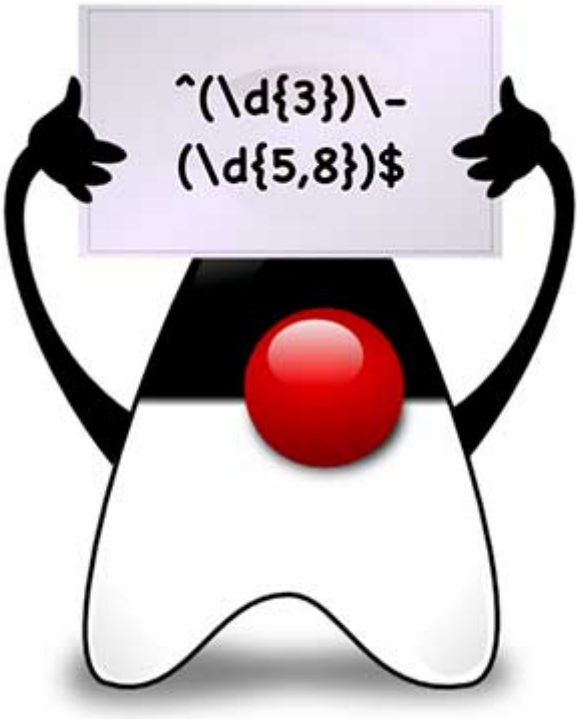
11 正则表达式

正则表达式是一种用来匹配字符串的强有力的武器。Java内置了强大的正则表达式的支持。

本章我们会详细介绍如何在Java程序中使用正则表达式。



正则表达式简介

在了解正则表达式之前，我们先看几个非常常见的问题：

- 如何判断字符串是否是有效的电话号码？例如：`010-1234567`，`123ABC456`，`13510001000`等；
- 如何判断字符串是否是有效的电子邮件地址？例如：`test@example.com`，`test#example`等；
- 如何判断字符串是否是有效的时间？例如：`12:34`，`09:60`，`99:99`等。

一种直观的想法是通过程序判断，这种方法需要为每种用例创建规则，然后用代码实现。下面是判断手机号的代码：

```
boolean isValidMobileNumber(String s) {
    // 是否是11位?
    if (s.length() != 11) {
        return false;
    }
    // 每一位都是0~9:
    for (int i=0; i<s.length(); i++) {
```

```
            char c = s.charAt(i);
            if (c < '0' || c > '9') {
                return false;
            }
        }
        return true;
    }
```

上述代码仅仅做了非常粗略的判断，并未考虑首位数字不能为`0`等更详细的情况。

除了判断手机号，我们还需要判断电子邮件地址、电话、邮编等等：

- boolean isValidMobileNumber(String s) { ... }
- boolean isValidEmail(String s) { ... }
- boolean isValidPhoneNumber(String s) { ... }
- boolean isValidZipCode(String s) { ... }
- ...

为每一种判断逻辑编写代码实在是太繁琐了。有没有更简单的方法？

有！用正则表达式！

正则表达式可以用字符串来描述规则，并用来匹配字符串。例如，判断手机号，我们用正则表达式`\d{11}`：

```
boolean isValidMobileNumber(String s) {
    return s.matches("\\d{11}");
}
```

使用正则表达式的好处有哪些？一个正则表达式就是一个描述规则的字符串，所以，只需要编写正确的规则，我们就可以让正则表达式引擎去判断目标字符串是否符合规则。

正则表达式是一套标准，它可以用于任何语言。Java标准库的`java.util.regex`包内置了正则表达式引擎，在Java程序中使用正则表达式非常简单。

举个例子：要判断用户输入的年份是否是`20##`年，我们先写出规则如下：

一共有4个字符，分别是：`2`，`0`，`0~9任意数字`，`0~9任意数字`。

对应的正则表达式就是：`20\d\d`，其中`\d`表示任意一个数字。

把正则表达式转换为Java字符串就变成了`20\\d\\d`，注意Java字符串用`\\`表示`\`。

最后，用正则表达式匹配一个字符串的代码如下：

```
public class Main {
    public static void main(String[] args) {
        String regex = "20\\d\\d";
        System.out.println("2019".matches(regex)); // true
        System.out.println("2100".matches(regex)); //
false
    }
}
```

