6.3　匹配选项

匹配选项可以指定正则表达式匹配中的行为，如忽略大小写、处理多行、处理单行、从右到左开始执行匹配等。

> **注意：**本节中下面介绍的匹配选项为.NET Framework 所支持。

.NET Framework 正则表达式中的常用匹配选项说明如表 1.11 所示。

表 1.11　常用匹配选项

选项	内联字符	说明
None	N/A	指定不设置任何选项
IgnoreCase	i	指定不区分大小写
Multiline	m	指定多行模式，即修改^和$的含义，以使它们分别与任何行的开头和结尾匹配
ExplicitCapture	n	指定必须指定分组的名称或组号
Compiled	N/A	指定正则表达式将被编译为程序集
Singleline	s	指定单行模式
IgnorePatternWhitespace	x	指定消除表达式中空白字符，并启用字符(#)后面的注释
RightToLeft	N/A	指定匹配是从右向左而不是从左向右进行的
ECMAScript	N/A	指定已为表达式启用了符合ECMAScript的行为
CultureInvariant	N/A	指定忽略语言中的区域性差异

1.6.4　注释

正则表达式中除了表达式的基本内容外，还可以包括注释。其中，注释一般通过表达式（?#注释）实现。以下正则表达式在第一个分组中添加了注释“不能以数字开头”。

```
(?<!\d+(?#不能以数字开头))[a-z_A-Z]+                    (78)
```

> **注意：**如果要在正则表达式中包含注释，则最好打开 IgnorePatternWhitespace 选项，即忽略模式里的空白字符。因此，此时可以在注释中添加空格、换行符号、制表符号等。一旦启用了该选项，则符号#之后的内容全部被忽略。

正则表达式(?<!\d+)[a-z_A-Z]+可以写成以下形式。

```
(?<!              # 断言是否能匹配该组中的表达式                                    (79)
\d+               # 长度至少为 1 的数字字符串
)                 # 表达式结束
[a-z_A-Z]+        # 只能包括大写字母、小写字母或下画线的长度至少为 1 的字符串
```

1.6.5 优先级顺序

正则表达式存在元字符、转义符、限定符、|等运算或表达式。在匹配过程中，正则表达式都事先规定了这些运算或表达式的优先级。正则表达式也可以像数学表达式一样来求值。也就是说，正则表达式可以从左至右、并按照一个给定的优先级来求值。表 1.12 是按照优先级从高到低的顺序列出的正则表达式运算符的优先级顺序表。

表 1.12 优先级顺序表

运算符或表达式	说明
\	转义符
()、(?:)、(?=)、[]	圆括号和方括号
*、+、?、{n}、{n,}、{n,m}	限定符
^、$、\（元字符）	位置和顺序
|	“或”运算

1.6.6 递归匹配

递归匹配在匹配具有嵌套结构的字符串时特别有效。给定算术表达式((1+2)*(3+4))，该表达式具有嵌套结构。如果需要使用正则表达式检查该表达式的结构是否正确，则使用递归匹配能够解决该问题。

> **注意：**本节下面介绍的递归匹配为.NET Framework 所支持。

在.NET Framework 中，正则表达式用于递归匹配的表达式说明如表 1.13 所示。

表 1.13 用于递归匹配的表达式说明

表达式	说明
(?<name>expression)	把匹配的内容命名为name，并压入堆栈
(?<-name>expression')	从堆栈中弹出最后压入的命名为name的匹配内容。如果堆栈为空，则当前组匹配失败
(?(name)yes|no)	如果堆栈上存在命名为name的匹配内容，则继续匹配yes部分的表达式，否则继续匹配no部分的表达式
(?!)	零宽负向先行断言。由于没有后缀表达式，因此匹配总是失败

以下正则表达式能够匹配算术表达式((1+2)*(3+4))。

```
\([^()]*(((?<bracket>\()[^()]*)+((?<-bracket>\))[^()]*)+)*(?(bracket)(?!))\)
                                                                    (80)
```

下面是对该正则表达式进行的详细分析。

```
\(                          # 匹配最外层的左括号
    [^()]*                  # 匹配最外层左括号后面的、不是括号“()”的内容
    (
        (
            (?<bracket>\()  # 如果匹配到左括号，则命名为 bracket，并压入堆栈
            [^()]*          # 匹配当前左括号后面的、不是括号“()”的内容
        )+
```

```
        (
            (?<-bracket>\()    # 如果匹配到右括号，则弹出命名为 bracket 的内容
            [^()]*             # 匹配右括号后面的，不是括号“()”的内容
        )+
    )*
    (?(bracket)(?!))       #如果匹配到最外层的右括号前面，则检查堆栈内容是否为空。如果不为空，
                           #则匹配失败
\)                         #匹配最外层的右括号
```

