Java正则表达式入门

众所周知，在程序开发中，难免会遇到需要匹配、查找、替换、判断字符串的情况发生，而这些情况有时又比较复杂，如果用纯编码方式解决，往往会浪费程序员的时间及精力。因此，学习及使用正则表达式，便成了解决这一矛盾的主要手段。

大家都知道，正则表达式是一种可以用于模式匹配和替换的规范，一个正则表达式就是由普通的字符（例如字符 a 到 z）以及特殊字符（元字符）组成的文字模式，它用以描述在查找文字主体时待匹配的一个或多个字符串。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。

自从 jdk1.4 推出 java.util.regex 包，就为我们提供了很好的 JAVA 正则表达式应用平台。

因为正则表达式是一个很庞杂的体系，所以我仅例举些入门的概念，更多的请参阅相关书籍及自行摸索。

\\ 反斜杠
\t 间隔 ('\u0009')
\n 换行 ('\u000A')
\r 回车 ('\u000D')
\d 数字 等价于[0-9]
\D 非数字 等价于[^0-9]
\s 空白符号 [\t\n\x0B\f\r]
\S 非空白符号 [^\t\n\x0B\f\r]
\w 单独字符 [a-zA-Z_0-9]
\W 非单独字符 [^a-zA-Z_0-9]
\f 换页符
\e Escape
\b 一个单词的边界
\B 一个非单词的边界
\G 前一个匹配的结束

^为限制开头
^java 条件限制为以 Java 为开头字符
$为限制结尾
java$ 条件限制为以 java 为结尾字符
.为限制一个任意字符
java.. 条件限制为 java 后除换行外任意两个字符

加入特定限制条件「[]」
[a-z] 条件限制在小写 a to z 范围中一个字符
[A-Z] 条件限制在大写 A to Z 范围中一个字符
[a-zA-Z] 条件限制在小写 a to z 或大写 A to Z 范围中一个字符
[0-9] 条件限制在小写 0 to 9 范围中一个字符
[0-9a-z] 条件限制在小写 0 to 9 或 a to z 范围中一个字符
[0-9[a-z]] 条件限制在小写 0 to 9 或 a to z 范围中一个字符(交集)

[]中加入^后加再次限制条件「[^]」
[^a-z]　　条件限制在非小写 a to z 范围中一个字符
[^A-Z]　　条件限制在非大写 A to Z 范围中一个字符
[^a-zA-Z]　条件限制在非小写 a to z 或大写 A to Z 范围中一个字符
[^0-9]　　条件限制在非小写 0 to 9 范围中一个字符
[^0-9a-z]　条件限制在非小写 0 to 9 或 a to z 范围中一个字符
[^0-9[a-z]]　条件限制在非小写 0 to 9 或 a to z 范围中一个字符(交集)

在限制条件为特定字符出现 0 次以上时，可以使用「*」
J*　　0 个以上 J
.*　　0 个以上任意字符
J.*D　　J 与 D 之间 0 个以上任意字符

在限制条件为特定字符出现 1 次以上时，可以使用「+」
J+　　1 个以上 J
.+　　1 个以上任意字符
J.+D　　J 与 D 之间 1 个以上任意字符

在限制条件为特定字符出现有 0 或 1 次以上时，可以使用「?」
JA?　　J 或者 JA 出现

限制为连续出现指定次数字符「{a}」
J{2}　　JJ
J{3}　　JJJ
文字 a 个以上，并且「{a,}」
J{3,}　　JJJ,JJJJ,JJJJJ,???(3 次以上 J 并存)
文字个以上，b 个以下「{a,b}」
J{3,5}　　JJJ 或 JJJJ 或 JJJJJ
两者取一「|」
J|A　　J 或 A
Java|Hello　　Java 或 Hello

「()」中规定一个组合类型
比如，我查询 index 中 <a href> 间的数据，可写作 <a.*href=\".*\">(.+?)

