Java正则表达式教程

正则表达式善于处理文本，对匹配、搜索和替换等操作都有意想不到的作用。正因如此，正则表达式现在是作为程序员七种基本技能之一*，因此学习和使用它在工作中都能达到很高的效率。

正则表达式应用于程序设计语言中，首次是出现在 Perl 语言，这也让 Perl 奠定了正则表达式旗手的地位。现在，它已经深入到了所有的程序设计语言中，在程序设计语言中，正则表达式可以说是标准配置了。

Java 中从 JDK 1.4 开始增加了对正则表达式的支持，至此正则表达式成为了 Java 中的基本类库，使用时不需要再导入第三方的类库了。Java 正则表达式的语法来源于象征着正则表达式标准的 Perl 语言，但也不是完全相同的，具体的可以参看 Pattern 类的 API 文档说明。

我在一次偶然中发现了位于 java.sun.com 站点上的 Java Tutorial，也在那里看到了关于 Java 的正则表达式教程，感觉它不同于其他的正则表达式教程，文中以大量的匹配实例来进行说明。为了能让 Java 学习者能更好地使用正则表达式，就将其完整地译出了。该教程中所介绍的正则表达式应用仅仅是最为简单的（并没有完全地涉及到 Pattern 类支持的所有正则表达式语法，也没有涉及到高级的应用），适合于从未接触过或者是尚未完全明白正则表达式基础的学习者。在学习完该教程后，应该对正则表达式有了初步的了解，并能熟练地运用 java.util.regex 包中的关于正则表达式的类库，为今后学习更高级的正则表达式技术奠定良好的基础。

教程中所有的源代码都在 src 目录下，可以直接编译运行。由于当前版本的 Java Tutorial 是基于 JDK 6.0 的，因此其中的示例程序也用到了 JDK 6.0 中的新增类库，但正则表达式在 JDK 1.4 就已经存在了，为了方便大家使用，改写了部分的源代码，源代码类名中后缀为“V4”的表示用于 JDK 1.4 或以上版本，“V5”的表示用于 JDK 5.0 或以上版本，没有这些后缀的类在各个版本中均可以正常使用。

* 这是由《程序员》杂志社评出的，刊登在《程序员》2007 年 3 月刊上。这七种基本技能是：数组，字符串与哈希表、正则表达式、调试、两门语言、一个开发环境、SQL 语言和编写软件的思想。

目录

昵称: glaivelee
园龄: 6年1个月
粉丝: 105
关注: 5

< 2010年12月 >

日	一	二	三	四	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	**15**	16	17	18
19	20	21	22	23	**24**	**25**
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

敦煌(12)
纪录片(12)
全集(12)
asp.net(2)
flex(1)
Java(1)
笔记本 硬件 参数 介绍(1)
大全(1)
教程(1)
华硕 ASUS 笔记本 K42DR 拆机 清灰 教程 图文详解(1)
更多

随笔分类(42)

ANDROID(1)
asp.net(3)
DataBase-Access
DataBase-SQL(3)
Dev(4)
Flash-ACTIONSCRIPT3(2)
IOS
java(1)
MOBILE
Others(7)
php(1)
project
VS2005

序返回目录

本文介绍如何使用 java.util.regex API 作为正则表达式模式匹配。虽然说这个包中可被接受的语法参数与 Perl 是相似的，但我们并不需要掌握 Perl 的语法知识。本教程将从基础开始，逐层深入到更多的高级技巧。下面是各章节的主要内容：

0 引言

粗略地看一下正则表达式，同时也介绍组成 API 的核心类。

1 测试用具

编写了一个简单的应用程序，用于测试正则表达式的模式匹配。

2 字符串

介绍基本的模式匹配、元字符和引用。

3 字符类

描述简单字符类、否定、范围、并集、交集和差集。

4 预定义字符类

描述空白字符、字母和数字字符等基本的预定义字符。

5 量词

使用贪婪（greedy）、勉强（reluctant）和侵占（possessive）量词，来匹配指定表达式 X 的次数。

6 捕获组

解释如何把多个字符作为一个单独的单元进行处理。

7 边界匹配器

描述行、单词和输入的边界。

8 Pattern 类的方法

测试了 Pattern 中一些有用的方法，以及探究一些高级的特性，诸如：带标记的编译和使用内嵌标记表达式。

9 Matcher 类的方法

描述了 Matcher 类中通常使用的方法。

10 PatternSyntaxException 类的方法

描述了如何检查一个 PatternSyntaxException 异常。

11 更多的资源

要了解更多正则表达式，可以参考这一节。

12 问题和练习

巩固一下本教程所介绍的正则表达式的基本知识，并附有答案。

为了区分文档中的正则表达式和普通字符串，均以\d[abc]{2}的形式表示正则表达式的模式。

0 引言返回目录

0.1 什么是正则表达式？返回目录

正则表达式（regular expressions）是一种描述字符串集的方法，它是以字符串集中各字符串的共有特征为

随笔档案(67)

相册(7)

最新评论

1. Re:Paypal注册教程及其相关提示（包括提取现金+手续费计算）

大家在注册账户，激活后会有一个步奏是账户认证，之前就两种方式：国内银行账户或信用卡。国内银行账户认证时，银行会打两笔小款，等待时间一般为3天左右，现在要是银联卡，就直接支持在线认证了。PayPal在职.....

--PayPal_William

2. Re:Paypal注册教程及其相关提示（包括提取现金+手续费计算）

好久的帖子了，楼主写的挺详细的。2015PayPal还是做了些改变，楼主主要是为海淘购物注册个人账户，如果是卖家的话，建议注册高级账户，这个时候在收款方面没有限制。大家在注册填写姓名时：（1）一定要使.....

--PayPal_William

3. Re:BAT 批处理脚本 教程

回头当文档看吧，这么长，果断没有想看的欲望。。。我只想写一个再简单不过的bat文件。。。

--木鸟飞

4. Re:BAT 批处理脚本 教程

mark下，使用时过来瞄2眼

--heaven_1025

5. Re:项目涉及到的50个Sql语句

是有用，但是你的有些题意不够明确？让人

依据的。正则表达式可以用于搜索、编辑或者是操作文本和数据。它超出了 Java 程序设计语言的标准语法，因此有必要去学习特定的语法来构建正则表达式。正则表达式的变化是复杂的，一旦你理解了它们是如何被构造的话，你就能解析或者构建任意的正则表达式了。

本教程讲授 java.util.regex API 所支持的正则表达式语法，以及介绍几个可运行的例子来说明不同的对象间是如何交互的。在正则表达式的世界中，有不同风格的选择，比如：grep[2]、Perl、Tcl、Python、PHP 和 awk。java.util.regex API 中的正则表达式语法与 Perl 中的最为相似。

0.2 java.util.regex 包是如何描述正则表达式的？返回目录

java.util.regex 包主要由三个类所组成：Pattern、Matcher 和 PatternSyntaxException。

Pattern 对象表示一个已编译的正则表达式。Pattern 类没有提供公共的构造方法。要构建一个模式，首先必须调用公共的静态 compile 方法，它将返回一个 Pattern 对象。这个方法接受正则表达式作为第一个参数。本教程的开始部分将教你必需的语法。

