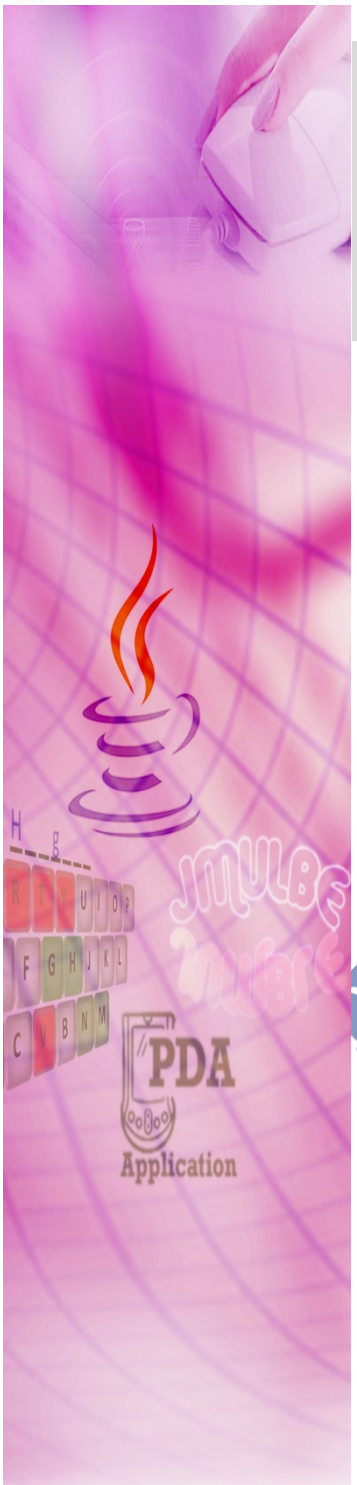


第 2 章

使用正则表达式和本地化

有时，您需要开发应用程序以提供需要处理字符串的功能，例如搜索或过滤。要开发此类应用程序，需要编写较长的代码，这是较为耗时的任务。为解决该问题，Java 支持正则表达式，也称为 regex。*正则表达式*允许您将一个字符序列与另一字符串匹配。

有时，您要开发可全球使用的应用程序。这些应用程序应能够根据部署区域显示各种信息，例如文本、货币和日期。为满足这一需求，Java 支持本地化。

在本章中，您将学习使用正则表达式处理字符串和实现本地化。

目标

在本章中，您将学习：

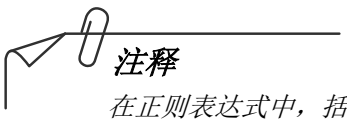
- 使用正则表达式处理字符串
- 实现本地化


NIIT
training
china
.com

使用正则表达式处理字符串

思考场景：需要开发一个 Java 应用程序以提供在特定文件中搜索文本 hello 的功能。为完成该任务，Java 提供了对正则表达式的支持。正则表达式是定义字符序列的字符串，称为模式。该模式可与另一字符串匹配。为使用正则表达式，Java 在 `java.util.regex` 包中提供了两个类 `Pattern` 和 `Matcher`。

此外，Java 还提供了字符类以处理字符组。此外，为确定字符中模式的出现次数，Java 还支持限定符。



注释

在正则表达式中，括在方括号中的一组字符称为字符类。

使用 Pattern 和 Matcher 类

`Pattern` 类表示编译的正则表达式。由于 `Pattern` 类不定义任何构造函数，因此 `Pattern` 类引用通过使用 `Pattern` 类的静态方法 `compile()` 获取。

以下代码段用于创建 `Pattern` 类的引用：

```
Pattern myPattern = Pattern.compile("Expression");
```

在上述代码段中，正则表达式 `Expression` 传递到返回 `Pattern` 类的对象的 `compile()` 方法。然后，`Pattern` 对象用于创建 `Matcher` 对象，如以下代码段所示：

```
Matcher myMatcher = myPattern.matcher("Expression");
```