在使用 Pattern.compile 函数时，可以加入控制正则表达式的匹配行为的参数：
Pattern Pattern.compile(String regex, int flag)

flag 的取值范围如下：
Pattern.CANON_EQ　　当且仅当两个字符的"正规分解(canonical decomposition)"都完全相同的情况下，才认定匹配。比如用了这个标志之后，表达式"a\u030A"会匹配"?"。默认情况下，不考虑"规范相等性(canonical equivalence)"。
Pattern.CASE_INSENSITIVE(?i)　　默认情况下，大小写不明感的匹配只适用于 US-ASCII 字符集。

这个标志能让表达式忽略大小写进行匹配。要想对 Unicode 字符进行大小不明感的匹配，只要将 UNICODE_CASE 与这个标志合起来就行了。

Pattern.COMMENTS(?x)　　在这种模式下，匹配时会忽略(正则表达式里的)空格字符(译者注：不是指表达式里的"\\s"，而是指表达式里的空格，tab，回车之类)。注释从#开始，一直到这行结束。可以通过嵌入式的标志来启用 Unix 行模式。

Pattern.DOTALL(?s)　　在这种模式下，表达式'.'可以匹配任意字符，包括表示一行的结束符。默认情况下，表达式'.'不匹配行的结束符。

Pattern.MULTILINE

(?m)　　在这种模式下，'^'和'$'分别匹配一行的开始和结束。此外，'^'仍然匹配字符串的开始，'$'也匹配字符串的结束。默认情况下，这两个表达式仅仅匹配字符串的开始和结束。

Pattern.UNICODE_CASE

(?u)　　在这个模式下，如果你还启用了 CASE_INSENSITIVE 标志，那么它会对 Unicode 字符进行大小写不明感的匹配。默认情况下，大小写不敏感的匹配只适用于 US-ASCII 字符集。

Pattern.UNIX_LINES(?d)　　在这个模式下，只有'\n'才被认作一行的中止，并且与'.'，'^'，以及'$'进行匹配。

抛开空泛的概念，下面写出几个简单的 Java 正则用例：

◆比如，在字符串包含验证时

```
//查找以 Java 开头,任意结尾的字符串
  Pattern pattern = Pattern.compile("^Java.*");
  Matcher matcher = pattern.matcher("Java 不是人");
  boolean b= matcher.matches();
  //当条件满足时，将返回 true，否则返回 false
  System.out.println(b);
```

◆以多条件分割字符串时

```
Pattern pattern = Pattern.compile("[, |]+");
String[] strs = pattern.split("Java Hello World  Java,Hello,,World|Sun");
for (int i=0;i<strs.length;i++) {
    System.out.println(strs[i]);
}
```

◆文字替换（首次出现字符）

```
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello World");
//替换第一个符合正则的数据
System.out.println(matcher.replaceFirst("Java"));
```

◆文字替换（全部）

```
Pattern pattern = Pattern.compile("正则表达式");
```

```
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello World");
//替换第一个符合正则的数据
System.out.println(matcher.replaceAll("Java"));
```

◆文字替换（置换字符）

```
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello World ");
StringBuffer sbr = new StringBuffer();
while (matcher.find()) {
    matcher.appendReplacement(sbr, "Java");
}
matcher.appendTail(sbr);
System.out.println(sbr.toString());
```

◆验证是否为邮箱地址

```
String str="ceponline@yahoo.com.cn";
Pattern pattern =
Pattern.compile("[\\w\\.\\-]+@([\\w\\-]+\\.)+[\\w\\-]+",Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(str);
System.out.println(matcher.matches());
```

◆去除 html 标记

```
Pattern pattern = Pattern.compile("<.+?>", Pattern.DOTALL);
Matcher matcher = pattern.matcher("<a href=\"index.html\">主页</a>");
String string = matcher.replaceAll("");
System.out.println(string);
```

◆查找 html 中对应条件字符串

```
Pattern pattern = Pattern.compile("href=\"(.+?)\"");
Matcher matcher = pattern.matcher("<a href=\"index.html\">主页</a>");
if(matcher.find())
  System.out.println(matcher.group(1));
}
```

◆截取 http://地址

```
//截取 url
Pattern pattern = Pattern.compile("(http://|https://){1}[\\w\\.\\-/:]+");
Matcher matcher = pattern.matcher("dsdsds<http://dsds//gfgffdfd>fdf");
StringBuffer buffer = new StringBuffer();
while(matcher.find()){
    buffer.append(matcher.group());
    buffer.append("\r\n");
```

```
System.out.println(buffer.toString());
}
```

◆替换指定{}中文字

```
String str = "Java 目前的发展史是由{0}年-{1}年";
String[][] object={new String[]{"\\{0\\}","1995"},new String[]{"\\{1\\}","2007"}};
System.out.println(replace(str,object));

public static String replace(final String sourceString,Object[] object) {
          String temp=sourceString;
          for(int i=0;i<object.length;i++){
                  String[] result=(String[])object[i];
             Pattern    pattern = Pattern.compile(result[0]);
             Matcher matcher = pattern.matcher(temp);
             temp=matcher.replaceAll(result[1]);
          }
          return temp;
}
```

