Expressions régulières en Java avec l'API Regex

Hugo Etiévant

Dernière mise à jour : 05 août 2004

Introduction

Les expressions régulières (dites aussi « expressions rationnelles ») sont issue des recherches en mathématiques dans le domaine des automates. Les abréviations reconnues sont « regexp » et « regex ».

Une regex s'apparente à une expression mathématique, car on y trouve des opérateurs, des valeurs et des variables. Les regex permettent de se lancer à la recherche de motifs décrits par la combinaison d'opérateurs et de valeurs.

Une utilisation récurrente des regex consiste en la recherche de mots clés dans des fichiers ou dans une base de données ou encore en la vérification des données saisies par l'utilisateur afin de s'assurer qu'elles respectent un format prédéfini, ou même d'opérer des conversions de format.

L'API Regex de Java

Java fourni dans son JDK depuis la version 1.4 une API standard permettant la manipulation d'expressions régulières.

La documentation (en anglais) de cette API se trouve ici :

http://java.sun.com/j2se/1.4.2/docs/api/index.html

L'API doit être importée en début de fichier de définition de classe comme suit :

```
import java.util.regex.*;
```

Objets de l'API Regex

Ils existe deux classes et une exception :

Pattern

Représentation compilée d'un motif.

Matcher

Moteur de recherche d'un motif dans une chaîne de caractères.

PatternSyntaxException

Exception lancée lorsqu'une erreur apparaît dans la syntaxe des motifs employés.