Matcher 是一个靠着输入的字符串来解析这个模式和完成匹配操作的对象。与 Pattern 相似，Matcher 也没有定义公共的构造方法，需要通过调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

PatternSyntaxException 对象是一个未检查异常，指示了正则表达式中的一个语法错误。

本教程的最后几节课程会详细地说明各个类。首当其冲的问题是：必须理解正则表达式是如何被构建的，因此下一节引入了一个简单的测试用具，重复地用于探究它们的语法。

1 测试用具返回目录

这节给出了一个可重用的测试用具 RegexTestHarness.java，用于探究构建 API 所支持的正则表达式。使用

java RegexTestHarness

这个命令来运行，没有被接受的命令行参数。这个应用会不停地循环执行下去[3]，提示用户输入正则表达式和字符串。虽然说使用这个测试用具是可选的，但你会发现它用于探究下文所讨论的测试用例将更为方便。

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTestHarness {

    public static void main(String[] args) {
        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        while (true) {
            Pattern pattern = Pattern.compile(console.readLine("%nEnter your regex: "));
            Matcher matcher = pattern.matcher(console.readLine("Enter input string to search: "));
            boolean found = false;
            while (matcher.find()) {
                console.format("I found the text \"%s\" starting at index %d " +
                        "and ending at index %d.%n",
                        matcher.group(), matcher.start(), matcher.end());
                found = true;
            }
            if (!found) {
                console.format("No match found.%n");
            }
        }
    }
}
```

在继续下一节之前，确认开发环境支持必需的包，并保存和编译这段代码。

【译者注】

要想一段时间才能相同的?

--往西走看日出

6. Re:DOS命令教程

作者，图片显示不出来啊。

--博客园_回音

阅读排行榜

评论排行榜

由于当前版本的 Java Tutorial 是基于 JDK 6.0 编写的，上述的测试用具由于使用到 JDK 6.0 中新增的类库（java.io.Console），所以该用具只能在 JDK 6.0 的环境中编译运行，由于 Console 访问操作系统平台上的控制台，因此这个测试用具只能在操作系统的字符控制台中运行，不能运行在 IDE 的控制台中。

正则表达式是 JDK 1.4 所增加的类库，为了兼容 JDK 1.4 和 JDK 5.0 的版本，重新改写了这个测试用具，让其能适用于不同的版本。

JDK 5.0 适用的测试用具（RegexTestHarnessV5.java，该用具可以在 IDE 中执行），建议 JDK 6.0 环境也采用该用具。

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexTestHarnessV5 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.printf("%nEnter your regex: ");
            Pattern pattern = Pattern.compile(scanner.nextLine());
            System.out.printf("Enter input string to search: ");
            Matcher matcher = pattern.matcher(scanner.nextLine());
            boolean found = false;
            while (matcher.find()) {
                System.out.printf(
                        "I found the text \"%s\" starting at index %d and ending at index %d.%n",
                        matcher.group(), matcher.start(), matcher.end()
                    );
                found = true;
            }
            if (!found) {
                System.out.printf("No match found.%n");
            }
        }
    }
}
```

JDK 1.4 适用的测试用具（RegexTestHarnessV4.java）：

```
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexTestHarnessV4 {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(
                new InputStreamReader(new BufferedInputStream(System.in))
            );
        while (true) {
            System.out.print("\nEnter your regex: ");
```

```
Pattern pattern = Pattern.compile(br.readLine());
System.out.print("Enter input string to search: ");
Matcher matcher = pattern.matcher(br.readLine());
boolean found = false;
while (matcher.find()) {
    System.out.println("I found the text \"" + matcher.group() +
            "\" starting at index " + matcher.start() +
            " and ending at index " + matcher.end() +
            ".");
    found = true;
}
if (!found) {
    System.out.println("No match found.");
}
    }
  }
}
```

2　字符串返回目录

在大多数的情况下，API所支持模式匹配的基本形式是匹配字符串，如果正则表达式是foo，输入的字符串也是foo，这个匹配将会是成功的，因为这两个字符串是相同的。试着用测试用具来测试一下：

Enter your regex: foo

Enter input string to search: foo

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

结果确实是成功的。注意当输入的字符串是 3 个字符长度的时候，开始的索引是 0，结束的索引是 3。这个是约定俗成的，范围包括开始的索引，不包括结束的索引，如下图所示：

图 1　字符串“foo”的单元格编号和索引值[4]

字符串中的每一个字符位于其自身的单元格（cell）中，在每个单元格之间有索引指示位。字符串“foo”始于索引 0 处，止于索引 3 处，即使是这些字符它们自己仅占据了 0、1 和 2 号单元格。

就子序列匹配而言，你会注意到一些重叠，下一次匹配开始索引与前一次匹配的结束索引是相同的：

Enter your regex: foo

Enter input string to search: foofoofoo

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

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

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

2.1　元字符返回目录

API 也支持许多可以影响模式匹配的特殊字符。把正则表达式改为cat.并输入字符串“cats”，输出如下所示：

Enter your regex: cat.

Enter input string to search: cats

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

虽然在输入的字符串中没有点（.），但这个匹配仍然是成功的。这是由于点（.）是一个元字符（metacharacters）（被这个匹配翻译成了具有特殊意义的字符了）。这个例子为什么能匹配成功的原因在于，元字符.指的是“任意字符”。

API 所支持的元字符有：([{\^-$|}])?*+.

注意：在学习过更多的如何构建正则表达式后，你会碰到这些情况：上面的这些特殊字符不应该被处理为元字符。然而也能够使用这个清单来检查一个特殊的字符是否会被认为是元字符。例如，字符 !、@ 和 # 决不会有特殊的意义。

有两种方法可以强制将元字符处理成为普通字符：

1. 在元字符前加上反斜线（\）；

2. 把它放在\Q（引用开始）和\E（引用结束）之间[5]。在使用这种技术时，\Q和\E能被放于表达式中的任何位置（假设先出现\Q[6]）

3　字符类返回目录

如果你曾看过 Pattern 类的说明，会看到一些构建正则表达式的概述。在这一节中你会发现下面的一些表达式：

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。

3.1　简单类（Simple Classes）返回目录

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。

```
Enter your regex: [bcr]at
    Enter input string to search: bat
    I found the text "bat" starting at index 0 and ending at index 3.
    Enter your regex: [bcr]at
    Enter input string to search: cat
    I found the text "cat" starting at index 0 and ending at index 3.
    Enter your regex: [bcr]at
    Enter input string to search: rat
    I found the text "rat" starting at index 0 and ending at index 3.
    Enter your regex: [bcr]at
    Enter input string to search: hat
    No match found.
```