◆以正则条件查询指定目录下文件

```
 //用于缓存文件列表
       private ArrayList files = new ArrayList();
       //用于承载文件路径
       private String _path;
       //用于承载未合并的正则公式
       private String _regexp;

       class MyFileFilter implements FileFilter {

            /**
             * 匹配文件名称
             */
            public boolean accept(File file) {
              try {
                Pattern pattern = Pattern.compile(_regexp);
                Matcher match = pattern.matcher(file.getName());
                return match.matches();
              } catch (Exception e) {
                return true;
              }
            }
```

```
    }

/**
 * 解析输入流
 * @param inputs
 */
FilesAnalyze (String path,String regexp){
    getFileName(path,regexp);
}

/**
 * 分析文件名并加入 files
 * @param input
 */
private void getFileName(String path,String regexp) {
    //目录
      _path=path;
      _regexp=regexp;
      File directory = new File(_path);
      File[] filesFile = directory.listFiles(new MyFileFilter());
      if (filesFile == null) return;
      for (int j = 0; j < filesFile.length; j++) {
        files.add(filesFile[j]);
      }
      return;
    }

/**
 * 显示输出信息
 * @param out
 */
public void print (PrintStream out) {
    Iterator elements = files.iterator();
    while (elements.hasNext()) {
        File file=(File) elements.next();
            out.println(file.getPath());
    }
}

public static void output(String path,String regexp) {

    FilesAnalyze fileGroup1 = new FilesAnalyze(path,regexp);
    fileGroup1.print(System.out);
}
```

```
public static void main (String[] args) {
    output("C:\\","[A-z|.]*");
}
```

Java 正则的功用还有很多，事实上只要是字符处理，就没有正则做不到的事情存在。（当然，正则解释时较耗时间就是了|||.....）

JAVA中正则表达式的应用（一）

陈广佳（cgjmail@163.net）
电子信息工程系工科学士
2001 年 12 月

由于工作的需要，本人经常要面对大量的文字电子资料的整理工作，因此曾对在 JAVA 中正则表达式的应用有所关注，并对其有一定的了解，希望通过本文与同行进行有关方面的心得交流。

正则表达式：

正则表达式是一种可以用于模式匹配和替换的强有力的工具，一个正则表达式就是由普通的字符（例如字符 a 到 z）以及特殊字符（称为元字符）组成的文字模式，它描述在查找文字主体时待匹配的一个或多个字符串。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。

正则表达式在字符数据处理中起着非常重要的作用，我们可以用正则表达式完成大部分的数据分析处理工作，如:判断一个串是否是数字、是否是有效的 Email 地址，从海量的文字资料中提取有价值的数据等等，如果不使用正则表达式，那么实现的程序可能会很长，并且容易出错。对这点本人深有体会，面对大量工具书电子档资料的整理工作，如果不懂得应用正则表达式来处理，那么将是很痛苦的一件事情，反之则将可以轻松地完成，获得事半功倍的效果。

由于本文目的是要介绍如何在 JAVA 里运用正则表达式，因此对刚接触正则表达式的读者请参考有关资料，在此因篇幅有限不作介绍。

JAVA 对正则表达式的支持：

在 JDK1.3 或之前的 JDK 版本中并没有包含正则表达式库可供 JAVA 程序员使用，之前我们一般都在使用第三方提供的正则表达式库，这些第三方库中有源代码开放的，也有需付费购买的，而现时在 JDK1.4 的测试版中也已经包含有正则表达式库---java.util.regex。

故此现在我们有很多面向 JAVA 的正则表达式库可供选择，以下我将介绍两个较具代表性的 **Jakarta-ORO** 和 **java.util.regex**，首先当然是本人一直在用的 **Jakarta-ORO**：

Jakarta-ORO 正则表达式库

1．简介：

Jakarta-ORO是最全面以及优化得最好的正则表达式API之一，Jakarta-ORO库以前叫做OROMatcher，是由Daniel F. Savarese编写，后来他将其赠与Jakarta Project，读者可在Apache.org的网站下载该API包。

许多源代码开放的正则表达式库都是支持 Perl5 兼容的正则表达式语法，Jakarta-ORO 正则表达式库也不例外，他与 Perl 5 正则表达式完全兼容。

2．对象与其方法：

★PatternCompiler 对象：

我们在使用 Jakarta-ORO API 包时，最先要做的是，创建一个 Perl5Compiler 类的实例，并把它赋值给 PatternCompiler 接口对象。Perl5Compiler 是 PatternCompiler 接口的一个实现，允许你把正则表达式编译成用来匹配的 Pattern 对象。

```
PatternCompiler compiler=new Perl5Compiler();
```