为比较这两个表达式，`Matcher` 类提供 `matches()` 方法。`matches()` 方法返回值 `true` 的条件是 `matcher` 对象中的表达式与 `pattern` 对象中指定的模式相匹配。否则它将返回值 `false`。以下代码段显示了 `matches()` 方法的用法：

```
boolean myBoolean = myMatcher.matches();
```

下表列出了 Pattern 类的最常用方法。

方法	*描述*	*示例*

`String pattern()`	*用于返回编译的正则表达式。*	`String s1 = "pattern";`
		`Pattern myPattern = Pattern.compile(s1);`
		`String s2 = myPattern.pattern();`
		`System.out.println(s2);`
		输出:
		`pattern`

`String[] split(CharSequence input, int limit)`	*用于根据模式和限制拆分指定输入序列。*	`Pattern pattern = Pattern.compile(":");`
		`String[] split = pattern.split("One:two:three", 2);`
		`for (String element : split)`
		`{`
		`System.out.println("element = " + element);`
		`}`
		输出:
		`element = One`
		`element = two:three`

`static boolean matches(String regex,CharSequence input)`	*用于编译指定正则表达式并尝试根据它来匹配指定输入。*	`boolean matches = Pattern.matches("is", "is");`
		`System.out.println("matches = " + matches);`
		输出:
		`matches = true`

Pattern 类的方法

下表列出了 Matcher 类的最常用方法。

方法	描述	示例

Matcher appendReplacement(StringBuffer sb, String replacement)	用于处理通过向 StringBuffer 变量添加输入字符序列处理 matcher 中指定的输入字符序列。如果在 Matcher 中找到了 Pattern 中指定的匹配，那么它将替换为替换字符串并追加到 StringBuffer 变量。然后，剩余字符序列将被截断。	Pattern pattern= Pattern.compile("John");
		Matcher matcher= pattern.matcher("John does this, and John does that");
		StringBuffer s3 = new StringBuffer();
		while(matcher.find())
		{
		matcher.appendReplacement(s3, "sam");
		System.out.println(s3.toString());
		}
		输出:
		sam
		sam does this, and sam

StringBuffer appendTail(StringBuffer sb)	用于处理通过向 StringBuffer 变量添加输入字符序列处理 Matcher 中指定的输入字符序列。如果在 Matcher 中找到了 Pattern 中指定的匹配，那么它将替换为替换字符串并追加到 StringBuffer 变量。然后，将追加剩余字符序列。	Pattern pattern= Pattern.compile("John");
		Matcher matcher= pattern.matcher("John does this, and John does that");
		StringBuffer s3 = new StringBuffer();
		while(matcher.find())
		{
		matcher.appendReplacement(s3, "sam");
		System.out.println(s3.toString());
		}
		matcher.appendTail(s3);
		System.out.println(s3.toString());
		输出:
		sam
		sam does this, and sam
		sam does this, and sam

方法	描述	示例
String replaceAll(String replacement)	用于替换与带有指定替换字符串的模式匹配的输入序列的每个子序列。	Pattern pattern= Pattern.compile("John"); Matcher matcher= pattern.matcher("John does this, and John does that"); String s2 = matcher.replaceAll("sam"); System.out.println("replaceAll = " + s2); 输出: replaceAll = sam does this, and sam does that
String replaceFirst(String replacement)	用于替换与带有指定替换字符串的模式匹配的输入序列的第一个子序列。	Pattern pattern= Pattern.compile("John"); Matcher matcher= pattern.matcher("John does this, and John does that"); String s2 = matcher.replaceFirst("sam"); System.out.println("replaceFirst = " + s2); 输出: replaceFirst = sam does this, and John does that

int start(int group)	用于返回指定组在匹配操作期间捕获的子序列的开始索引。	Pattern pattern = Pattern.compile("a");
		Matcher matcher = pattern.matcher("This is a text");
		while(matcher.find())
		{
		System.out.println("Match started at:"+ matcher.start(0));
		}
		输出:
		Match started at:8