在上面的例子中，在第一个字符匹配字符类中所定义字符中的一个时，整个匹配就是成功的。

3.1.1　否定返回目录

要匹配除那些列表之外所有的字符时，可以在字符类的开始处加上^元字符，这种就被称为否定（negation）。

```
Enter your regex: [^bcr]at
    Enter input string to search: bat
    No match found.
    Enter your regex: [^bcr]at
    Enter input string to search: cat
    No match found.
    Enter your regex: [^bcr]at
    Enter input string to search: rat
    No match found.
    Enter your regex: [^bcr]at
    Enter input string to search: hat
    I found the text "hat" starting at index 0 and ending at index 3.
```

在输入的字符串中的第一个字符不包含在字符类中所定义字符中的一个时，匹配是成功的。

3.1.2　范围返回目录

有时会想要定义一个包含值范围的字符类，诸如，"a 到 h"的字母或者是"1 到 5"的数字。指定一个范围，只要在被匹配的首字符和末字符间插入-元字符，比如：[1-5]或者是[a-h]。也可以在类里每个的边上放置不同的范围来提高匹配的可能性，例如：[a-zA-Z]将会匹配 a 到 z（小写字母）或者 A 到 Z（大写字母）中的任何一个字符。

下面是一些范围和否定的例子：

```
Enter your regex: [a-c]
    Enter input string to search: a
    I found the text "a" starting at index 0 and ending at index 1.
    Enter your regex: [a-c]
    Enter input string to search: b
    I found the text "b" starting at index 0 and ending at index 1.
    Enter your regex: [a-c]
```

Enter input string to search: c

I found the text "c" starting at index 0 and ending at index 1.

Enter your regex: [a-c]

Enter input string to search: d

No match found.

Enter your regex: foo[1-5]

Enter input string to search: foo1

I found the text "foo1" starting at index 0 and ending at index 4.

Enter your regex: foo[1-5]

Enter input string to search: foo5

I found the text "foo5" starting at index 0 and ending at index 4.

Enter your regex: foo[1-5]

Enter input string to search: foo6

No match found.

Enter your regex: foo[^1-5]

Enter input string to search: foo1

No match found.

Enter your regex: foo[^1-5]

Enter input string to search: foo6

I found the text "foo6" starting at index 0 and ending at index 4.

3.1.3 并集返回目录

可以使用并集（union）来建一个由两个或两个以上字符类所组成的单字符类。构建一个并集，只要在一个字符类的边上嵌套另外一个，比如：[0-4[6-8]]，这种奇特方式构建的并集字符类，可以匹配 0，1，2，3，4，6，7，8 这几个数字。

Enter your regex: [0-4[6-8]]

Enter input string to search: 0

I found the text "0" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]

Enter input string to search: 5

No match found.

Enter your regex: [0-4[6-8]]

Enter input string to search: 6

I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]

Enter input string to search: 8

I found the text "8" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]

Enter input string to search: 9

No match found.

3.1.4 交集返回目录

建一个仅仅匹配自身嵌套类中公共部分字符的字符类时，可以像[0-9&&[345]]中那样使用&&。这种方式构建出来的交集（intersection）简单字符类，仅仅以匹配两个字符类中的 3，4，5 共有部分。

Enter your regex: [0-9&&[345]]

Enter input string to search: 3

I found the text "3" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]

Enter input string to search: 4

I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]

Enter input string to search: 5

I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]

Enter input string to search: 2

No match found.

Enter your regex: [0-9&&[345]]

Enter input string to search: 6

No match found.

下面演示两个范围交集的例子：

Enter your regex: [2-8&&[4-6]]

Enter input string to search: 3

No match found.

Enter your regex: [2-8&&[4-6]]

Enter input string to search: 4

I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]

Enter input string to search: 5

I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]

Enter input string to search: 6

I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]

Enter input string to search: 7

No match found.

3.1.5 差集返回目录

最后，可以使用差集（subtraction）来否定一个或多个嵌套的字符类，比如：[0-9&&[^345]]，这个是构建一个匹配除 3，4，5 之外所有 0 到 9 间数字的简单字符类。

Enter your regex: [0-9&&[^345]]

Enter input string to search: 2

I found the text "2" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]

Enter input string to search: 3

No match found.

Enter your regex: [0-9&&[^345]]

Enter input string to search: 4

No match found.

Enter your regex: [0-9&&[^345]]

Enter input string to search: 5

No match found.

Enter your regex: [0-9&&[^345]]

Enter input string to search: 6

I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]

Enter input string to search: 9

I found the text "9" starting at index 0 and ending at index 1.

到此为止，已经涵盖了如何建立字符类的部分。在继续下一节之前，可以试着回想一下那张字符类表。

4 预定义字符类返回目录

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] 上表中，左列是构造右列字符类的简写形式。例如：\d指的是数字范围（0～9），\w指的是单词字符（任何大小写字母、下划线或者是数字）。无论何时都有可能使用预定义字符类，它可以使代码更易阅读，更易从难看的字符类中排除错误。

以反斜线（\）开始的构造称为转义构造（escaped constructs）。回顾一下在 字符串 一节中的转义构造，在那里我们提及了使用反斜线，以及用于引用的\Q和\E。在字符串中使用转义构造，必须在一个反斜线前再增加一

个反斜用于字符串的编译，例如：

```
private final String REGEX = "\\d";       // 单个数字
```

这个例子中\d是正则表达式，另外的那个反斜线是用于代码编译所必需的。但是测试用具读取的表达式，是直接从控制台中输入的，因此不需要那个多出来的反斜线。

下面的例子说明了预字义字符类的用法：

```
Enter your regex: .
    Enter input string to search: @
    I found the text "@" starting at index 0 and ending at index 1.
    Enter your regex: .
    Enter input string to search: 1
    I found the text "1" starting at index 0 and ending at index 1.
    Enter your regex: .
    Enter input string to search: a
    I found the text "a" starting at index 0 and ending at index 1.
    Enter your regex: \d
    Enter input string to search: 1
    I found the text "1" starting at index 0 and ending at index 1.
    Enter your regex: \d
    Enter input string to search: a
    No match found.
    Enter your regex: \D
    Enter input string to search: 1
    No match found.
    Enter your regex: \D
    Enter input string to search: a
    I found the text "a" starting at index 0 and ending at index 1.
    Enter your regex: \s
    Enter input string to search:
    I found the text " " starting at index 0 and ending at index 1.
    Enter your regex: \s
    Enter input string to search: a
    No match found.
    Enter your regex: \S
    Enter input string to search:
    No match found.
    Enter your regex: \S
    Enter input string to search: a
    I found the text "a" starting at index 0 and ending at index 1.
    Enter your regex: \w
    Enter input string to search: a
    I found the text "a" starting at index 0 and ending at index 1.
    Enter your regex: \w
    Enter input string to search: !
    No match found.
    Enter your regex: \W
    Enter input string to search: a
    No match found.
    Enter your regex: \W
    Enter input string to search: !
    I found the text "!" starting at index 0 and ending at index 1.
```