★Pattern 对象：

要把所对应的正则表达式编译成 Pattern 对象，需要调用 compiler 对象的 compile()方法，并在调用参数中指定正则表达式。举个例子，你可以按照下面这种方式编译正则表达式"s[ahkl]y"：

```
Pattern pattern=null;

      try {

              pattern=compiler.compile("s[ahkl]y ");

      } catch (MalformedPatternException e) {

              e.printStackTrace();

      }
```

在默认的情况下，编译器会创建一个对大小写敏感的模式（pattern）。因此，上面代码编译得到的模式只匹配"say"、"shy"、 "sky"和"sly"，但不匹配"Say"和"skY"。要创建一个大小写不敏感的模式，你应该在调用编译器的时候指定一个额外的参数:

```
pattern=compiler.compile("s[ahkl]y",Perl5Compiler.CASE_INSENSITIVE_MASK);
```

Pattern 对象创建好之后，就可以通过 PatternMatcher 类用该 Pattern 对象进行模式匹配。

★PatternMatcher 对象:

PatternMatcher 对象依据 Pattern 对象和字符串展开匹配检查。你要实例化一个 Perl5Matcher 类并把结果赋值给 PatternMatcher 接口。Perl5Matcher 类是 PatternMatcher 接口的一个实现，它根据 Perl 5 正则表达式语法进行模式匹配：

```
PatternMatcher matcher=new Perl5Matcher();
```

PatternMatcher 对象提供了多个方法进行匹配操作，这些方法的第一个参数都是需要根据正则表达式进行匹配的字符串：

1. boolean matches(String input, Pattern pattern)：当要求输入的字符串 input 和正则表达式 pattern 精确匹配时使用该方法。也就是说当正则表达式完整地描述输入字符串时返回真值。
2. boolean matchesPrefix(String input, Pattern pattern)：要求正则表达式匹配输入字符串起始部分时使用该方法。也就是说当输入字符串的起始部分与正则表达式匹配时返回真值。
3. boolean contains(String input, Pattern pattern)：当正则表达式要匹配输入字符串的一部分时使用该方法。当正则表达式为输入字符串的子串时返回真值。

但以上三种方法只会查找输入字符串中匹配正则表达式的第一个对象，如果当字符串可能有多个子串匹配给定的正则表达式时，那么你就可以在调用上面三个方法时用 PatternMatcherInput 对象作为参数替代 String 对象，这样就可以从字符串中最后一次匹配的位置开始继续进行匹配，这样就方便的多了。

用 PatternMatcherInput 对象作为参数替代 String 时，上述三个方法的语法如下：

1. boolean matches(PatternMatcherInput input, Pattern pattern)
2. boolean matchesPrefix(PatternMatcherInput input, Pattern pattern)
3. boolean contains(PatternMatcherInput input, Pattern pattern)

★Util.substitute()方法:

查找后需要要进行替换，我们就要用到 Util.substitute()方法，其语法如下：

```
```

```
public static String substitute(PatternMatcher matcher,

       Pattern pattern,Substitution sub,String input,

       int numSubs)
```

前两个参数分别为 PatternMatcher 和 Pattern 对象。而第三个参数是个 Substiution 对象，由它来决定替换操作如何进行。第四个参数是要进行替换操作的目标字符串，最后一个参数用来指定是否替换模式的所有匹配子串（Util.SUBSTITUTE_ALL），或只进行指定次数的替换。

在这里我相信有必要详细解说一下第三个参数 Substiution 对象，因为它将决定替换将怎样进行。

Substiution:
Substiution 是一个接口类，它为你提供了在使用 Util.substitute()方法时控制替换方式的手段，它有两个标准的实现类：StringSubstitution 与 Perl5Substitution。当然，同时你也可以生成自己的实现类来定制你所需要的特殊替换动作。

StringSubstitution：
StringSubstitution 实现的是简单的纯文字替换手段，它有两个构造方法：

StringSubstitution()->缺省的构造方法，初始化一个包含零长度字符串的替换对象。

StringSubstitution(java.lang.String substitution)->初始化一个给定字符串的替换对象。

Perl5Substitution：
Perl5Substitution 是 StringSubstitution 的子类，它在实现纯文字替换手段的同时也允许进行针对 MATH 类里各匹配组的 PERL5 变量的替换，所以他的替换手段比其直接父类 StringSubstitution 更为多元化。

它有三个构造器：

Perl5Substitution()

Perl5Substitution(java.lang.String substitution)

