par

Avant-propos

Les expressions régulières (également appelées Regex ou Regexp) sont une traduction contreversée de "regular expressions" que les puristes appeleront les expressions rationnelles. Par la suite, nous utiliserons donc le terme "Regex" afin de rester neutre :)

1 - Les expressions régulières

1.1 - Leur utilisation

Les Regex sont un outil (ou plutôt un système) très puissant permettant de vérifier la syntaxe d'une chaîne de caractères. Plus précisément, c'est vérifier que la chaîne de caractères à examiner respecte un motif ou une série de motifs (notre Regex) désignant la syntaxe attendue de la phrase.

Les Regex sont utilisées pour trois grands types d'action :

- Vérifier la syntaxe(forme) d'une chaîne de caractères (ex : adresse ip de forme chiffre.chiffre.chiffre.chiffre)

- Remplacer une partie de la chaîne (token) par un élément spécifique

- Découper une chaîne de caractères

Leur utilisation la plus courante est de vérifier la syntaxe d'une chaîne, c'est-à-dire contrôler qu'elle respecte un format prédéfini et normé, décrit par les combinaisons d'opérateurs et de valeurs de la chaîne Regex.

1.2 - Leur syntaxe

Les Regex sont donc une suite de motifs qui se composent de métacaractères, de classes sous .NET et d'alias.

Voici tout d'abord, une série de métacaractères pouvant être utilisés et ayant chacun une correspondance bien précise.

Symbole	Correspondance	Exemple
\	Caractère d'échappement	[\.] contient un "."
^	Début de ligne	^b$ contient uniquement b
.	N'importe quel caractère	^.$ contient un seul caractère
$	Fin de ligne	er$ finit par "er"
|	Alternative	^(a|A) commence par a ou A
( )	Groupement	^((a)|(er)) commence par a ou er
-	Intervalle de caractères	^[a-d] commence par a,b,c ou d
[ ]	Ensemble de caractères	[0-9] contient un chiffre
[^]	Tout sauf un ensemble de caractères	^[^a] ne commence pas par a
+	1 fois ou plus	^(a)+ commence par un ou plusieurs a
?	0 ou 1 fois	^(a)? commence ou non par un a
*	0 fois ou plus	^(a)* peut ou non commencer par a
{x}	x fois exactement	a{2} deux fois "a"
{x,}	x fois ou moins	a{2,} deux fois "a" ou moins
{x, y}	x fois minimum, y maximum	a{2,4} deux, trois ou quatre fois "a"

Ces métacaractères nous permettent donc de faire des combinaisons infinies pouvant correspondre à tous les besoins. Néanmoins, pour vérifier des chaînes de caractères de taille importante, l'écriture de ces métacaractères peut être fastidieuse, en plus du fait que la chaine Regex n'est pas très lisible pour le developpeur. Heureusement, il existe une série d'alias permettant de faciliter cette tâche :

2 - Les expressions régulières et .Net

Les Regex s'utilisent énormément sous UNIX avec PERL mais également dans les langages web comme le PHP. Même si la syntaxe de base est identique, l'utilisation de celle-ci diffère un peu pour chaque langage, notamment les méthodes qui sont différentes. Ainsi, nous ne verrons pas ici les fonctions POSIX comme **ereg( )**, mais nous nous intéresserons uniquement aux méthodes propres aux langages .NET.

2.1 - Utilisation en .Net

Les Regex nécessitent l'utilisation de l'espace de noms **System.Text.RegularExpressions**

```
using System.Text.RegularExpressions;
```

En .Net, les Regex sont à indiquer entre guillemets, si celles-ci comprennent des antislashs (" \ "), il faut alors les doubler. Néanmoins, il existe une astuce utile ici, qui consiste à placer le symbole @ avant les guillemets.

Ainsi : @"^[\t]$" est équivalent à "^[\\t]$"

De plus, pour faciliter la vérification de vos Regex, je mets à disposition un petit programme qui permettra également de vérifier la validité des exemples que nous verrons par la suite. RegexMaker (32ko) Miroir

2.2 - Validation d'une chaîne

La validation d'une chaîne de caractères se fait grâce à la méthode IsMatch() qui retourne une valeur booléenne égale à true si la chaîne correspond à la Regex et false dans le cas contraire.