在开始的三个例子中，正则表达式是简单的，.（“点”元字符）表示“任意字符”，因此，在所有的三个例子（随意地选取了“@”字符，数字和字母）中都是匹配成功的。在接下来的例子中，都使用了预定义字符类表格中的单个正则表达式构造。你应该可以根据这张表指出前面每个匹配的逻辑：

\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

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。首先粗略地看一下，量词X?、X??和X?+都允许匹配 X 零次或一次，精确地做同样的事情，但它们之间有着细微的不同之处，在这节结束前会进行说明。

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 那我们现在就从贪婪量词开始，构建三个不同的正则表达式：字母a后面跟着?、*和+。接下来看一下，用这些表达式来测试输入的字符串是空字符串时会发生些什么：

```
Enter your regex: a?
        Enter input string to search:
        I found the text "" starting at index 0 and ending at index 0.
        Enter your regex: a*
        Enter input string to search:
        I found the text "" starting at index 0 and ending at index 0.
        Enter your regex: a+
        Enter input string to search:
        No match found.
```

5.1 零长度匹配返回目录

在上面的例子中，开始的两个匹配是成功的，这是因为表达式a?和a*都允许字符出现零次。就目前而言，这个例子不像其他的，也许你注意到了开始和结束的索引都是 0。输入的空字符串没有长度，因此该测试简单地在索引 0 上匹配什么都没有，诸如此类的匹配称之为零长度匹配（zero-length matches）。零长度匹配会出现在以下几种情况：输入空的字符串、在输入字符串的开始处、在输入字符串最后字符的后面，或者是输入字符串中任意两个字符之间。由于它们开始和结束的位置有着相同的索引，因此零长度匹配是容易被发现的。

我们来看一下关于零长度匹配更多的例子。把输入的字符串改为单个字符“a”，你会注意到一些有意思的事情：

```
Enter your regex: a?
        Enter input string to search: a
        I found the text "a" starting at index 0 and ending at index 1.
        I found the text "" starting at index 1 and ending at index 1.
        Enter your regex: a*
        Enter input string to search: a
        I found the text "a" starting at index 0 and ending at index 1.
        I found the text "" starting at index 1 and ending at index 1.
        Enter your regex: a+
        Enter input string to search: a
        I found the text "a" starting at index 0 and ending at index 1.
```

所有的三个量词都是用来寻找字母“a”的，但是前面两个在索引 1 处找到了零长度匹配，也就是说，在输入字符串最后一个字符的后面。回想一下，匹配把字符“a”看作是位于索引 0 和索引 1 之间的单元格中，并且测试用具一直循环下去直到不再有匹配为止。依赖于所使用的量词不同，最后字符后面的索引“什么也没有”的存在可以或者不可以触发一个匹配。

现在把输入的字符串改为一行 5 个“a”时，会得到下面的结果：

Enter your regex: a?

Enter input string to search: aaaaa

I found the text "a" starting at index 0 and ending at index 1.

I found the text "a" starting at index 1 and ending at index 2.

I found the text "a" starting at index 2 and ending at index 3.

I found the text "a" starting at index 3 and ending at index 4.

I found the text "a" starting at index 4 and ending at index 5.

I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a*

Enter input string to search: aaaaa

I found the text "aaaaa" starting at index 0 and ending at index 5.

I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a+

Enter input string to search: aaaaa

I found the text "aaaaa" starting at index 0 and ending at index 5.

在“a”出现零次或一次时，表达式a?寻找到所匹配的每一个字符。表达式a*找到了两个单独的匹配：第一次匹配到所有的字母“a”，然后是匹配到最后一个字符后面的索引 5。最后，a+匹配了所有出现的字母“a”，忽略了在最后索引处“什么都没有”的存在。

在这里，你也许会感到疑惑，开始的两个量词在遇到除了“a”的字母时会有什么结果。例如，在“ababaaaab”中遇到了字母“b”会发生什么呢?

下面我们来看一下：

Enter your regex: a?

Enter input string to search: ababaaaab

I found the text "a" starting at index 0 and ending at index 1.

I found the text "" starting at index 1 and ending at index 1.

I found the text "a" starting at index 2 and ending at index 3.

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

I found the text "a" starting at index 4 and ending at index 5.

I found the text "a" starting at index 5 and ending at index 6.

I found the text "a" starting at index 6 and ending at index 7.

I found the text "a" starting at index 7 and ending at index 8.

I found the text "" starting at index 8 and ending at index 8.

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

Enter your regex: a*

Enter input string to search: ababaaaab

I found the text "a" starting at index 0 and ending at index 1.

I found the text "" starting at index 1 and ending at index 1.

I found the text "a" starting at index 2 and ending at index 3.

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

I found the text "aaaa" starting at index 4 and ending at index 8.

I found the text "" starting at index 8 and ending at index 8.

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

Enter your regex: a+

Enter input string to search: ababaaaab

I found the text "a" starting at index 0 and ending at index 1.

I found the text "a" starting at index 2 and ending at index 3.

I found the text "aaaa" starting at index 4 and ending at index 8.

即使字母“b”在单元格 1、3、8 中出现，但在这些位置上的输出报告了零长度匹配。正则表达式a?不是特意地去寻找字母“b”，它仅仅是去找字母“a”存在或者其中缺少的。如果量词允许匹配“a”零次，任何输入的字符不是“a”时将会作为零长度匹配。在前面的例子中，根据讨论的规则保证了 a 被匹配。

对于要精确地匹配一个模式 n 次时，可以简单地在一对花括号内指定一个数值：

Enter your regex: a{3}

Enter input string to search: aa

No match found.

Enter your regex: a{3}

Enter input string to search: aaa

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

Enter your regex: a{3}

Enter input string to search: aaaa

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

这里，正则表确定式a{3}在一行中寻找连续出现三次的字母“a”。第一次测试失败的原由在于，输入的字符串没有足够的 a 用来匹配；第二次测试输出的字符串正好包括了三个“a”，触发了一次匹配；第三次测试也触发了一次匹配，这是由于在输出的字符串的开始部分正好有三个“a”。接下来的事情与第一次的匹配是不相关的，如果这个模式将在这一点后继续出现，那它将会触发接下来的匹配：

Enter your regex: a{3}

Enter input string to search: aaaaaaaaa

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

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

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

对于需要一个模式出现至少 n 次时，可以在这个数字后面加上一个逗号（,）：

Enter your regex: a{3,}

Enter input string to search: aaaaaaaaa

I found the text "aaaaaaaaa" starting at index 0 and ending at index 9.

输入一样的字符串，这次测试仅仅找到了一个匹配，这是由于一个中有九个“a”满足了“至少”三个“a”的要求。

最后，对于指定出现次数的上限，可以在花括号添加第二个数字。

Enter your regex: a{3,6} // 寻找一行中至少连续出现 3 个（但不多于 6 个）“a”

Enter input string to search: aaaaaaaaa

I found the text "aaaaaa" starting at index 0 and ending at index 6.