Premier exemple

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

public class testRegex {

    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String args[]) {
        pattern = Pattern.compile("Hugo");
        matcher = pattern.matcher("Hugo Etiévant");
        while(matcher.find()) {
            System.out.println("Trouvé !");
        }
    }
}
```

Cet exemple recherche le motif « Hugo » dans la chaîne « Hugo Etiévant » et affiche « Trouvé ! » pour chaque occurrence du motif dans la chaîne.

Syntaxe des motifs

La syntaxe des motifs est très riche.

1. Chaînes littérales
2. Méta caractères
3. Classes de caractères
4. Quantificateurs
5. Groupes de capture
6. Frontières de recherche

Chaînes littérales

La syntaxe la plus aisée consiste à recherche une simple chaîne de caractères au sein d'une autre.

Exemple :

Motif : "simple chaîne de caractères"
Chaîne à traiter : "autre simple chaîne de caractères"
Résultat trouvé : "simple chaîne de caractères"

Méta caractères (1)

L'étape suivante dans la complexité consiste à ajouter des symboles particuliers dont la signification est qu'ils remplacent d'autres caractères. Un peu comme la lettre blanche aux scrabbles qui représente n'importe quelle lettre.

Sauf qu'ici un grand nombre de symboles existent, ils peuvent être combinés entre eux et donner des expressions complexes.

Exemple :

Motif : "voiture."
Chaîne à traiter : "les voitures"
Résultat trouvé : "voitures"

Le caractères spécial . remplace n'importe quel caractère.

Méta caractères (2)

Voici la liste des méta caractères :

Caractère	Description
.	Remplace tout caractère
*	Remplace une chaîne de 0, 1 ou plusieurs caractères
?	Remplace exactement un caractère
()	Groupe capturant
[]	Intervalle de caractères
{}	Quantificateur
\	Déspécialise le caractère spécial qu'il précède
^	Négation ou début de ligne
$	Fin de ligne
|	Ou logique entre deux sous-motifs
+	Numérateur

Classes de caractères (1)

Une classe de caractères est un ensemble de caractères. Il existe des classes prédéfinies par l'API mais d'autres ensembles peuvent être construits par le programmeur.

Voici la liste des classes prédéfinies :

Classe	Description
\d	Un chiffre, équivalent à : [0-9]
\D	Un non chiffre : [^0-9]
\s	Un caractère blanc : [\t\n\x0B\f\r]
\S	Un non caractère blanc : [^\s]
\w	Un caractère de mot : [a-zA-Z_0-9]
\W	Un caractère de non mot : [^\w]
.	Tout caractère

Attention : le caractère \ est un caractère spécial, il doit être déspécialisé lorsque un alias de classe ou tout autre mot clé des regex l'utilise. Il est déspécialisé lorsqu'il est doublé \\.

Classes de caractères (2)

Exemples :

Motif : "\d"
Chaîne à traiter : "j'ai 2 voitures"
Résultat trouvé : "2"

Motif : "\W"
Chaîne à traiter : "j'ai 2 voitures"
Résultats trouvés : "'", " ", " " (l'apostrophe et les deux espaces)

Classes de caractères (3)

Voici les règles de constructions des classes personnalisées :

Classe	Description
[abc]	**Ensemble simple**, remplace tout caractère parmi l'un des caractères suivants : a, b et c
[^ abc]	**Négation** de l'ensemble précédent
[a-z]	**Ensemble complexe**, remplace tout caractère parmi ceux de l'alphabet naturel compris entre a et z
[a-zA-Z] [a-z[A-Z]]	**Union** d'ensembles, remplace tout caractère de l'alphabet minuscule ou majuscule
[abc&&[a-z]]	**Intersection** d'ensembles, remplace tout caractère faisant parti de l'ensemble : a, b, c et aussi de l'ensemble de a jusqu'à z (c'est-à-dire uniquement a, b et c)
[a-z&&[^ abc]]	**Soustraction** d'ensembles, remplace tout caractère de l'alphabet compris entre a et z, excepté ceux de l'intervalle suivant : a, b et c

Classes de caractères (4)

Exemples :

Motif	Chaîne	Résultat(s)
"[A-Z]"	"abc"	aucun
"[AbfTz]"	"l'Amour"	"A"
"[^0-9]"	"as"	"a", "s"
"[\w&&[^13579]]"	"getId23"	"g", "e", "t", "I", "d", "2"
"[123&&[1-9]]"	"5"	aucun
"[123&&[1-9]]"	"1"	"1"
"[0-9&&[^123]]"	"3"	aucun
"[0-9&&[^123]]"	"8"	"8"

Quantificateurs (1)

Un quantificateur permet de spécifier le nombre d'occurrences d'un sous-motif du motif. En voici la liste :

Quantificateurs			Description
Avide	Réticent	Possessif	
X?	X??	X?+	Une fois ou pas du tout
X*	X*?	X*+	Zéro, une ou plusieurs fois
X+	X+?	X++	Une ou plusieurs fois
X{n}	X{n}?	X{n}+	Exactement n fois
X{n,}	X{n,}?	X{n,}+	Au moins n fois
X{n, m}	X{n, m}?	X{n, m}+	Au moins n fois et jusqu'à m fois

Quantificateurs (2)

Il existe trois classes de quantificateurs :

- les **avides** (greedy) : lecture de toute la chaîne d'entrée avant de rechercher des occurrences en partant de la fin dans le but de trouver le maximum d'occurrences,
- les **réticents** (reluctant) : lecture de la chaîne d'entrée caractère par caractère à la recherche des occurrences,
- les **possessifs** (possessive) : lecture de toute la chaîne d'entrée avant de chercher une occurrence.

Certains quantificateurs rendent satisfaisant des chaînes qui ne comportent pas la chaîne du motif. Par exemple le motif "a*" correspond à trouver zéro fois ou plus la lettre a. Parmi les résultats de la recherche, un chaîne vide "" apparaîtra car vérifiera le motif.

Quantificateurs (3)

Exemples :

Motif	Chaîne	Résultat(s)
"(to)+"	"toto"	"toto"
"(to)*"	"toto"	"toto", "" chaîne vide de fin
"(to)?"	"toto"	"to", "to", "" chaîne vide de fin
"a{2}"	"aaaa"	"aa", "aa"
"a?"	"aaaa"	"a", "a", "a", "a", "" chaîne vide de fin
"a+"	"aaaa"	"aaaa"
"a++"	"aaaa"	"aaaa"
"a+?"	"aaaa"	"a", "a", "a", "a"
"a{2,4}"	"aaaa"	"aaaa"
"[0-9]{4}"	"from 1997 to 2004 for the 2nd time"	"1997", "2004"

Groupes de captures (1)

Les parenthèses utilisées dans les regex permettent de créer des groupes de sous-motifs.

Par exemple, le motif "(a((bc)(d)))" défini 4 groupes : "(a((bc)(d)))", "((bc)(d))", "(bc)" et "(d)". Il sont numérotés de gauche à droite selon l'ordre de leur parenthèse ouvrante. Le groupe 0 contient toujours l'expression entière même si aucun groupe n'est défini.

Lors de l'exécution de la regex par le moteur Matcher sur une chaîne, les sous-chaînes vérifiant les sous-motifs définis par chacun des groupes seront capturées, c'est-à-dire conservées en mémoire et pourront être réutilisées.

Groupes de captures (2)

Exemple :

Motif : "(a((b)(c)))"
- groupe 0 : (a((b)(c)))
- groupe 1 : (a((b)(c)))
- groupe 2 : ((b)(c))
- groupe 3 : (b)
- groupe 4 : (c)

Chaîne à traiter : "abc"

Résultats trouvés :
- groupe 0 : "abc",
- groupe 1 : "abc",
- groupe 2 : "bc",
- groupe 3 : "b",
- groupe 4 : "c"

Groupes de captures (3)

Après application de la regex sur une chaîne, il est possible de connaître le nombre de sous-chaînes capturées avec la méthode **groupCount**() de l'objet Matcher.

La méthode **group**(int group) retourne la sous-chaîne capturée par le groupe n°group.

Groupes de captures – exemple a (4)