Voici comme exemple, la méthode **ValidMail()** vérifie la validité d'une adresse mail passée en paramètre.

```
private bool ValidMail(string adresse)
{
    System.Text.RegularExpressions.Regex myRegex = new Regex(@"^([\w]+)@([\w]+)\.([\w]+)$");
    //([\w]+) ==> caractère alphanumérique apparaissant une fois ou plus
    return myRegex.IsMatch(adresse); // retourne true ou false selon la vérification
}
```

Le fonctionnement de cette méthode consiste donc à créer un objet Regex que l'on initialise avec notre expression, puis on passe à cet objet la chaîne à valider (ici la chaine *adresse*).

2.3 - Remplacement dans une chaîne

Une deuxième utilité des Regex est de pouvoir rechercher des occurrences de chaînes spécifiques et de les remplacer par une autre chaîne. Dans ce cas précis, nous utiliserons la méthode Replace(). Celle-ci est plus ou moins identique à la méthode Replace() de l'objet string, mais cette dernière ne permet de remplacer qu'une seule chaîne de caractères par une autre. Prenons comme exemple, la méthode, **SMS** qui s'assure de remplacer certains termes par d'autres:

```
private string SMS(string chaine)
{
    System.Text.RegularExpressions.Regex myRegex = new Regex("(lut|salut|yop)");
```

2.4.1 - Découpage par séparateur

Grâce aux Regex, il est également possible de découper une chaîne de caractères de différentes manières.

La première méthode est la méthode Split() qui génère un tableau de chaînes de caractères en découpant une chaîne de départ via un séparateur : mais ici, nous pouvons déclarer des séparateurs dynamiques.

Voici comme exemple, la méthode **Splitter()** qui découpe une chaîne caractères en plusieurs chaînes dont les séparateurs sont des chiffres

```
private string[] Splitter(string chaine)
{
   Regex myRegex=new Regex(@"\d+");
   return myRegex.Split(chaine);
}
```

En lui passant la chaîne "Voici3234un2exemple23423427de2chaîne3424de767caractères"; elle renverra un tableau contenant {Voici, un, exemple, de, chaîne, de, caractères}

2.4.2 - Découpage par regroupement

La deuxième méthode permet également de ranger dans un tableau, différents éléments "extraits" de la chaîne de caractères à travailler. Puis une astuce permet de réutiliser ces éléments "extraits" selon les besoins que l'on a d'eux. Voici donc une méthode qui va renvoyer un tableau des informations concernant le serveur à partir de son chemin LDAP, puis réaffiche les résultats. Vous noterez que nous utilisons ici encore les "parenthèses de capture" (cf chapitre précédent) afin de grouper les différents motifs à récupérer.

```
Regex myRegex=new Regex(@"^LDAP://CN=([\w]+), CN=([\w]+), DC=([\w]+), DC=([\w]+)$");
Match m= myRegex.Match("LDAP://CN=moderateurs, CN=pharaonix, DC=com, DC=developpez");
if(m.Success)
{
   MessageBox.Show("Utilisateur: " + m.Groups[2].Value);
   MessageBox.Show("Groupe: " + m.Groups[1].Value);
   MessageBox.Show("Serveur: " + m.Groups[4].Value+"."+m.Groups[3].Value);
}
```

Conclusion

Nous avons donc vu qu'il existe différentes manières d'utiliser les Regex, et que malgré leur aspect complexe au premier abord, elles ont pour but la simplification du travail sur les chaînes. Il est ainsi très facile de vérifier la validité d'une chaîne et d'en ressortir les erreurs et/ou de les corriger. J'espère que vous saurez maintenant comprendre le fonctionnement des expressions rationnelles et que vous saurez les utiliser à bon escient, quelque soit le langage choisi.

Liens

Je joins également les sources des codes transcrits en Delphi par Laurent DARDENNE:

Sources Delphi

Voici quelques liens utiles en rapport avec les expressions régulières:

Article sur les Regex et Java (Hugo Etievant)

Article sur les Regex et php (Hugo Etievant)

Site complet sur les différentes utilisations des Regex

Outil de Regex

Des centaines de chaînes Regex toutes faites

Un remerciement tout spécial à Laurent DARDENNE pour tout le temps qu'il a passé à m'aider à améliorer cet article ainsi qu'à Emerica