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

这里，第一次匹配在 6 个字符的上限时被迫终止了。第二个匹配包含了剩余的三个 a（这是匹配所允许最小的字符个数）。如果输入的字符串再少掉一个字母，这时将不会有第二个匹配，之后仅剩余两个 a。

5.2 捕获组和字符类中的量词返回目录

到目前为止，仅仅测试了输入的字符串包括一个字符的量词。实际上，量词仅仅可能附在一个字符后面一次，因此正则表达式abc+的意思就是“a 后面接着 b，再接着一次或者多次的 c”，它的意思并不是指abc一次或者多次。然而，量词也可能附在字符类和捕获组的后面，比如，[abc]+表示一次或者多次的 a 或 b 或 c，(abc)+表示一次或者多次的“abc”组。

我们来指定(dog)组在一行中三次进行说明。

Enter your regex: (dog){3}

Enter input string to search: dogdogdogdogdogdog

I found the text "dogdogdog" starting at index 0 and ending at index 9.

I found the text "dogdogdog" starting at index 9 and ending at index 18.

Enter your regex: dog{3}

Enter input string to search: dogdogdogdogdogdog

No match found.

上面的第一个例子找到了三个匹配，这是由于量词用在了整个捕获组上。然而，把圆括号去掉，这时的量词{3}现在仅用在了字母“g”上，从而导致这个匹配失败。

类似地，也能把量词应用于整个字符类：

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.

上面的第一个例子中，量词{3}应用在了整个字符类上，但是第二个例子这个量词仅用在字母“c”上。

5.3 贪婪、勉强和侵占量词间的不同返回目录

在贪婪、勉强和侵占三个量词间有着细微的不同。

贪婪量词之所以称之为“贪婪的”，这是由于它们强迫匹配器读入（或者称之为吃掉）整个输入的字符串，来优先尝试第一次匹配，如果第一次尝试匹配（对于整个输入的字符串）失败，匹配器会通过回退整个字符串的一个字符再一次进行尝试，不断地进行处理直到找到一个匹配，或者左边没有更多的字符来用于回退了。赖于在表达式中使用的量词，最终它将尝试地靠着 1 或 0 个字符的匹配。

但是，勉强量词采用相反的途径：从输入字符串的开始处开始，因此每次勉强地吞噬一个字符来寻找匹配，最终它们会尝试整个输入的字符串。

最后，侵占量词始终是吞掉整个输入的字符串，尝试着一次（仅有一次）匹配。不像贪婪量词那样，侵占量词绝不会回退，即使这样做是允许全部的匹配成功。

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　捕获组返回目录

在上一节中，学习了每次如何把量词放在一个字符、字符类或者捕获组中。到目前为止，还没有详细地讨论过捕获组的概念。

捕获组（capturing group）是将多个字符作为单独的单元来对待的一种方式。构建它们可以通过把字符放在一对圆括号中而成为一组。例如，正则表达式(dog)建了单个的组，包括字符“d”“o”和“g”。匹配捕获组输入的字符串部分将会存放于内存中，稍后通过反向引用再次调用。（在 6.2 节 中将会讨论反向引用）

6.1　编号方式返回目录

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)

要找出当前的表达式中有多少组，通过调用 Matcher 对象的 groupCount 方法。groupCount 方法返回 int 类型值，表示当前 Matcher 模式中捕获组的数量。例如，groupCount 返回 4 时，表示模式中包含有 4 个捕获组。

有一个特别的组——组 0，它表示整个表达式。这个组不包括在 groupCount 的报告范围内。以(?开始的组是纯粹的非捕获组（non-capturing group），它不捕获文本，也不作为组总数而计数。（可以看 8 Pattern 类的方法 一节中非捕获组的例子。）

Matcher 中的一些方法，可以指定 int 类型的特定组号作为参数，因此理解组是如何编号的是尤为重要的。

public int start(int group)：返回之前的匹配操作期间，给定组所捕获的子序列的初始索引。

public int end(int group)：返回之前的匹配操作期间，给定组所捕获子序列的最后字符索引加 1。

public String group (int group)：返回之前的匹配操作期间，通过给定组而捕获的输入子序列。

6.2　反向引用返回目录

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　边界匹配器返回目录

就目前而言，我们的兴趣在于指定输入字符串中某些位置是否有匹配，还没有考虑到字符串的匹配产生在什么地方。

通过指定一些边界匹配器（boundary matchers）的信息，可以使模式匹配更为精确。比如说你对某个特定的单词感兴趣，并且它只出现在行首或者是行尾时。又或者你想知道匹配发生在单词边界（word boundary），或者是上一个匹配的尾部。

下表中列出了所有的边界匹配器及其说明。

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。[7]

8 Pattern 类的方法返回目录

到目前为止，仅使用测试用具来建立最基本的 Pattern 对象。在这一节中，我们将探讨一些诸如使用标志构建模式、使用内嵌标志表达式等高级的技术。同时也探讨了一些目前还没有讨论过的其他有用的方法。

8.1 使用标志构建模式返回目录

Pattern 类定义了备用的 compile 方法，用于接受影响模式匹配方式的标志集。标志参数是一个位掩码，可以是下面公共静态字段中的任意一个：

Pattern.CANON_EQ

启用规范等价。在指定此标志后，当且仅当在其完整的规范分解匹配时，两个字符被视为匹配。例如，表达式 a\u030A[8]在指定此标志后，将匹配字符串“\u00E5”（即字符 å）。默认情况下，匹配不会采用规范等价。指定此标志可能会对性能会有一定的影响。

Pattern.CASE_INSENSITIVE

启用不区分大小写匹配。默认情况下，仅匹配 US-ASCII 字符集中的字符。Unicode 感知（Unicode-aware）的不区分大小写匹配，可以通过指定 UNICODE_CASE 标志连同此标志来启用。不区分大小写匹配也能通过内嵌标志表达式(?i)来启用。指定此标志可能会对性能会有一定的影响。

Pattern.COMMENTS

模式中允许存在空白和注释。在这种模式下，空白和以#开始的直到行尾的内嵌注释会被忽略。注释模式也能通过内嵌标志表达式(?x)来启用。

Pattern.DOTALL

启用 dotall 模式。在 dotall 模式下，表达式.匹配包括行结束符在内的任意字符。默认情况下，表达式不会匹配行结束符。dotall 模式也通过内嵌标志表达式(?x)来启用。［s 是“单行（single-line）”模式的助记符，与 Perl 中的相同。］

Pattern.LITERAL

启用模式的字面分析。指定该标志后，指定模式的输入字符串作为字面上的字符序列来对待。输入序列中的元字符和转义字符不具有特殊的意义了。CASE_INSENSITIVE 和 UNICODE_CASE 与此标志一起使用时，会对匹配产生一定的影响。其他的标志就变得多余了。启用字面分析没有内嵌标志表达式。

Pattern.MULTILINE

启用多行（multiline）模式。在多行模式下，表达式^和$分别匹配输入序列行结束符前面和行结束符的前面。默认情况下，表达式仅匹配整个输入序列的开始和结尾。多行模式也能通过内嵌标志表达式(?m)来启用。