```
// compilation de la regex
Pattern p = Pattern.compile("(a((b)(c)))");
// création d'un moteur de recherche
Matcher m = p.matcher("abc");
// lancement de la recherche de toutes les occurrences
boolean b = m.matches();

// si recherche fructueuse
if(b) {

  // pour chaque groupe
  for(int i=0; i<=m.groupCount(); i++) {
    // affichage de la sous-chaîne capturée
    System.out.println("Groupe " + i + " : " + m.group(i));
  }

}
```

Affiche :

```
Groupe 0 : abc
Groupe 1 : abc
Groupe 2 : bc
Groupe 3 : b
Groupe 4 : c
```

Groupes de captures – exemple b (5)

```
// compilation de la regex
Pattern p = Pattern.compile("(a((b)(c)))");
// création d'un moteur de recherche
Matcher m = p.matcher("abc");

// lancement de la recherche de toutes les occurrences successives
while(m.find()) {
  // affichage de la sous-chaîne capturée,
  // de l'index de début dans la chaîne originale
  // et de l'index de fin
  System.out.println("Le texte \"" + m.group() +
    "\" débute à " + m.start() + " et termine à " + m.end());
}
```

Affiche : Le texte "abc" débute à 0 et termine à 3.

Groupes de captures – méthodes (6)

Voici quelques méthodes courantes relatives aux sous-chaînes capturées, c'est-à-dire aux résultats de la recherche du motif dans une chaîne d'entrée :

Méthode	Description
int groupCount()	Nombre de sous-chaînes capturées
string group()	Sous-chaîne capturée par la dernière recherche
string group(int group)	Sous-chaîne capturée par le groupe group
boolean find()	Recherche de la prochaine sous-chaîne satisfaisant la regex
boolean find(int start)	Recherche de la prochaine sous-chaîne satisfaisant la regex, en commençant la recherche à l'index start
int start()	Index de début de la sous-chaîne capturée
int start(int group)	Index de début de la sous-chaîne capturée par le groupe group
int end()	Index de fin de la sous-chaîne capturée
int end(int group)	Index de fin de la sous-chaîne capturée par le groupe group

Groupes de captures – références (7)

Au sein du motif, on peut ajouter une référence à un groupe du même motif.

Syntaxe : \i où i est le numéro de groupe.

Exemples :

Motif : "(\d\d)\1"
Chaîne à traiter : "1515"
Résultat trouvé : "1515"

Motif : "(\d\d)\1"
Chaîne à traiter : "1789"
Résultat trouvé : aucun

Dans cet exemple, le premier groupe capturant est (\d\d) c'est-à-dire deux chiffres successifs. La suite du motif : \1 signifie qu'il faut trouver à la suite de la sous-chaîne vérifiant \d\d une sous-chaîne identique à celle capturée.

1) Ici, 15 est la sous-chaîne capturée par (\d\d), à sa suite, 15 est effectivement identique à la première.

2) Ici, 17 est capturée, mais 89 qui la suit ne lui est pas égale, même si elle vérifie le motif initial \d\d elle n'est pas égale à l'occurrence capturée.

Frontières de recherche (1)

Il est désormais intéressant de forcer l'emplacement des motifs recherchés : en début de ligne, en fin de mot…
Les « spécificateurs de frontière » sont résumés dans le tableau suivant :

Spécificateur	Description
^	Début de ligne
$	Fin de ligne
\b	Extrémité de mot
\B	Extrémité d'un non mot
\A	Début de la chaîne soumise
\G	Fin de l'occurrence précédente
\Z	Fin de la chaîne soumise, à l'exclusion du caractère final
\z	Fin de la chaîne soumise

Frontières de recherche (2)

Exemples :

Motif	Chaîne	Résultat(s)
"^java$"	"java"	"java"
"^java$"	"le java"	aucun
"ciné\b"	"je vais au ciné"	"ciné"
"ciné\b"	"je vais au cinéma"	aucun
"\Gjava"	"java java"	"java" (le premier, car le deuxième n'apparaît pas après le premier mais après le caractère espace)
"\Gjava"	"javajava"	"java", "java" (les deux)

Options pour les regex (1)

La méthode de compilation d'expression régulière prend pour paramètres la regex et un paramètre optionnel flags.

Syntaxe : static **Pattern compile**(String regex, int flags)

Liste des options :

Constante	Description
CANON_EQ	Autorise l'équivalence canonique
CASE_INSENSITIVE	Insensibilité à la casse
COMMENTS	Autorise les espaces et commentaires dans la regex
DOTALL	Autorise le mode « point à tout » (dotall)
MULTILINE	Autorise le mode multilignes
UNICODE_CASE	Autorise la gestion des caractères Unicode
UNIX_LINES	Autorise le codage Unix des fins de ligne

Options pour les regex (2)

Ces options existent sous la forme de constantes de type entier (static int) dans la classe Pattern.

Plusieurs options peuvent être combinées grâce à l'opérateurs OU binaire : |.

Exemples :