可见，使用正则表达式，不必编写复杂的代码来判断，只需给出一个字符串表达的正则规则即可。

小结

- 正则表达式是用字符串描述的一个匹配规则，使用正则表达式可以快速判断给定的字符串是否符合匹配规则。Java标准库`java.util.regex`内建了正则表达式引擎。

匹配规则

正则表达式的匹配规则是从左到右按规则匹配。我们首先来看如何使用正则表达式来做精确匹配。

对于正则表达式`abc`来说，它只能精确地匹配字符串`"abc"`，不能匹配`"ab"`，`"Abc"`，`"abcd"`等其他任何字符串。

如果正则表达式有特殊字符，那就需要用`\`转义。例如，正则表达式`a\&c`，其中`\&`是用来匹配特殊字符`&`的，它能精确匹配字符串`"a&c"`，但不能匹配`"ac"`、`"a-c"`、`"a&&c"`等。

要注意正则表达式在Java代码中也是一个字符串，所以，对于正则表达式`a\&c`来说，对应的Java字符串是`"a\\&c"`，因为`\`也是Java字符串的转义字符，两个`\\`实际上表示的是一个`\`：

```
public class Main {
    public static void main(String[] args) {
        String re1 = "abc";
        System.out.println("abc".matches(re1));
        System.out.println("Abc".matches(re1));
        System.out.println("abcd".matches(re1));

        String re2 = "a\\&c"; // 对应的正则是a\&c
        System.out.println("a&c".matches(re2));
        System.out.println("a-c".matches(re2));
        System.out.println("a&&c".matches(re2));
    }
}
```

如果想匹配非ASCII字符，例如中文，那就用`\u####`的十六进制表示，例如：`a\u548cc`匹配字符串`"a和c"`，中文字符`和`的Unicode编码是`548c`。

匹配任意字符

精确匹配实际上用处不大，因为我们直接用`String.equals()`就可以做到。大多数情况下，我们想要的匹配规则更多的是模糊匹配。我们可以用`.`匹配一个任意字符。

例如，正则表达式`a.c`中间的`.`可以匹配一个任意字符，例如，下面的字符串都可以被匹配：

- `"abc"`，因为`.`可以匹配字符`b`；
- `"a&c"`，因为`.`可以匹配字符`&`；
- `"acc"`，因为`.`可以匹配字符`c`。

但它不能匹配`"ac"`、`"a&&c"`，因为`.`匹配一个字符且仅限一个字符。

匹配数字

用`.`可以匹配任意字符，这个口子开得有点大。如果我们只想匹配`0`~`9`这样的数字，可以用`\d`匹配。例如，正则表达式`00\d`可以匹配：

- `"007"`，因为`\d`可以匹配字符`7`；
- `"008"`，因为`\d`可以匹配字符`8`。

它不能匹配`"00A"`，`"0077"`，因为`\d`仅限单个数字字符。

匹配常用字符

用`\w`可以匹配一个字母、数字或下划线，w的意思是word。例如，`java\w`可以匹配：

- `"javac"`，因为`\w`可以匹配英文字符`c`；
- `"java9"`，因为`\w`可以匹配数字字符`9`；。
- `"java_"`，因为`\w`可以匹配下划线`_`。

它不能匹配`"java#"`，`"java "`，因为`\w`不能匹配`#`、空格等字符。

匹配空格字符

用`\s`可以匹配一个空格字符，注意空格字符不但包括空格`，还包括tab字符（在Java中用\t表示）。例如，a\sc`可以匹配：

- `"a c"`，因为`\s`可以匹配空格字符``；
- `"a	c"`，因为`\s`可以匹配tab字符`\t`。

它不能匹配`"ac"`，`"abc"`等。

匹配非数字

用`\d`可以匹配一个数字，而`\D`则匹配一个非数字。例如，`00\D`可以匹配：

- `"00A"`，因为`\D`可以匹配非数字字符`A`；
- `"00#"`，因为`\D`可以匹配非数字字符`#`。

`00\d`可以匹配的字符串`"007"`，`"008"`等，`00\D`是不能匹配的。

类似的，`\W`可以匹配`\w`不能匹配的字符，`\S`可以匹配`\s`不能匹配的字符，这几个正好是反着来的。

```
public class Main {
    public static void main(String[] args) {
        String re1 = "java\\d"; // 对应的正则是java\d
        System.out.println("java9".matches(re1));
        System.out.println("java10".matches(re1));
        System.out.println("javac".matches(re1));

        String re2 = "java\\D";
        System.out.println("javax".matches(re2));
        System.out.println("java#".matches(re2));
        System.out.println("java5".matches(re2));
    }
}
```