使用工具 Regex Tester 测试正则表达式\([^()]*(((?<bracket>\()[^()]*)+((?<-bracket>\))[^()]*)+)*(?(bracket)(?!))\)，结果如图 1.36 所示。

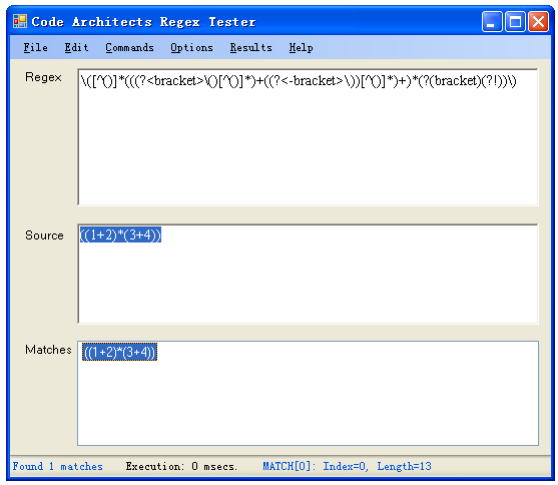

图 1.36　测试正则表达式\([^()]*(((?<bracket>\()[^()]*)+((?<-bracket>\))[^()]*)+)*(?(bracket)(?!))\)

1.7　典型正则表达式解释

本节解释基于正则表达式基础理论的典型正则表达式。如匹配 Windows 运算系统的名称、匹配 HTML 标记、匹配 HTML 标记之间的内容。

1.7.1　匹配 Windows 运算系统的名称

Windows 运算系统存在很多版本，如 Windows 95、Windows 98、Windows 2000、Windows ME、Windows XP、Windows 2003、Windows 7、Windows 8 等。以下正则表达式能够精确匹配 Windows 运算系统的名称。

```
Windows\s*((95)|(98)|(2000)|(2003)|(ME)|(XP)|(7)|(8))                    (81)
```

上述表达式能够精确匹配 Windows 95、Windows 98、Windows 2000、Windows ME、Windows XP、Windows 2003、Windows 7、Windows 8 等运算系统的名称。然而，精确匹配 Windows 运算系统名称的正则表达式比较冗长，以下正则表达式能够简单匹配 Windows 运算系统的名称。

```
Windows\s\w+                                                              (82)
```

1.7.2　匹配 HTML 标记

HTML 标记一般被尖括号包围，如<a>、<table>、
、<input>等。以下正则表达式能够匹配 HTML 标记。

```
<[a-zA-Z][^>]*>                                                    (83)
```

该正则表达式解释如下。

- <匹配 HTML 标记的左尖括号。
- 字符类[a-zA-Z]可以匹配一个英文字母，它匹配 HTML 标记中除去左尖括号的第一个字符。
- 字符类[^>]可以匹配除右尖括号之外的任何字符。
- [^>]*可以匹配空字符串，或者由除右尖括号之外的任何字符组成的字符串。
- >匹配 HTML 标记的右尖括号。
- [a-zA-Z][^>]*匹配 HTML 标记的名称。

使用工具 Regex Tester 测试正则表达式<[a-zA-Z][^>]*>，结果如图 1.37 所示。

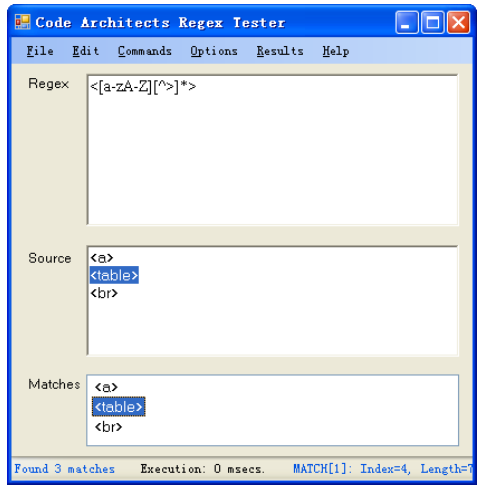

图 1.37 测试正则表达式<[a-zA-Z][^>]*>

1.7.3 匹配 HTML 标记之间的内容

如果要匹配 HTML 标记之间的内容，则可以使用以下代码：

```
(?<=<(?<tag>\w+)>).*(?=<\/\k<tag>>)                                (84)
```