方法	*描述*	*示例*

`int end(int group)`	*用于返回指定组在匹配操作期间捕获子序列的最后一个字符之后的偏移量。*	`Pattern pattern = Pattern.compile("a");`
		`Matcher matcher = pattern.matcher("This is a text");`
		`while(matcher.find())`
		`{`
		`System.out.println("Match ended at:"+ matcher.end(0));`
		`}`
		输出:
		`Match ended at:9`

Matcher 类的方法

使用字符类

思考场景：需要在文件中搜索可能具有前缀“a”、“b”或“c”以及后缀“at”（例如 bat 和 cat）的特定词汇。为此，正则表达式提供字符类。字符类指定将成功匹配指定输入字符串中的单个字符的字符。

下表列出了字符类的构造。

构造	*描述*
`[def]`	*如果给定输入字符串以字符 d、e 或 f 开头，则匹配成功。*
`[^def]`	*如果给定输入字符串以 d、e 或 f 之外的任意字符开头（逻辑非），则匹配成功。*
`[a-zA-Z]`	*如果给定输入字符串以 a 到 z 或 A 到 Z（含端点）之间的任意字符开头（范围），则匹配成功。*
`[b-e[n-q]]`	*如果给定输入字符串以 b 到 e 或 n 到 q 之间的任意字符开头，则匹配成功。*
`[a-z&&[abc]]`	*如果给定输入字符串以字符 a、b 或 c（交集）开头，则匹配成功。*
`[a-z&&[^bcd]]`	*如果给定输入字符串以 a 到 z 中除 b、c 和 d 之外的任意字符开头，即 [ae-z]（减法），则匹配成功。*

构造	*描述*
[a-z&&[^n-p]]	*如果给定输入字符串以 a 到 z 中除 n 到 p 之外的任意字符开头，即 [a-mq-z]（减法），则匹配成功。*

字符类的构造

思考以下代码，它演示字符类的用法：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class TestRegx
{
public static void main(String[] args)
{
Pattern myPattern = Pattern.compile("[abc]at");
Matcher myMatcher = myPattern.matcher("bat");
boolean myBoolean = myMatcher.matches();
if(myBoolean)
System.out.println("Expression Matched");
else
System.out.println("Expression Not Matched");
}
}
```

执行上述代码时，会显示以下输出：

```
Expression Matched
```

在上述代码中，只有当第一个字母与字符类定义的其中一个字符匹配时，`myBoolean` 才会具有值 `true`。

使用计数器

思考场景：您需要指定一个或一串字符在表达式中出现的次数。例如，您要确定字符 Ro 在表达式 “When in Rome, do as the Romans”中出现的次数。要完成该任务，正则表达式提供了限定符帮助您指定与指定字符串匹配的字符串的出现次数。此外，限定符使您能够轻松选择文件中的一系列字符。限定符具有以下三种类型：

- **Greedy**：用于匹配与模式匹配的最长的字符串。
- **Reluctant**：用于匹配与模式匹配的最短的字符串。
- **Possessive**：用于将正则表达式与整个字符串匹配。仅在整个字符串满足条件时匹配。

下表列出了不同类型的限定符。

Greedy	*Reluctant*	*Possessive*	*描述*
X?	*X??*	*X?+*	*X，一次或完全没有*
*X**	*X*?*	*X*+*	*X，0 或多次*
X+	*X+?*	*X++*	*X，一次或多次*
X{n}	*X{n}?*	*X{n}+*	*X，正好 n 次*
X{n,}	*X{n,}?*	*X{n,}+*	*X，至少 n 次*
X{n,m}	*X{n,m}?*	*X{n,m}+*	*X，至少 n 次但不超过 m 次*

限定符的类型

思考以下代码段，它测试表达式 When in Rome, do as the Romans 中以 Ro 开头的单词的出现次数：