重复匹配

我们用`\d`可以匹配一个数字，例如，`A\d`可以匹配`"A0"`，`"A1"`，如果要匹配多个数字，比如`"A380"`，怎么办？

修饰符`*`可以匹配任意个字符，包括0个字符。我们用`A\d*`可以匹配：

- `A`：因为`\d*`可以匹配0个数字；
- `A0`：因为`\d*`可以匹配1个数字`0`；
- `A380`：因为`\d*`可以匹配多个数字`380`。

修饰符`+`可以匹配至少一个字符。我们用`A\d+`可以匹配：

- `A0`：因为`\d+`可以匹配1个数字`0`；
- `A380`：因为`\d+`可以匹配多个数字`380`。

但它无法匹配`"A"`，因为修饰符`+`要求至少一个字符。

修饰符`?`可以匹配0个或一个字符。我们用`A\d?`可以匹配：

- `A`：因为`\d?`可以匹配0个数字；
- `A0`：因为`\d+`可以匹配1个数字`0`。

但它无法匹配`"A33"`，因为修饰符`?`超过1个字符就不能匹配了。

如果我们想精确指定n个字符怎么办？用修饰符`{n}`就可以。`A\d{3}`可以精确匹配：

- `A380`：因为`\d{3}`可以匹配3个数字`380`。

如果我们想指定匹配n~m个字符怎么办？用修饰符`{n,m}`就可以。`A\d{3,5}`可以精确匹配：

- `A380`：因为`\d{3,5}`可以匹配3个数字`380`；
- `A3800`：因为`\d{3,5}`可以匹配4个数字`3800`；
- `A38000`：因为`\d{3,5}`可以匹配5个数字`38000`。

如果没有上限，那么修饰符`{n,}`就可以匹配至少n个字符。

练习

请编写一个正则表达式匹配国内的电话号码规则：$3_{4位区号加7}8$位电话，中间用`-`连接，例如：`010-12345678`。

```
import java.util.*;

public class Main {
    public static void main(String[] args) throws
Exception {
        String re = "\\d";
        for (String s : List.of("010-12345678", "020-
9999999", "0755-7654321")) {
            if (!s.matches(re)) {
                System.out.println("测试失败: " + s);
                return;
            }
        }
        for (String s : List.of("010 12345678", "A20-
9999999", "0755-7654.321")) {
            if (s.matches(re)) {
                System.out.println("测试失败: " + s);
                return;
            }
        }
        System.out.println("测试成功!");
    }
}
```

下载练习：电话匹配练习 （推荐使用IDE练习插件快速下载）

进阶：国内区号必须以0开头，而电话号码不能以0开头，试修改正则表达式，使之能更精确地匹配。

提示：`\d`和`\D`这种简单的规则暂时做不到，我们需要更复杂规则，后面会详细讲解。

小结

单个字符的匹配规则如下：

正则表达式	规则	可以匹配
`A`	指定字符	`A`
`\u548c`	指定Unicode字符	`和`
`.`	任意字符	`a`，`b`，`&`，`0`
`\d`	数字0~9	`0~9`
`\w`	大小写字母，数字和下划线	`a`~`z`，`A`~`Z`，`0~9`，`_`
`\s`	空格、Tab键	空格，Tab
`\D`	非数字	`a`，`A`，`&`，`_`，……
`\W`	非\w	`&`，`@`，`中`，……
`\S`	非\s	`a`，`A`，`&`，`_`，……

多个字符的匹配规则如下：

正则表达式	规则	可以匹配
`A*`	任意个数字符	空，`A`，`AA`，`AAA`，……
`A+`	至少1个字符	`A`，`AA`，`AAA`，……
`A?`	0个或1个字符	空，`A`
`A{3}`	指定个数字符	`AAA`
`A{2,3}`	指定范围个数字符	`AA`，`AAA`
`A{2,}`	至少n个字符	`AA`，`AAA`，`AAAA`，……
`A{0,3}`	最多n个字符	空，`A`，`AA`，`AAA`