Pattern.UNICODE_CASE

启用可折叠感知 Unicode（Unicode-aware case folding）大小写。在指定此标志后，需要通过 CASE_INSENSITIVE 标志来启用，不区分大小写区配将在 Unicode 标准的意义上来完成。默认情况下，不区分大小写匹配仅匹配 US-ASCII 字符集中的字符。可折叠感知 Unicode 大小写也能通过内嵌标志表达式(?u)来启用。指定此标志可能会对性能会有一定的影响。

Pattern.UNIX_LINES

启用 Unix 行模式。在这种模式下，.、^和$的行为仅识别“\n”的行结束符。Unix 行模式可以通过内嵌标志表达式(?d)来启用。

接下来，将修改测试用具 RegexTestHarness.java，用于构建不区分大小写匹配的模式。

首先，修改代码去调用 complie 的另外一个备用的方法：

```
Pattern pattern = Pattern.compile(
    console.readLine("%nEnter your regex: "),
    Pttern.CASE_INSENSITIVE
);
```

编译并运行这个测试用具，会得出下面的结果：

```
Enter your regex: dog
        Enter input string to search: DoGDOg
        I found the text "DoG" starting at index 0 and ending at index 3.
        I found the text "DOg" starting at index 3 and ending at index 6.
```

正如你所看到的，不管是否大小写，字符串字面上是“dog”的都产生了匹配。使用多个标志来编译一个模式，使用按位或操作符“|”分隔各个标志。为了更清晰地说明，下面的示例代码使用硬编码（hardcode）的方式，来取代控制台中的读取：

```
pattern = Pattern.compile("[az]$", Pattern.MULTILINE | Pattern.UNIX_LINES);
```

也可以使用一个 int 类型的变量来代替：

```
final int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
Pattern pattern = Pattern.compile("aa", flags);
```

8.2 内嵌标志表达式返回目录

使用内嵌标志表达式（embedded flag expressions）也可以启用不同的标志。对于两个参数的 compile 方法，内嵌标志表达式是可选的，因为它在自身的正则表达式中被指定了。下面的例子使用最初的测试用具（RegexTestHarness.java），使用内嵌标志表达式(?i)来启用不区分大小写的匹配。

```
Enter your regex: (?i)foo
        Enter input string to search: FOOfooFoOfoO
        I found the text "FOO" starting at index 0 and ending at index 3.
        I found the text "foo" starting at index 3 and ending at index 6.
        I found the text "FoO" starting at index 6 and ending at index 9.
        I found the text "foO" starting at index 9 and ending at index 12.
```

所有匹配无关大小写都一次次地成功了。

内嵌标志表达式所对应 Pattern 的公用的访问字段表示如下表：

常　　量	等价的内嵌标志表达式
Pattern.CANON_EQ	没有
Pattern.CASE_INSENSITIVE	(?i)
Pattern.COMMENTS	(?x)
Pattern.MULTILINE	(?m)
Pattern.DOTALL	(?s)
Pattern.LITERAL	没有
Pattern.UNICODE_CASE	(?u)
Pattern.UNIX_LINES	(?d)

8.3 使用 matches(String, CharSequence) 方法返回目录

Pattern 类定义了一个方便的 matches 方法，用于快速地检查模式是否表示给定的输入字符串。与使用所有的公共静态方法一样，应该通过它的类名来调用 matches 方法，诸如 Pattern.matches("\\d","1");。这个例子中，方法返回 true，这是由于数字“1”匹配了正则表达式\d。

8.4 使用 split(String) 方法返回目录

split 方法是一个重要的工具，用于收集依赖于被匹配的模式任一边的文本。如下面的 SplitDemo.java 所示，split 方法能从“one:two:three:four:five”字符串中解析出“one two three four five”单词：

```
import java.util.regex.Pattern;

public class SplitDemo {

    private static final String REGEX = ":";
    private static final String INPUT = "one:two:three:four:five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
```

```
            System.out.println(s);
        }
    }
}
```

输出：

```
one
        two
        three
        four
        five
```

简而言之，已经使用冒号（:）取代了复杂的正则表达式匹配字符串文字。以后仍会使用 Pattern 和 Matcher 对象，也能使用 split 得到位于任意正则表达式各边的文本。下面的 SplitDemo2.java 是个一样的例子，使用数字作为 split 的参数：

```
import java.util.regex.Pattern;

public class SplitDemo2 {

    private static final String REGEX = "\\d";
    private static final String INPUT = "one9two4three7four1five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

输出：

```
one
        two
        three
        four
        five
```

8.5 其他有用的方法返回目录

你可以从下面的方法中找到比较好用的方法：

public static String quote(String s)[9]：返回指定字符串字面模式的字符串。此方法会产生一个字符串，能被用于构建一个与字符串 s 匹配的 Pattern，好像它是一个字面上的模式。输入序列中的元字符和转义序列将没有特殊的意义了。

public String toString()：返回这个模式的字符串表现形式。这是一个编译过的模式中的正则表达式。

8.6 在 java.lang.String 中等价的 Pattern 方法返回目录

java.lang.String 通过模拟 java.util.regex.Pattern 行为的几个方法，也可以支持正则表达式。方便起见，下面主要摘录了出现在 API 关键的方法。

public boolean matches(String regex)：告知字符串是否匹配给定的正则表达式。调用 str.matches(regex)方法所产生的结果与作为表达式的 Pattern.matches(regex, str)的结果是完全一致。

public String[] split(String regex, int limit)：依照匹配给定的正则表达式来拆分字符串。调用 str.split(regex, n)方法所产生的结果与作为表达式的 Pattern.compile(regex).split(str, n) 的结果完全一致。

public String[] split(String regex)：依照匹配给定的正则表达式来拆分字符串。这个方法与调用两个参数的 split 方法是相同的，第一个参数使用给定的表达式，第二个参数限制为 0。在结果数组中不包括尾部的空字符串。

还有一个替换方法，把一个 CharSequence 替换成另外一个：

public String replace(CharSequence target,CharSequence replacement)：将字符串中每一个匹配替换匹配字面目标序列的子字符串，替换成指定的字面替换序列。这个替换从字符串的开始处理直至结束，例如，把

字符串“aaa”中的“aa”替换成“b”，结果是“ba”，而不是“ab”。

9　Matcher 类的方法返回目录

在这一节中来看看 Matcher 类中其他一些有用的方法。方便起见，下面列出的方法是按照功能来分组的。

索引方法

索引方法（index methods）提供了一些正好在输入字符串中发现匹配的索引值：

public int start()：返回之前匹配的开始索引。

public int start(int group)：返回之前匹配操作中通过给定组所捕获序列的开始索引。

public int end()：返回最后匹配字符后的偏移量。

public int end(int group)：返回之前匹配操作中通过给定组所捕获序列的最后字符之后的偏移量。

研究方法

研究方法（study methods）回顾输入的字符串，并且返回一个用于指示是否找到模式的布尔值。

public boolean lookingAt()：尝试从区域开头处开始，输入序列与该模式匹配。

public boolean find()：尝试地寻找输入序列中，匹配模式的下一个子序列。

public boolean find(int start)：重置匹配器，然后从指定的索引处开始，尝试地寻找输入序列中，匹配模式的下一个子序列。

public boolean matches()：尝试将整个区域与模式进行匹配

替换方法

替换方法（replacement methods）用于在输入的字符串中替换文本有用处的方法。

public Matcher appendReplacement(StringBuffer sb, String replacement)：实现非结尾处的增加和替换操作。

public StringBuffer appendTail(StringBuffer sb)：实现结尾处的增加和替换操作。

public String replaceAll(String replacement)：使用给定的替换字符串来替换输入序列中匹配模式的每一个子序列。

public String replaceFirst(String replacement)：使用给定的替换字符串来替换输入序列中匹配模式的第一个子序列。

public static String quoteReplacement(String s)：返回指定字符串的字面值来替换字符串。这个方法会生成一个字符串，用作 Matcher 的 appendReplacement 方法中的字面值替换 s。所产生的字符串将与作为字面值序列的 s 中的字符序列匹配。斜线（\）和美元符号（$）将不再有特殊意义了。

9.1　使用 start 和 end 方法返回目录

示例程序 MatcherDemo.java 用于计算输入序列中单词“dog”的出现次数。

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MatcherDemo {

    private static final String REGEX = "\\bdog\\b";
    private static final String INPUT = "dog dog dog doggie dogg";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       Matcher m = p.matcher(INPUT);        // 获得匹配器对象
       int count = 0;
       while (m.find()) {
          count++;
          System.out.println("Match number " + count);
          System.out.println("start(): " + m.start());
          System.out.println("end(): " + m.end());
       }
    }
}
```