```
 String text="When in Rome, do as the Romans ";
String textSplit[]=text.split(" ");
Pattern myPattern=Pattern.compile("Ro.+");
for(int i=0;i<textSplit.length;i++)
{
   Matcher myMatcher=myPattern.matcher(textSplit[i]);
   boolean myBoolean =myMatcher.matches();
   System.out.println(myBoolean);
}
```

在上述代码段中，`split()` 方法拆分表达式“`When in Rome, do as the Romans`”。之后，模式 `Ro` 将与每个拆分表达式比较。如果单词的前两个字符是“`Ro`”，那么 `System.out.println(myBoolean);` 语句将打印值 `true`。

思考接受用户的模式和匹配器的以下代码。

```
import java.util.regex.*;
import java.util.*;
public class PatternMethods
{
public static void main(String args[])
   {
Scanner input = new Scanner(System.in);
System.out.println("Enter the desired pattern:");
String pattern = input.nextLine();
System.out.println("Enter the text:");
String matcher = input.nextLine();
Pattern myPattern2 = Pattern.compile(pattern);
Matcher myMatcher2 = myPattern2.matcher(matcher);
```

```
Boolean myBoolean2 = myMatcher2.matches();
boolean b = myBoolean2;
if (b == true)
{

System.out.println("I found the text:"+ myMatcher2.group() + "\n"
+ "Starting at:"+ myMatcher2.start() + "\n"
+ "Ending at index:"+ myMatcher2.end());

}
else if (b == false)
{
System.out.println("No match found");
}

   }
}
```

在执行上述代码时，如果提供模式 `.*Friday` 和匹配器 `XFriday`，那么将显示以下输出：

```
I found the text:XFriday
Starting at:0
Ending at index:7
```

生成上述输出是因为模式使用的是 greedy 限定符。由于限定符是 greedy，因此一开始便使用整个输入字符串。由于整个输入字符串与模式不匹配，因此匹配器一次一个字母地往回遍历，直到找到最右侧出现的“`Friday`”。此时，匹配成功，搜索结束。

在执行上述代码时，如果提供模式 `.*?Friday` 和匹配器 `XFriday`，那么将显示以下输出：

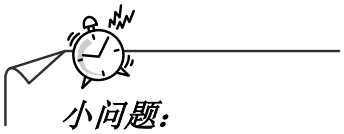
```
I found the text:XFriday
Starting at:0
Ending at index:7
```

生成上述输出是因为模式使用的是 reluctant 限定符。由于限定符是 reluctant，因此一开始不使用输入字符串的任何字符。之后，将逐个使用字符串的各个字符。

在执行上述代码时，如果提供模式 `.*+Friday` 和匹配器 `XFriday`，那么将显示以下输出：

```
No match found
```

生成上述输出是因为模式使用的是 possessive 限定符。由于限定符是 possessive，因此一开始便使用整个输入字符串。由于该限定符不会往回遍历，因此将无法找到匹配。



小问题：

以下哪种方法用于编译给定正则表达式并尝试 将给定输入与指定模式匹配？

1. `static boolean matches(String regex,CharSequence input)`
2. `Pattern pattern()`
3. `static Pattern compile(String regex,int flags)`
4. `Matcher appendReplacement(StringBuffer sb,String replacement)`

答案：

1. `static boolean matches(String regex,CharSequence input)`

活动 2.1：使用正则表达式处理字符串

实现本地化

思考场景：您要开发可全球使用的酒店管理应用程序。您希望应用程序中的文本应根据区域显示。例如，如果在法国使用该应用程序，文本应显示为法语，货币应显示为欧元。要发开此类应用程序，您需要实现本地化，这是将应用程序定制为特定语言环境和文化的过程。可对不同类型数据实现本地化，例如日期、货币和文本。要本地化不同类型的数据，需要确定语言和国家。要确定语言，Java 提供了预定义的语言代码集，例如 zh 代表中文，en 代表英文。此外，还提供了预定义的国家代码集，例如 AU 代表澳大利亚，CN 代表中国。要使用本地化，还需要使用 java.util 包的 Locale 类。可以创建 Locale 类的实例，如以下代码段所示：

```
Locale l=new Locale("de","DE");
```

在上述代码段中，de 指定语言代码，DE 指定国家代码。

本地化日期

要本地化日期，您需要使用各种日期格式。要根据语言环境确定日期格式，可以使用 java.text.DateFormat 类。要显示德语日期，使用以下代码：