复杂匹配规则

匹配开头和结尾

用正则表达式进行多行匹配时，我们用`^`表示开头，`$`表示结尾。例如，`^A\d{3}$`，可以匹配`"A001"`、`"A380"`。

匹配指定范围

如果我们规定一个7~8位数字的电话号码不能以`0`开头，应该怎么写匹配规则呢？`\d{7,8}`是不行的，因为第一个`\d`可以匹配到`0`。

使用`[...]`可以匹配范围内的字符，例如，`[123456789]`可以匹配`1`~`9`，这样就可以写出上述电话号码的规则：`[123456789]\d{6,7}`。

把所有字符全列出来太麻烦，`[...]`还有一种写法，直接写`[1-9]`就可以。

要匹配大小写不限的十六进制数，比如`1A2b3c`，我们可以这样写：`[0-9a-fA-F]`，它表示一共可以匹配以下任意范围的字符：

- `0-9`：字符`0`~`9`；
- `a-f`：字符`a`~`f`；
- `A-F`：字符`A`~`F`。

如果要匹配6位十六进制数，前面讲过的`{n}`仍然可以继续配合使用：`[0-9a-fA-F]{6}`。

`[...]`还有一种排除法，即不包含指定范围的字符。假设我们要匹配任意字符，但不包括数字，可以写`[^1-9]{3}`：

- 可以匹配`"ABC"`，因为不包含字符`1~9`；
- 可以匹配`"A00"`，因为不包含字符`1~9`；
- 不能匹配`"A01"`，因为包含字符`1`；
- 不能匹配`"A05"`，因为包含字符`5`。

或规则匹配

用`|`连接的两个正则规则是*或*规则，例如，`AB|CD`表示可以匹配`AB`或`CD`。

我们来看这个正则表达式`java|php`：

```
public class Main {
    public static void main(String[] args) {
        String re = "java|php";
        System.out.println("java".matches(re));
        System.out.println("php".matches(re));
        System.out.println("go".matches(re));
    }
}
```

它可以匹配`"java"`或`"php"`，但无法匹配`"go"`。

要把`go`也加进来匹配，可以改写为`java|php|go`。

使用括号

现在我们想要匹配字符串`learn java`、`learn php`和`learn go`怎么办？一个最简单的规则是`learn\sjava|learn\sphp|learn\sgo`，但是这个规则太复杂了，可以把公共部分提出来，然后用`(...)`把子规则括起来表示成`learn\\s(java|php|go)`。

```
public class Main {
    public static void main(String[] args) {
        String re = "learn\\s(java|php|go)";
        System.out.println("learn java".matches(re));
        System.out.println("learn Java".matches(re));
        System.out.println("learn php".matches(re));
        System.out.println("learn Go".matches(re));
    }
}
```

上面的规则仍然不能匹配`learn Java`、`learn Go`这样的字符串。试修改正则，使之能匹配大写字母开头的`learn Java`、`learn Php`、`learn Go`。

小结

复杂匹配规则主要有：

正则表达式	规则	可以匹配
^	开头	字符串开头
$	结尾	字符串结束
[ABC]	[...]内任意字符	A，B，C
[A-F0-9xy]	指定范围的字符	`A`，……，`F`，`0`，……，`9`，`x`，`y`
[^A-F]	指定范围外的任意字符	非`A`~`F`
AB|CD|EF	AB或CD或EF	`AB`，`CD`，`EF`

分组匹配

我们前面讲到的`(...)`可以用来把一个子规则括起来，这样写`learn\s(java|php|go)`就可以更方便地匹配长字符串了。

实际上`(...)`还有一个重要作用，就是分组匹配。

我们来看一下如何用正则匹配`区号-电话号`码这个规则。利用前面讲到的匹配规则，写出来很容易：

```
\d{3,4}\-\d{6,8}
```

虽然这个正则匹配规则很简单，但是往往匹配成功后，下一步是提取区号和电话号码，分别存入数据库。于是问题来了：如何提取匹配的子串？

当然可以用`String`提供的`indexOf()`和`substring()`这些方法，但它们从正则匹配的字符串中提取子串没有通用性，下一次要提取`learn\s(java|php)`还得改代码。

正确的方法是用`(...)`先把要提取的规则分组，把上述正则表达式变为`(\d{3,4})\-(\d{6,8})`。

现在问题又来了：匹配后，如何按括号提取子串？

现在我们没办法用`String.matches()`这样简单的判断方法了，必须引入`java.util.regex`包，用`Pattern`对象匹配，匹配后获得一个`Matcher`对象，如果匹配成功，就可以直接从`Matcher.group(index)`返回子串：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("(\\d{3,4})\\-
(\\d{7,8})");
        Matcher m = p.matcher("010-12345678");
        if (m.matches()) {
            String g1 = m.group(1);
            String g2 = m.group(2);
            System.out.println(g1);
```