输出：

```
Match number 1
          start(): 0
          end(): 3
```

Match number 2

start(): 4

end(): 7

Match number 3

start(): 8

end(): 11

可以看出，这个例子使用了单词边界，用于确保更长单词中的字母“d”“o”“g”就不是子串了。它也输出了一些有用的信息，在输入的字符串中什么地方有匹配。start 方法返回在以前的匹配操作期间，由给定组所捕获子序列的开始处索引，end 方法返回匹配到最后一个字符索引加 1。

9.2　使用 matches 和 lookingAt 方法返回目录

matches 和 lookingAt 方法都是尝试该模式匹配输入序列。然而不同的是，matches 要求匹配整个输入字符串，而 lookingAt 不是这样。这两个方法都是从输入字符串的开头开始的。下面是 MatchesLooking.java 完整的代码：

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] args) {

        // 初始化
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: " + REGEX);
        System.out.println("Current INPUT is: " + INPUT);

        System.out.println("lookingAt(): " + matcher.lookingAt());
        System.out.println("matches(): " + matcher.matches());
    }
}
```

输出：

```
Current REGEX is: foo
        Current INPUT is: fooooooooooooooooo
        lookingAt(): true
        matches(): false
```

9.3　使用 replaceFirst(String) 和 replaceAll(String) 方法返回目录

replaceFirst 和 replaceAll 方法替换匹配给定正则表达式的文本。从它们的名字可以看出，replaceFirst 替换第一个匹配到的，而 replaceAll 替换所有匹配的。下面是 ReplaceDemo.java 的代码：

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ReplaceDemo {

    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
```

```
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);       // 获得匹配器对象
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

输出：

The cat says meow. All cats say meow.

在上面的例子中，所有的 dog 都被替换成了 cat。但是为什么在这里停下来了呢？你可以替换匹配任何正则表达式的文本，这样优于替换一个简单的像 dog 一样的文字。

这个方法的 API 描述了"给定正则表达式a*b，在输入'aabfooaabfooabfoob'和替换的字符串是'-'情况下，表达式的匹配器调用方法后，会产生成字符串'-foo-foo-foo-'。"

下面是 ReplaceDemo2.java 的代码：

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class ReplaceDemo2 {

    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);       // 获得匹配器对象
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

输出：

-foo-foo-foo-

仅要替换模式一次时，可以简单地调用 replaceFirst 用于取代 replaceAll，它接受相同的参数。

9.4 使用 appendReplacement(StringBuffer, String) 和 appendTail(StringBuffer) 方法返回目录

Matcher 类也提供了 appendReplacement 和 appendTail 两个方法用于文本替换。下面的这个例子（RegexDemo.java）使用了这两个方法完成与 replaceAll 相同的功能。

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexDemo {
    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);       // 获得匹配器对象
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
```

```
}
```

输出：

```
-foo-foo-foo-
```

9.5 在 java.lang.String 中等价的 Matcher 方法返回目录

为了使用方便，String 类看上去还不错地模仿了 Matcher 的两个方法：

public String replaceFirst(String regex, String replacement)：使用给定的替换字符串替换该字符串中匹配了给定正则表达式的第一个子字符串。调用 str.replaceFirst(regex, repl)方法与使用 Pattern.compile(regex).matcher(str).replaceFirst(repl)产生的结果是完全相同的。

public String replaceAll(String regex, String replacement)：使用给定的替换字符串替换该字符串中匹配了给定正则表达式的每一个子字符串。调用 str.replaceAll(regex, repl)方法与使用 Pattern.compile(regex).matcher(str).replaceAll(repl)产生的结果是完全相同的。

10 PatternSyntaxException 类的方法返回目录

PatternSyntaxException 是未检查异常，指示正则表达式模式中的语法错误。PatternSyntaxException 类提供了下面的一些方法，用于确定在什么地方发生了错误：

public String getDescription()：获得错误描述。

public int getIndex()：获得错误索引。

public String getPattern()：获得字符串形式的错误正则表达式。

public String getMessage()：获得一个多行的字符串，包括语法错误和错误的索引、错误的正则表达式模式，以及模式内可视化的索引指示。

下面的源代码（RegexTestHarness2.java[10]）更新了测试用具，用于检查不正确的正则表达式：

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.regex.PatternSyntaxException;

public class RegexTestHarness2 {

public static void main(String[] args){
    Pattern pattern = null;
    Matcher matcher = null;

    Console console = System.console();
    if (console == null) {
        System.err.println("No console.");
        System.exit(1);
    }
    while (true) {
        try {
            pattern = Pattern.compile(console.readLine("%nEnter your regex: "));
            matcher = pattern.matcher(console.readLine("Enter input string to search: "));
        } catch (PatternSyntaxException pse){
    console.format("There is a problem with the regular expression!%n");
    console.format("The pattern in question is: %s%n", pse.getPattern());
    console.format("The description is: %s%n", pse.getDescription());
    console.format("The message is: %s%n", pse.getMessage());
    console.format("The index is: %s%n", pse.getIndex());
    System.exit(0);
}
boolean found = false;
while (matcher.find()) {
    console.format("I found the text \"%s\" starting at " +
            "index %d and ending at index %d.%n",
            matcher.group(), matcher.start(), matcher.end()
        );
```