```
import java.text.DateFormat;
import java.util.*;
public class DateDemo {
public static void main(String args[]){
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, new
Locale("de","DE"));
String date = df.format(new Date());
System.out.println(date);
}
}
```

上述代码生成以下输出：

```
4.Juni 2013
```

注释

上述输出因当前系统日期而异。

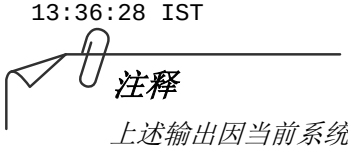
在上述代码中，DateFormat.getDateInstance 用于指定德语语言环境的格式化样式。DateFormat.Long 根据语言环境指定日期的描述。String date = df.format(new Date()); 语句将当前日期格式化为日期字符串。

同样的，要显示德语时间，使用以下代码：

```
import java.text.*;
import java.util.*;
public class TimeDemo {
public static void main(String args[]){
DateFormat df = DateFormat.getTimeInstance(DateFormat.LONG, new
Locale("de","DE"));
String time = df.format(new Date());
System.out.println(time);
}
}
```

上述代码生成以下输出：

```
13:36:28 IST
```



注释

上述输出因当前系统时间而异。

在上述代码中，`getTimeInstance()` 方法用于指定德语语言环境的格式化样式。`DateFormat.Long` 根据语言环境指定日期的描述。`String time = df.format(new Date());` 语句将当前时间格式化为字符串格式。

本地化货币

要本地化货币，需要使用各种货币格式。要根据语言环境确定货币格式，您可以使用 `java.text.NumberFormat` 类。要显示德语货币，使用以下代码：

```
import java.text.*;
import java.util.*;
public class CurrencyDemo {
    public static void main(String args[]){
        NumberFormat nft = NumberFormat.getCurrencyInstance(new
Locale("de","DE"));
        String formatted = nft.format(1000000);
        System.out.println(formatted);
    }
}
```

上述代码生成以下输出：

```
1.000.000,00 €
```

在上述代码中，`NumberFormat.getCurrencyinstance` 返回该语言环境的货币格式。`String formatted = nft.format(1000000);` 语句将货币格式化字符串。

本地化文本

要本地化文本，您需要使用资源束。资源束是包含特定于语言环境的数据的属性文件。要使用资源束中存储的数据，需要使用 `ResourceBundle` 类。`ResourceBundle` 是封装一组资源的类。要访问特定资源，需要获取 `ResourceBundle` 类的引用并调用其 `getObject()` 方法。

要具有特定于语言环境的对象，程序需要使用 `getBundle` 方法装入 `ResourceBundle` 对象，如以下代码段所示：

```
ResourceBundle myResourceBundle = ResourceBundle.getBundle("MessagesBundle");
```

在上述代码段中，加载了包含本地化资源的 `MessagesBundle` 资源束。 调用 `getBundle()` 方法时，`ResourceBundle` 尝试加载具有指定的名称变体的资源文件。它搜索名称中附加了显式指定的语言环境值的资源束。然后搜索具有缺省语言环境值的资源束。例如，如果默认语言环境是 `Locale.US`，那么 `getBundle()` 将加载美国语言环境的资源束文件。

思考场景：您需要开发一个 Java 应用程序以显示德语或中文的消息 Welcome。为此，您需要创建德语和中文资源束。

您可以创建德语资源束文件，例如 `MessageBundle_de.properties`，并在文件中添加以下代码段将消息 Welcome 本地化为德语：

```
message=willkommen
```

您可以创建中文资源束文件，例如 `MessageBundle_de.properties`，并在文件中添加以下代码段将消息 Welcome 本地化为中文：

```
message=歡迎
```

注释

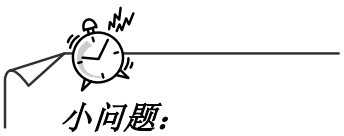
要创建特定于语言的资源束文件，需要指定属性文件名 `<resourcebundle name>_<language code>.properties`。

以下代码演示如何本地化文本：