```
            System.out.println(g2);
        } else {
            System.out.println("匹配失败!");
        }
    }
}
```

运行上述代码，会得到两个匹配上的子串`010`和`12345678`。

要特别注意，`Matcher.group(index)`方法的参数用1表示第一个子串，2表示第二个子串。如果我们传入0会得到什么呢？答案是`010-12345678`，即整个正则匹配到的字符串。

Pattern

我们在前面的代码中用到的正则表达式代码是`String.matches()`方法，而我们在分组提取的代码中用的是`java.util.regex`包里面的`Pattern`类和`Matcher`类。实际上这两种代码本质上是一样的，因为`String.matches()`方法内部调用的就是`Pattern`和`Matcher`类的方法。

但是反复使用`String.matches()`对同一个正则表达式进行多次匹配效率较低，因为每次都会创建出一样的`Pattern`对象。完全可以先创建出一个`Pattern`对象，然后反复使用，就可以实现编译一次，多次匹配：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d{3,4})\\-
(\\d{7,8})");
        pattern.matcher("010-12345678").matches(); // true
        pattern.matcher("021-123456").matches(); // true
        pattern.matcher("022#1234567").matches(); // false
        // 获得Matcher对象:
        Matcher matcher = pattern.matcher("010-12345678");
        if (matcher.matches()) {
            String whole = matcher.group(0); // "010-
12345678", 0表示匹配的整个字符串
            String area = matcher.group(1); // "010", 1表
示匹配的第1个子串
            String tel = matcher.group(2); // "12345678",
2表示匹配的第2个子串
            System.out.println(area);
            System.out.println(tel);
        }
    }
}
```

使用`Matcher`时，必须首先调用`matches()`判断是否匹配成功，匹配成功后，才能调用`group()`提取子串。

利用提取子串的功能，我们轻松获得了区号和号码两部分。

练习

利用分组匹配，从字符串`"23:01:59"`提取时、分、秒。

下载练习：分组匹配 （推荐使用IDE练习插件快速下载）

小结

正则表达式用`(...)`分组可以通过`Matcher`对象快速提取子串：

- `group(0)`表示匹配的整个字符串；
- `group(1)`表示第1个子串，`group(2)`表示第2个子串，以此类推。

非贪婪匹配

在介绍非贪婪匹配前，我们先看一个简单的问题：

给定一个字符串表示的数字，判断该数字末尾`0`的个数。例如：

- `"123000"`：3个`0`
- `"10100"`：2个`0`
- `"1001"`：0个`0`

可以很容易地写出该正则表达式：`(\d+)(0*)`，Java代码如下：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d+)(0*)");
        Matcher matcher = pattern.matcher("1230000");
        if (matcher.matches()) {
            System.out.println("group1=" +
matcher.group(1)); // "1230000"
            System.out.println("group2=" +
matcher.group(2)); // ""
        }
    }
}
```

然而打印的第二个子串是空字符串`""`。

实际上，我们期望分组匹配结果是：

INPUT	\D+	0*
123000	"123"	"000"
10100	"101"	"00"
1001	"1001"	""

但实际的分组匹配结果是这样的：

INPUT	\D+	0*
123000	"123000"	""
10100	"10100"	""
1001	"1001"	""

仔细观察上述实际匹配结果，实际上它是完全合理的，因为`\d+`确实可以匹配后面任意个`0`。

这是因为正则表达式默认使用贪婪匹配：任何一个规则，它总是尽可能多地向后匹配，因此，`\d+`总是会把后面的`0`包含进来。

要让`\d+`尽量少匹配，让`0*`尽量多匹配，我们就必须让`\d+`使用非贪婪匹配。在规则`\d+`后面加个`?`即可表示非贪婪匹配。我们改写正则表达式如下：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d+?)(0*)");
        Matcher matcher = pattern.matcher("1230000");
        if (matcher.matches()) {
            System.out.println("group1=" +
matcher.group(1)); // "123"
            System.out.println("group2=" +
matcher.group(2)); // "0000"
        }
    }
}
```