Perl5Substitution(java.lang.String substitution, int numInterpolations)

前两种构造方法与 StringSubstitution 一样，而第三种构造方法下面将会介绍到。

在 **Perl5Substitution** 的替换字符串中可以包含用来替代在正则表达式里由小扩号围起来的匹配组的变量，这些变量是由$1，$2,$3 等形式来标识。我们可以用一个例子来解释怎样使用替换变量来进行替换：

假设我们有正则表达式模式为 b\d ：（也就是 b[0-9] ：），而我们想把所有匹配的字符串中的"b"都改为"a",而"："则改为"-"，而其余部分则不作修改，如我们输入字符串为" EXAMPLE b123:"，经过替换后就应该变成"EXAMPLE a123-"。要做到这点，我们就首先要把不做替换的部分用分组符号小括号包起来，这样正则表达式就变为"b(\d):"，而构造 Perl5Substitution 对象时其替换字符串就应该是"a$1-"，也就是构造式为 Perl5Substitution（"a$1-"），表示在使用 Util.substitute()方法时只要在目标字符串里找到和正则表达式" b(\d): "相匹配的子串都用替换字符串来替换，而变量$1 表示如果和正则表达式里第一个组相匹配的内容则照般原文插到$1 所在的为置，如在"EXAMPLE b123："中和正则表达式相匹配的部分是"b123："，而其中和第一分组"(\d)"相匹配的部分则是"123"，所以最后替换结果为"EXAMPLE a123-"。

有一点需要清楚的是，如果你把构造器 Perl5Substitution(java.lang.String substitution,int numInterpolations)

中的 numInterpolations 参数设为 INTERPOLATE_ALL，那么当每次找到一个匹配字串时，替换变量（$1，$2 等）所指向的内容都根据目前匹配字串来更新，但是如果 numInterpolations 参数设为一个正整数 N 时，那么在替换时就只会在前 N 次匹配发生时替换变量会跟随匹配对象来调整所代表的内容，但 N 次之后就以一致以第 N 次替换变量所代表内容来做为以后替换结果。

举个例子会更好理解：

假如沿用以上例子中的正则表达式模式以及替换内容来进行替换工作，设目标字符串为"Tank b123: 85 Tank b256: 32 Tank b78: 22"，并且设 numInterpolations 参数为 INTERPOLATE_ALL，而 Util.substitute()方法中的 numSub 变量设为 SUBSTITUTE_ALL（请参考上文 Util.substitute()方法内容），那么你获得的替换结果将会是：
Tank a123- 85 Tank a256- 32 Tank a78- 22

但是如果你把 numInterpolations 设为 2，并且 numSubs 依然设为 SUBSTITUTE_ALL，那么这时你获得的结果则会是：
Tank a123- 85 Tank a256- 32 Tank a256- 22

你要注意到最后一个替换所用变量$1 所代表的内容与第二个$1 一样为"256"，而不是预期的"78"，因为在替换进行中，替换变量$1 只根据匹配内容进行了两次更新，最后一次就使第二次匹配时所更新的结果，那么我们可以由此知道，如果 numInterpolations 设为 1，那么结果将是：
Tank a123- 85 Tank a123- 32 Tank a123- 22

3．应用示例：

刚好前段时间公司准备出一个《伊索预言》的英语学习互动教材，其中有电子档资料的整理工作，我们就以此为例来看一下 Jakarta-ORO 与 JDBC2.0 API 结合起来对数据库内的资料进行简单提取与整理的实现。假设由录入部的同事送过来的存放在 MS SQLSERVER 7 数据库里的电子档的表结构如下（注：或许在不同的 DBMS 中有相应的正则表达式的应用，但这不在本文讨论范围内）：

表名：AESOP，表中每条记录包含有三列：

ID（int）：单词索引号

WORD（varchar）：单词

CONTENT(varchar)：存放单词的相关解释与例句等内容

其中 CONTENT 列中内容的格式如下：

[音标] [词性] （解释）{(例句一/例句解释/例句中该词的词性：单词在句中的意思) (例句二/例句解释/例句中该词的词性：单词在句中的意思)}

如对应单词 Kevin,CONTENT 中的内容如下：

['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词：凯文)(Kevin is living in ZhuHai now./凯文现住在珠海/名词：凯文)}