该正则表达式解释如下。

- (?<tag>\w+)是一个分组，它的名称为“tag”。在该正则表达式匹配过程中，它将保存被匹配的内容。
- <(?<tag>\w+)>匹配 HTML 标记的开头标记，如<a>、<table>等。
- (?<=<(?<tag>\w+)>)是一个零宽度正回顾后发断言，它断言自身位置的前面能够匹配<(?<tag>\w+)>所匹配的内容。在此，该表达式断言被匹配的字符串的开头部分是 HTML 标记的开头标记。
- \k<tag>反向引用名称为“tag”的分组，即被匹配到的 HTML 标记的名称。
- \/匹配字符/。
- <\/\k<tag>>匹配 HTML 标记的结尾标记，如、</table>等。
- (?=<\/\k<tag>>)是一个零宽度正预测先行断言。在此，该表达式断言被匹配的字符串的结尾部分是 HTML 标记的结尾标记。
- .*匹配 HTML 标记之间的任何字符。

使用工具 Regex Tester 测试正则表达式(?<=<(?<tag>\w+)>).*(?=<\/\k<tag>>)，结果如图 1.38 所示。

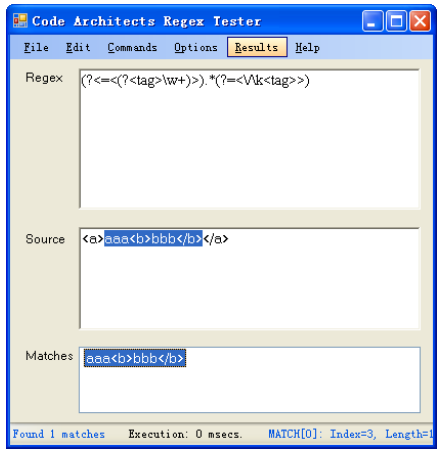

图 1.38　测试正则表达式(?<=<(?<tag>\w+)>).*(?=<\/\k<tag>>)

1.7.4　匹配 CSV 文件内容

CSV 文件是非常特殊的一种文件，它的内容满足以下 4 个条件。

- 数据被逗号（,）分割。在此，设被分割后的数据称为单个数据。
- 单个数据的长度不能为 0。
- 如果某单个数据中包含逗号（,），那么该单个内容被双引号（"）包围。
- 如果单个数据中包含双引号（"），则用两个双引号表示一个双引号。

以下内容是某个 CSV 文件的一部分。

```
aaa,bbb,ccc,ddd,"ab,de",eee,fff,ggg,"this is ""aa",hhhf
```

以下正则表达式能够匹配不包含逗号（,）或者双引号（"）的单个数据。

```
[^",]+                                                    (85)
```

如果单个数据包含逗号（,）或者双引号（"）时，那么该单个数据被双引号（"）包围。因此，匹配该类型单个数据的正则表达式形式如下：

```
"[匹配单个数据内容的正则表达式]"                          (86)
```

该类型单个数据要么是两个连续双引号（""），要么是除单个双引号（""）外的任意字符。因此，以下正则表达式能够匹配包含逗号（,）或者双引号（"）的单个数据。

```
"(?:([^"]|""))+"                                          (87)
```

综上所述，以下正则表达式能够匹配 CSV 文件中的单个数据。

```
[^",]+|("(?:([^"]|""))+")                                 (88)
```

以下正则表达式能够匹配 CSV 文件的内容。

```
([^",]+|("(?:([^"]|""))+"))(\s*,\s*([^",]+|("(?:([^"]|""))+")))*    (89)
```

该正则表达式解释如下。

- [^",]+|("(?:([^"]|""))+")匹配 CSV 文件中的单个数据。
- \s*匹配逗号（,）两边的空白字符。
- \s*,\s*([^",]+|("(?:([^"]|""))+"))匹配“逗号（,）＋单个数据”组成的字符串，并且还包含逗号（,）两边的空白字符。

- (\s*,\s*([^",]+|("(?:([^"]|"")+")))*匹配 0 个或多个“逗号（,）＋单个数据”组成的字符串。同样，也包含逗号（,）两边的空白字符。

使用工具 Regex Tester 测试正则表达式([^",]+|("(?:([^"]|"")+"))(\s*,\s*([^",]+|("(?:([^"]|"")+")))*，结果如图 1.39 所示。

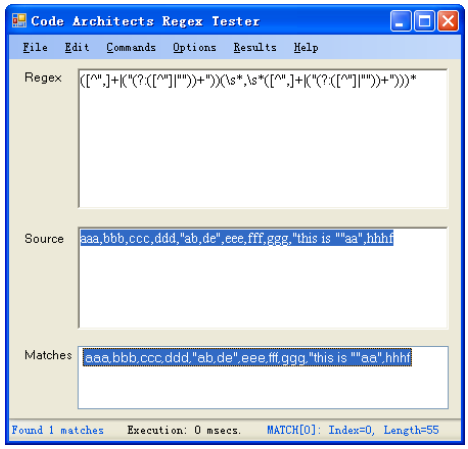

图 1.39 测试正则表达式([^",]+|("(?:([^"]|"")+"))(\s*,\s*([^",]+|("(?:([^"]|"")+")))*