因此，给定一个匹配规则，加上`?`后就变成了非贪婪匹配。

我们再来看这个正则表达式`(\d??)(9*)`，注意`\d?`表示匹配0个或1个数字，后面第二个`?`表示非贪婪匹配，因此，给定字符串`"9999"`，匹配到的两个子串分别是`""`和`"9999"`，因为对于`\d?`来说，可以匹配1个`9`，也可以匹配0个`9`，但是因为后面的`?`表示非贪婪匹配，它就会尽可能少的匹配，结果是匹配了0个`9`。

小结

- 正则表达式匹配默认使用贪婪匹配，可以使用`?`表示对某一规则进行非贪婪匹配。
- 注意区分`?`的含义：`\d??`。

搜索和替换

分割字符串

使用正则表达式分割字符串可以实现更加灵活的功能。`String.split()`方法传入的正是正则表达式。我们来看下面的代码：

```
"a b c".split("\\s"); // { "a", "b", "c" }
"a b  c".split("\\s"); // { "a", "b", "", "c" }
"a, b ;; c".split("[\\,\\;\\s]+"); // { "a", "b", "c" }
```

如果我们想让用户输入一组标签，然后把标签提取出来，因为用户的输入往往是不规范的，这时，使用合适的正则表达式，就可以消除多个空格、混合`,`和`;`这些不规范的输入，直接提取出规范的字符串。

搜索字符串

使用正则表达式还可以搜索字符串，我们来看例子：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        String s = "the quick brown fox jumps over the
lazy dog.";
        Pattern p = Pattern.compile("\\wo\\w");
        Matcher m = p.matcher(s);
        while (m.find()) {
            String sub = s.substring(m.start(), m.end());
            System.out.println(sub);
        }
    }
}
```

我们获取到`Matcher`对象后，不需要调用`matches()`方法（因为匹配整个串肯定返回false），而是反复调用`find()`方法，在整个串中搜索能匹配上`\\wo\\w`规则的子串，并打印出来。这种方式比`String.indexOf()`要灵活得多，因为我们搜索的规则是3个字符：中间必须是`o`，前后两个必须是字符`[A-Za-z0-9_]`。

替换字符串

使用正则表达式替换字符串可以直接调用`String.replaceAll()`，它的第一个参数是正则表达式，第二个参数是待替换的字符串。我们还是来看例子：

```
public class Main {
    public static void main(String[] args) {
        String s = "The     quick\t\t brown   fox  jumps
over the  lazy dog.";
        String r = s.replaceAll("\\s+", " ");
        System.out.println(r); // "The quick brown fox
jumps over the lazy dog."
    }
}
```

上面的代码把不规范的连续空格分隔的句子变成了规范的句子。可见，灵活使用正则表达式可以大大降低代码量。

反向引用

如果我们要把搜索到的指定字符串按规则替换，比如前后各加一个`xxxx`，这个时候，使用`replaceAll()`的时候，我们传入的第二个参数可以使用`$1`、`$2`来反向引用匹配到的子串。例如：

```
public class Main {
    public static void main(String[] args) {
        String s = "the quick brown fox jumps over the
lazy dog.";
        String r = s.replaceAll("\\s([a-z]{4})\\s", "
<b>$1</b> ");
        System.out.println(r);
    }
}
```

上述代码的运行结果是：

```
the quick brown fox jumps <b>over</b> the <b>lazy</b> dog.
```

它实际上把任何4字符单词的前后用`xxxx`括起来。实现替换的关键就在于`" $1 "`，它用匹配的分组子串`([a-z]{4})`替换了`$1`。

练习

模板引擎是指，定义一个字符串作为模板：

```
Hello, ${name}! You are learning ${lang}!
```

其中，以`${key}`表示的是变量，也就是将要被替换的内容

当传入一个`Map`给模板后，需要把对应的key替换为Map的value。

例如，传入`Map`为：

```
{
    "name": "Bob",
    "lang": "Java"
}
```

然后，`${name}`被替换为`Map`对应的值"Bob”，`${lang}`被替换为`Map`对应的值"Java"，最终输出的结果为：

```
Hello, Bob! You are learning Java!
```

请编写一个简单的模板引擎，利用正则表达式实现这个功能。

提示：参考Matcher.appendReplacement()方法。

下载练习：模板引擎练习 （推荐使用IDE练习插件快速下载）

小结

使用正则表达式可以：

- 分割字符串：`String.split()`
- 搜索子串：`Matcher.find()`
- 替换字符串：`String.replaceAll()`