```
            found = true;
        }
        if (!found){
            console.format("No match found.%n");
        }
      }
   }
}
```

运行该测试，输入?i)foo作为正则表达式。这是个臆想出来的错误，程序员在使用内嵌标志表达式(?i)时忘记输入左括号了。这样做会产生下面的结果：

```
Enter your regex: ?i)
        There is a problem with the regular expression!
        The pattern in question is: ?i)
        The description is: Dangling meta character '?'
        The message is: Dangling meta character '?' near index 0
        ?i)
        ^
        The index is: 0
```

从这个输出中，可以看出在索引 0 处的元字符（?）附近有语法错误。缺少左括号是导致这个错误的最魁祸首。

11 更多的资源返回目录

现在已经结束了正则表达式的课程，你也许会发现，主要引用了 Pattern、Matcher 和 PatternSyntaxException 类的 API 文档。

构建正则表达式更详细地描述，推荐阅读 Jeffrey E.F.Friedl 的Mastering Regular Expressions[11]。

12 问题和练习返回目录

〖问题〗

1. 在 java.util.regex 包中有哪三个公共的类？描述一下它们的作用。
2. 考虑一下字符串“foo”，它的开始索引是多少？结束索引是多少？解释一下这些编号的意思。
3. 普通字符和元字符有什么不同？各给出它们的一个例子。
4. 如何把元字符表现成像普通字符那样?
5. 附有方括号的字符集称为什么？它有什么作用?
6. 这里是三个预定义的字符类：\d、\s和\w。描述一下它们各表示什么？并使用方括号的形式将它们重写。
7. 对于\d、\s和\w，写出两个简单的表达式，匹配它们相反的字符集。
8. 思考正则表达式(dog){3}，识别一下其中的两个子表达式。这个表达式会匹配什么字符串?

〖练习〗

1. 使用反向引用写一个表达式，用于匹配一个人的名字，假设这个人的 first 名字与 last 名字是相同的。

【问题答案】

1. 问：在 java.util.regex 包中有哪三个公共的类？描述一下它们的作用。

答：

编译后的 Pattern 实例表示正则表达式。

Matcher 实例是解析模式和靠着输入的字符串完成匹配操作的引擎。

PatternSyntaxException 定义一个未检查异常，指示正则表达式中的语法错误。

2. 问：考虑一下字符串“foo”，它的开始索引是多少？结束索引是多少？解释一下这些编号的意思。

答：字符串中的每一个字符位于其自身的单元格中。索引位置在两个单元格之间。字符串“foo”开始于索引 0，结束于索引 3，即便是这些字符仅占用了 0、1 和 2 号单元格。

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。

5. 问：附有方括号的字符集称为什么？它有什么作用?

答：是一个字符类。通过方括号间的表达式，匹配指定字符类中的任意一个字符。

6. 问：这里是三个预定义的字符类：\d、\s和\w。描述一下它们各表示什么？并使用方括号的形式将它们重写。

答：\d 匹配任意数字[0-9]

\s 匹配任意空白字符[\t\n-x0B\f\r]

\w 匹配任意单词字符[a-zA-Z_0-9]

7. 问：对于\d、\s和\w，写出两个简单的表达式，匹配它们相反的字符集。

答：\d \D [^\d]

\s \S [^\s]

\w \W [^\w]

8. 问：思考正则表达式(dog){3}，识别一下其中的两个子表达式。这个表达式会匹配什么字符串?

答：表达式由捕获组(dog)和接着的贪婪量词{3}所组成。它匹配字符串“dogdogdog”。

【练习答案】

1. 练习：使用反向引用写一个表达式，用于匹配一个人的名字，假设这个人的 first 名字与 last 名字是相同的。

解答：([A-Z][a-zA-Z]*)\s\1

注释返回目录

[1] 本文全文译自 Java Tutorial 的 Regular Expressions，标题是译者自拟的。——译者注

[2] Unix 工具，用于文件中的字符串查找，它是最早的正则表达式工具之一。——译者注

[3] 若要退出可以使用 Ctrl + C 来中断。——译者注

[4] 图中的“索引 3”指示是译者所加，原文中并没有。——译者注

[5] 这种方式在 JDK 6.0 以前版本使用需要注意，在字符类中使用这种结构是有 bug 的，不过在 JDK 6.0 中已经修正。——译者注

[6] 若\E前没有\Q时会产生 PatternSyntaxException 异常指示语法错误。——译者注

[7] 第一次匹配时仅匹配字符串的开始部分，与\A类似。（引自 Jeffrey E.F.Friedl, Mastering Regular Expressions, 3rd ed., §3.5.3.3, O'Reilly, 2006.）——译者注

[8] \u030A，即字符 å 上半部分的小圆圈（？）（该字符在 IE 浏览器上无法正确显示，在 Firefox 浏览器上可以正常地显示）。——译者注

[9] JDK 5.0 新增的方法，JDK 1.4 中不能使用。——译者注

[10] JDK 1.4 和 JDK 5.0 适用的版本在所附的源代码中。适用于 JDK 1.4 的文件名为 RegexTestHarness2V4.java，JDK 1.5 的文件名为 RegexTestHarness2V5.java。——译者注

[11] 第三版是本书的最新版本。第三版的中译本《精通正则表达式》已由电子工业出版社于 2007 年 7 月出版。——译者注

译后记返回目录

带着忐忑不安的心情完成了我的第一篇译篇，但愿这个教程能让大家对 Java 中的正则表达式有更一步的认识。

虽然这是一个关于 Java 正则表达式很好的一个入门教程，但这个教程也有其不足之处，其中仅仅涉及了最为简单的正则表达式，对介绍到的有些问题并未完全展开，比如：字符类中的转义、内嵌标志表达式具体的用法等。对有些常用的表达式，如|（选择结构）也没有涉及。对于非捕获组来说，仅仅提到了内嵌标志表达式，对于诸如(?:X)、(?=X)、(?!X)、(?<=X)、(?<!X)、(?>X)等等之类的非捕获组结构完全没有涉及。正如译者在序中提到的，这篇文章只为今后学习更高级的正则表达式技术奠定良好的基础。

分类: java

标签: Java, 正则表达式, 教程

glaivelee
关注 - 5
粉丝 - 105
+加关注

« 上一篇: 纪录片《敦煌》全集链接
» 下一篇: 神奇3D圣诞树祝广大技术人员圣诞快乐!

posted @ 2010-12-25 19:21 glaivelee 阅读(1740) 评论(0) 编辑 收藏

最新IT新闻:

- 马斯克超级高铁或首先在亚洲建成
- 中国智能手机市场萎缩 全球占比从29%下滑至25%
- 大公司不来 美国22个小镇自发建千兆光纤宽带
- 美国运营商们开始推不限流量的包月套餐了
- Windows 10变身服务 有几点可能你不太了解

» 更多新闻...

最新知识库文章:

- 也谈怎么学好英语这件小事
- 前后端分离了，然后呢?
- 撰写合格的REST API
- 深入NGINX：我们如何设计它的性能和扩展性
- 文化编码（Coding Culture）：帮你构建更强的团队，创造更好的产品

» 更多知识库文章...