```
import java.util.Locale;
import java.util.ResourceBundle;
public class TestLocale {
public static void main(String args[]){
Locale l1=new Locale("de","DE");
ResourceBundle rb1 = ResourceBundle.getBundle("MessageBundle",l1);
System.out.println(rb1.getString("message"));
Locale l2=new Locale("zn","ZN");
ResourceBundle rb2 = ResourceBundle.getBundle("MessageBundle",l2);
System.out.println(rb2.getString("message"));
}
```

```
}
```

在上述代码中，`getBundle()` 方法用于在程序需要特定于语言环境的对象时加载 `ResourceBundle` 类。要从资源束检索对象，使用 `getString()` 方法。



小问题：

以下哪个语句表示创建 `Locale` 类的实例？

1. `Locale l=new Locale(“de”, “DE”);`
2. `Locale l=new Locale(“DE”, “de”);`
3. `Locale l=new Locale(‘de’, ‘DE’);`
4. `Locale l=new Locale(‘DE’, ‘de’);`

答案：

1. `Locale l=new Locale(“de”, “DE”);`

活动 2.2：实现本地化

□ □ □ □

1. 说明以下语句是正确还是错误。

   本地化是将应用程序定制为特定语言环境和文化的过程。

2. 以下哪个模式类方法用于返回用于编译模式的正则表达式？

   a. `String pattern()`
   b. `static boolean matches(String regex,CharSequence input)`
   c. `static Pattern compile (String regex)`

3. 以下哪个限定符用于匹配与模式匹配的最短字符串？

   a. Greedy
   b. Reluctant
   c. Possessive

4. 如果要求给定输入字符串以 a 到 z 中除 b、c 和 d 之外的任意字符开头即匹配成功，请确定应使用以下哪个选项。

   a. `[a-z&&[abc]]`
   b. `[b-z&&[^bcd]]`
   c. `[a-z&&[^n-p]]`
   d. `[abc]`

5. 确定用于将模式与最长字符串匹配的正确选项。

   a. Greedy
   b. Reluctant
   c. Possessive

小结

在本章中，您学习了：

- 正则表达式允许您将一个字符序列与另一字符串匹配。
- 正则表达式是定义字符序列的字符串，称为模式。
- 为使用正则表达式，Java 在 `java.util.regex` 包中提供了两个类 `Pattern` 和 `Matcher`。
- Java 提供了字符类以处理字符组。
- 为确定字符中模式的出现次数，Java 还支持限定符。
- `Pattern` 类表示编译的正则表达式。
- 由于 `Pattern` 类不定义任何构造函数，因此 `Pattern` 类引用通过使用 `Pattern` 类的静态方法 `compile()` 获取。
- 字符类指定将成功匹配指定输入字符串中的单个字符的字符。
- 正则表达式提供了限定符帮助您指定与指定字符串匹配的字符串的出现次数。
- 限定符具有以下三种类型：
  - Greedy
  - Reluctant
  - Possessive
- 本地化是将应用程序定制为特定语言环境和文化的过程。
- 可对不同类型数据实现本地化，例如日期、货币和文本。
- 要本地化不同类型的数据，需要确定语言和国家。
- 要确定语言，Java 提供了预定义的语言代码集，例如 `zh` 代表中文，`en` 代表英文。
- 要使用本地化，需要使用 `java.util` 包的 `Locale` 类。
- 要本地化日期，您需要使用各种日期格式。要根据语言环境确定日期格式，可以使用 `java.text.DateFormat` 类。
- 要本地化货币，需要使用各种货币格式。要根据语言环境确定货币格式，您可以使用 `java.text.NumberFormat` 类。
- 要本地化文本，您需要使用资源束。
- 资源束是包含特定于语言环境的数据的属性文件。要使用资源束中存储的数据，需要使用 `ResourceBundle` 类。
- 要具有特定于语言环境的对象，程序需要使用 `getBundle` 方法装入 `ResourceBundle` 对象。