我们的例子主要针对 CONTENT 列中内容进行字符串处理。

★查找单个匹配：

首先，让我们尝试把 CONTNET 列中的[音标]字段的内容列示出来，由于所有单词的记录中都有这一项并且都在字串开始位置，所以这个查找工作比较简单：

1. **确定相应的正则表达式：\[[^]] \]**

   这个是很简单的正则表达式，其意思是要求相匹配的字符串必须为以一对中括号包含的所有内容，如['kevin] 、[名词]等，但内容中不包括"]"符号，也就是要避免出现"[][]"会作为一个匹配对象的情况出现（有关正则表达式的基础知识请参照有关资料，这里不再详述）。

   注意，在 Java 中，你必须对每一个向前的斜杠（"\"）进行转义处理。所以我们要在上面的正则表达式里每个"\"前面加上一个"\"以免出现编译错误，也就是在 JAVA 中初始化正则表达式的字符串的语句应该为：

   ```
   String restring=" \\[[^]] \\]";
   ```

   并且在表达式里每个符号中间不能有空格，否则就会同样出现编译错误。

2. **实例化 PatternCompiler 对象，创建 Pattern 对象**

   ```
   PatternCompiler compiler=new Perl5Compiler();
   ```

Pattern pattern=compiler.compile(restring);

3. **创建 PatternMatcher 对象，调用 PatternMatcher 接口的 contain()方法检查匹配情况：**

```
PatternMatcher matcher=new Perl5Matcher();
      if (matcher.contains(content,pattern)) {
               //处理代码片段
      }
```

这里 matcher.contains(content,pattern)中的参数 content 是从数据库里取来的字符串变量。该方法只会查到第一个匹配的对象字符串，但是由于音标项均在 CONETNET 内容字符串中的起始位置，所以用这个方法就已经可以保证把每条记录里的音标项找出来了,但更为直接与合理的办法是使用 boolean matchesPrefix(PatternMatcherInput input, Pattern pattern)方法，该方法验证目标字符串是否以正则表达式所匹配的字串为起始。

具体实现的完整的程序代码如下：

```
package RegularExpressions;
//import……
import org.apache.oro.text.regex.*;
//使用 Jakarta-ORO 正则表达式库前需要把它加到 CLASSPATH 里面，如果用 IDE 是//JBUILDER，那么也可以在 JBUILDER 里直接自建新库。
public class yisuo{
  public static void main(String[] args){
  try{
//使用 JDBC DRIVER 进行 DBMS 连接，这里我使用的是一个第三方 JDBC
//DRIVER，Microsoft 本身也有一个面向 SQLSERVER7/2000 的免费 JDBC //DRIVER，但其性能真的是奇差，不用也罢。
        Class.forName("com.jnetdirect.jsql.JSQLDriver");
          Connection con=DriverManager.getConnection
```

```
          ("jdbc:JSQLConnect://kevin:1433","kevin chen","re");
          Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATABLE);
//为使用 Jakarta-ORO 库而创建相应的对象
String rsstring=" \\[[^]] \\]";
          PatternCompiler orocom=new Perl5Compiler();
          Pattern pattern=orocom.compile(rsstring);
          PatternMatcher matcher=new Perl5Matcher();
          ResultSet uprs = stmt.executeQuery("SELECT * FROM aesop");
          while (uprs.next()) {
Stirng  word=uprs.getString("word");
          Stirng  content=uprs.getString("content");
            if(matcher.contains(content,pattern)){
          //或 if(matcher.matchesPrefix(content,pattern)){
                MatchResult result=matcher.getMatch();
                Stirng pure=result.toString();
                System.out.println(word "的音标为：" pure);
            }
          }
       }
  catch(Exception e) {
             System.out.println(e);
       }
  }
}
```

输出结果为：kevin 的音标为['kevin]

在这个处理中我是用 toString()方法来取得结果，但是如果正则表达式里是用了分组符号（圆括号），那么就可以用 group(int gid)的方法来取得相应各组匹配的结果，如正则表达式改为" (\[[^]] \])"，那么就可以用以下方法来取得结果：pure=result.group(0);

用程序验证，输出结果同样为：kevin 的音标为['kevin]

而如果正则表达式为（\[[^]] \]）（\[[^]] \]），则会查找到两个连续的方括号所包含的内容，也就找到[音标] [词性]两项，但是两项的结果分别在两个组里面，分别由下面语句获得结果：

result.group(0)->返回[音标] [词性]两项内容，也就是与整个正则表达式相匹配的结果字符串，在这里也就为['kevin] [名词]

result.group(1) ->返回[音标]项内容，结果应是['kevin]

result.group(2) ->返回[词性]项内容，结果应是[名词]