```
Pattern p = Pattern.compile("^[abc]$",
            Pattern.CASE_INSENSITIVE);

Pattern p = Pattern.compile ("^[abc]$",
            Pattern.MULTILINE | Pattern.UNIX_LINES);
```

Options embarquées (3)

Nous avons vu comment passer ces options en paramètre à la méthode compile(). Il est également possible d'écrire ces options directement dans le motif de la regex. Elles doivent être placées en tout début.

Constante	Équivalent embarqué
CANON_EQ	aucun
CASE_INSENSITIVE	(?i)
COMMENTS	(?x)
DOTALL	(?s)
MULTILINE	(?m)
UNICODE_CASE	(?u)
UNIX_LINES	(?d)

Options embarquées (4)

Exemple :

Motif : "**(?i)**foobar"
Chaîne à traiter : "FooBar, foobar, FOOBAR"
Résultats trouvés : "FooBar", "foobar", "FOOBAR"

Méthode matches() – Matcher (1)

La méthode matches() retourne vrai (true) si une chaîne vérifie un motif. Cette méthode existe dans les classes Matcher et Pattern.

Objet Matcher

Syntaxe :

```
boolean matches() ;
```

Exemple :

```
// compilation de la regex
Pattern p = Pattern.compile("(a((b)(c)))");
// création d'un moteur de recherche
Matcher m = p.matcher("abc");
// lancement de la recherche de toutes les occurrences
boolean b = m.matches();
```

Méthode matches() – Pattern (2)

Dans la classe Pattern, cette méthode peut être appelée plus rapidement.

Objet Pattern

Syntaxe :

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

Exemple :

```
// lancement de la recherche de toutes les occurrences
boolean b = Pattern.matches("(a((b)(c)))", "abc");
```

Méthode split() (1)

La méthode split() de la classe Pattern permet de scinder une chaîne en plusieurs sous-chaînes grâce à un délimiteur défini par un motif. Le paramètre optionnel limit permet de fixer le nombre maximum de sous-chaînes générées. Elle retourne un tableau de String.

Syntaxe :

```
String[] split(CharSequence input [, int limit])
```

Exemple :

```
// compilation de la regex
Pattern p = Pattern.compile(":");
// séparation en sous-chaînes
String[] items = p.split("un:deux:trois");
```

Méthode split() – exemple (2)

Exemple complet :

```
// compilation de la regex
Pattern p = Pattern.compile("\\W");
// séparation en sous-chaînes
String[] items = p.split("J'aime le chocolat.", 10);
// parcours du tableau des sous-chaînes
for(int i=0; i<items.length; i++) {
    System.out.println(items[i]);
}
```

Cet exemple scinde une phrase en ses 10 premiers mots. Le motif \W signifie tout caractère de non mot.

Le résultat est le suivant : J, aime, le, chocolat.

Remplacements (1)

La classe Matcher offre des fonctions qui permettent de remplacer les occurrences d'un motif par une autre chaîne.

Syntaxe :

String **replaceFirst**(String replacement)

String **replaceAll**(String replacement)

Ces méthodes remplacent respectivement la première occurrence et toutes les occurrences du motif de la regex compilée associés au moteur.

Remplacements – exemple (2)

Exemple complet :

```
// compilation de la regex avec le motif : "thé"
Pattern p = Pattern.compile("thé");
// création du moteur associé à la regex sur la chaîne "J'aime le thé."
Matcher m = p.matcher("J'aime le thé.");
// remplacement de toutes les occurrences de "thé" par "chocolat"
String s = m.replaceAll("chocolat");
```

Dans cette exemple, la chaîne d'arrivée s contient : "J'aime le chocolat.".

Historique

- **05 août 2004** : corrections mineures
- **22 mai 2004** : première publication (36 diapos)
- **20 mai 2004** : création du document (20 diapos)

Agissez sur la qualité de ce document en envoyant vos critiques et suggestions à l'auteur.

Pour toute question technique, se reporter au forum Java de Developpez.com

Hugo Etiévant
cyberzoide@yahoo.fr
http://cyberzoide.developpez.com/