继续用程序验证，发现输出并不正确，主要是当内容有中文时就不能成功匹配，考虑到可能是 Jakarta-ORO 正则表达式库版本不支持中文的问题，回看一下原来我一直用的还是 2.0.1 的老版本，马上到 Jakarta.org 上下载最新的 2.0.4 版本装上再用程序验证，得出的结果就和预期一样正确。

★查找多个匹配：
经过第一步的尝试使用 Jakarta-ORO 后，我们已经知道了如何正确使用该 API 包来查找目标字符串里一个匹配的子串，下面我们接着来看一看当目标字符串里包含不止一个匹配的子串时我们如何把它们一个接一个找出来进行相应的处理。

首先我们先试个简单的应用，假设我们想把 CONTNET 字段内容里所有用方括号包起来的字串都找出来，很清楚地，CONTNET 字段的内容里面就只有两项匹配的内容：[音标]和 [词性]，刚才我们其实已经把它们分别找出来了，但是我们所用的方法是分组方法，把"[音标] [词性]"作为一整个正则表达式匹配的内容先找到，再根据分组把[音标]和 [词性]分别挑出来。但是现在我们需要做的是把[音标]和[词性]分别做为与同一个正则表达式匹配的内容，先找到一个接着再找下一个，也就是刚才我们的表达式为（\[[^]] \]）（\[[^]] \]）,而现在应为" \[[^]] \] "。

我们已经知道在匹配操作的三个方法里只要用 PatternMatcherInput 对象作为参数替代 String 对象就可以从字符串中最后一次匹配的位置开始继续进行匹配，实现的程序片段如下：

```
PatternMatcherInput input=new PatternMatcherInput(content);
            while (matcher.contains(input,pattern)) {
                result=matcher.getMatch();
                System.out.println(result.group(0))
            }
```

输出结果为:['kevin]
[名词]

接着我们来做复杂一点的处理，就是我们要先把下面内容：
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)(Kevin is living in ZhuHai now. /凯文现住在珠海/名词: 凯文)}中的整个例句部分（也就是由大括号所包含的部分）找出来，再分别把例句一和例句二找出，而各例句中的各项内容（英文句、中文句、词性、解释）也要分项列出。

第一步当然是要定出相应的正则表达式，需要有两个，一是和整个例句部分（也就是由大括号包起来的部分）匹配的正则表达式："\{. \}",

另一个则要和每个例句部分匹配（也就是小括号中的内容），：\(([^)] \)

而且由于要把例句的各项分离出来，所以要再把里面的各部分用分组的方法匹配出来：" ([^(])/(.)/(.):([^)]) "。

为了简便起见，我们不再和从数据库里读出，而是构造一个包含同样内容的字符串变量，程序片段如下：

```
try{
        String content="['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词:凯文)
(Kevin is living in ZhuHai now./凯文现住在珠海/名词: 凯文)}";
        String ps1="\\{. \\}";
        String ps2="\\([^)] \\)";
        String ps3="([^(] )/(. )/(. ):([^)] )";
        String sentence;
        PatternCompiler orocom=new Perl5Compiler();
        Pattern pattern1=orocom.compile(ps1);
        Pattern pattern2=orocom.compile(ps2);
        Pattern pattern3=orocom.compile(ps3);
        PatternMatcher matcher=new Perl5Matcher();
//先找出整个例句部分
```

```
            if (matcher.contains(content,pattern1)) {
            MatchResult result=matcher.getMatch();
            String example=result.toString();
            PatternMatcherInput input=new PatternMatcherInput(example);
        //分别找出例句一和例句二
            while (matcher.contains(input,pattern2)){
                result=matcher.getMatch();
                sentence=result.toString();
        //把每个例句里的各项用分组的办法分隔出来
                if (matcher.contains(sentence,pattern3)){
                  result=matcher.getMatch();
                  System.out.println("英文句: " result.group(1));
                  System.out.println("句子中文翻译: " result.group(2));
                  System.out.println("词性: " result.group(3));
                  System.out.println("意思: " result.group(4));
                }
            }
       }
      }
  catch(Exception e) {
             System.out.println(e);
      }
```

输出结果为：

英文句: Kevin loves comic.
句子中文翻译: 凯文爱漫画
词性: 名词
意思: 凯文
英文句: Kevin is living in ZhuHai now.
句子中文翻译: 凯文现住在珠海

词性：名词
意思：凯文

★查找替换：
以上的两个应用都是单纯在查找字符串匹配方面的，我们再来看一下查找后如何对目标字符串进行替换。

例如我现在想把第二个例句进行改动，换为：Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。

也就是把
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词：凯文)(Kevin is living in ZhuHai now. /凯文现住在珠海/名词：凯文)}

改为：
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词：凯文)(Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}

之前，我们已经了解 Util.substitute()方法与 Substiution 接口，以及 Substiution 的两个实现类 StringSubstitution 和 Perl5Substitution，我们就来看看怎么用 Util.substitute()方法配合 Perl5Substitution 来完成我们上面提出的替换要求，确定正则表达式：

我们要先找到其中的整个例句部分，也就是由大括号包起来的字串，并且把两个例句分别分组，所以正则表达式为："\{(\([^)] \))(\([^)] \))\}"，如果用替换变量来代替分组，那么上面的表达式可以看为"\{$1$2\}",这样就可以更容易看出替换变量与分组间的关系。

根据上面的正则表达式 Perl5Substitution 类可以这样构造：
Perl5Substitution("{$1(Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}")

再根据这个 Perl5Substitution 对象来使用 Util.substitute()方法便可以完成替换了，实现的代码片段如下：

```
try{
   String content="['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词：凯文)(Kevin lives in ZhuHai now./凯文现住在珠海/名词：凯文)}";
   String ps1="\\{(\\([^)] \\))(\\([^)] \\))\\}";
   String sentence;
   String pure;
```

```
   PatternCompiler orocom=new Perl5Compiler();
   Pattern pattern1=orocom.compile(ps1);
   PatternMatcher matcher=new Perl5Matcher();
       String result=Util.substitute(matcher,
        pattern1,new Perl5Substitution(
       "{$1( Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}",1),
        content,Util.SUBSTITUTE_ALL);
        System.out.println(result);
   }
  catch(Exception e) {
            System.out.println(e);
       }
```

输出结果是正确的，为：
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)(Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}

至于有关使用 numInterpolations 参数的构造器用法，读者只要根据上面的介绍自己动手试一下就会清楚了，在此就不再例述。

总结：
本文首先介绍了 Jakarta-ORO 正则表达式库的对象与方法，并且接着举例让读者对实际应用有进一步的了解，虽然例子都比较简单，但希望读者们在看了该文后对 Jakarta-ORO 正则表达式库有一定的认知，在实际工作中有所帮助与启发。

其实在Jakarta org里除了Jakarta-ORO外还有一个百分百的纯JAVA正则表达式库，就是由Jonathan Locke赠与Jakarta ORG的Regexp，在该包里面包含了完整的文档以及一个用于调试的Applet例子，对其有兴趣的读者可以到此下载。

参考资料：

- 本文的主要参考文章，该文在介绍Jakarta-ORO的同时也为读者详尽解析了正则表达式的基本语法。
- 一个基于PERL的正则表达式详尽教程（虽然该教程是基于PERL的，但是你并不需要有PERL的经验，虽然那会有所帮助），以及一个不错的正则表达式简例教程。
- 最不可缺少的当然是Jakarta-ORO的帮助文档http://jakarta.apache.org/oro/api/

关于作者

陈广佳 Kevin Chen,汕头大学电子信息工程系工科学士，台湾大新出版社珠海区开发部，现正围绕中日韩

*
input1.txt 文件文件内容如下：
1 book at 12.49
1 music CD at 14.99
1 chocolate bar at 0.85

我现在要做的是：
读取文件的每一行，然后将每一行的开始数字、name、at 之后的数字取出来
例：
读取行 1 music CD at 14.99
结果为 ID = 1; name = music CD; price = 14.99;
我对正则表达式一点也不懂，有详细代码最好！
望各位指点！（急）
**/

```
package test1;
import java.util.regex.*;
import java.io.*;

public class Test2
...{

    /** *//**
     * @param args
     */
    public static void main(String[] args) {
        String regex="(\d+)\s(.*?)at\s([0-9\.])+";
        //String regex="(\d+) (.*?) at ([0-9\.])+";
        Pattern pt=Pattern.compile(regex);

        //读取文件数据
        try{
            FileInputStream fis=new FileInputStream("test.txt");
            InputStreamReader isr=new InputStreamReader(fis);
            BufferedReader br=new BufferedReader(isr);

            String s;
            while((s=br.readLine())!=null) {
                Matcher mt=pt.matcher(s);
                while(mt.find()){
```

```
                    System.out.println("ID="+mt.group(1));
                    System.out.println("Name="+mt.group(2));
                    System.out.println("Price="+mt.group(3));
                    System.out.println();

                }
            }

            br.close();
            isr.close();
            fis.close();


        }
        catch(FileNotFoundException e).{
            e.printStackTrace();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

}
```

出处：http://blog.csdn.net/zhuche110/archive/2008/04/19/2306933.aspx