

Systèmes d'exploitation

RegExp: Higher and Deeper

Yves STADLER

Codasystem, UPV-M

12 septembre 2011

1/19

Commandes

POSIX

- `ereg` (matching)
- `eregi` (matching / case insensitive)
- `ereg_replace` (replacement)
- `eregi_replace` (case insensitive replacement)
- `split` (découpe)
- `spliti` (découpe insensible à la casse)

3/19

Objectifs

- Savoir quelles commandes peuvent-être utilisées avec des regexp.
- Savoir utiliser la capture et la *backreference*
- Contrôler sa gourmandise
- Rechercher toutes les occurrences

2/19

Commandes

PCRE

- `preg_grep` — Retourne un tableau avec les résultats de la recherche
- `preg_match` — Une seule occurrence
- `preg_match_all` — Toutes occurrences
- `preg_replace` — rechercher et remplacer
- `preg_replace_callback` — Recherche, remplace en utilisant une fonction
- `preg_split` — Eclatement de chaîne

4/19

La parenthèse

- Utiliser une parenthèse créer une pseudo-variable \i qui stocke la valeur du sous-motif.

0

Exemple

- `[0-9]{2}-[0-9]{2}-[0-9]{4}` désigne une chaîne de type date JJ-MM-AAAA
- `([0-9]{2})-([0-9]{2})-([0-9]{4})`
- `\1` prend la valeur de JJ
- `\2` prend la valeur de MM
- `\3` prend la valeur de AAAA

5/19

Non capturante, groupements nommés, backreference

PCRE only !

- On peut annuler la capture avec les PCRE (et sans doute aussi dans POSIX étendu)
- `(?:...)`
- Pour utiliser une backreference, il faut au moins le nombre de parenthèses requis.

7/19

Exemple

- `(.*)\1` représente les phrases en deux parties identiques ("pouetpouet", "haha", "toto")
- `t([aeiou])t\1` désigne toutes les chaînes tXtY ou X=Y= une voyelle
- `0[1-6]([0-9]{2}){2}\1\2` désigne les numéros de téléphones qui se terminent par deux groupes de 4 chiffres identiques (06 08 12 08 12, 03 87 70 87 70, ...)

6/19

La gourmandise

Analysons ceci :

- La RE : `.*A.*`
- Et la phrase : "zAzAzAz"
- Comment va réagir la correspondance ?

Possibilités

- `.* = z | A | .* = zAzAz`
- `.* = zAz | A | .* = zAz`
- `.* = zAzAz | A | .* = zAz`

8/19

La gourmandise

Est-un vilain défaut

- Dans les RE POSIX, les quantifieurs sont toujours gourmands (Plus c'est long... plus c'est bon)
- Avec les PCRE, les quantifieurs ne sont gourmands que par défaut.

Cure de désintox

- Pour supprimer la gourmandise d'un opérateur on utilise ?
- `.*?A.*`
- Capture : `.* = z | A | .* = zAzAz`

9/19

Assertions

Assertions simples

- `\b` Limite d'un mot
- `\B` pas limite d'un mot
- `^` début de phrase
- `$` fin de phrase

11/19

La gourmandise

Avec PHP

- Fonction `ereg()`
- `ereg('', $text, $out);`
- `$text="Une phrase avec que je veux pouvoir matcher mais pas le reste"!`
- `out` contiendra le résultat de la capture
- POSIX : `'http ://www.codasystem.com/mapage.php'> que je veux pouvoir matcher</a'`

PCRE

- Fonction `preg_match()`
- `preg_match('', $text, $out);`
- `out` contiendra le résultat de la capture
- PCRE : `"http ://www.codasystem.com/mapage.php"`

10/19

Assertions

Lookahead assertions

- `(?=...)`, réussit si la regexp match l'expression à cette position.
- `(?!...)`, l'inverse
- L'assertion laisse le curseur de lecture là où l'on commence à tester l'assertion (c'est à dire que si l'assertion est vraie on va relire depuis le début du groupe assertif)

Exemple

- Matcher les fichiers et leurs extensions
- `.*[.].*$`
- Mais sans les `.bat`

12/19

Ce qui ne marche pas

- `.*[.][^b].*$`
- `.*[.]([^b]...|. [^a]...|^t)$`
- `.*[.]([^b].?.?|. [^a].?.?|^t.?.?)$`

Solution avec lookahead

- `.*[.](?!bat$).*$`
- Si après le `.` il n'y a pas `bat`, on continue.
- Le `.*` après l'assertion matchera l'extension (la lecture reste bloquée au début de l'assertion/)

13/19

Masques conditionnels

Syntaxe

- `(?(assertion ou backreference) motif si vrai | motif sur faux)`

Exemple

- Si je détecte `From` dans un mail, je traite un motif d'adresse email
- Si je détecte `Subject`, je traite une suite de mots

15/19

Un pas en avant, trois pas en arrière !

- `(?<=...)`, réussit si la regexp match l'expression en arrière de cette position.
- `(?<!...)`, réussit si ne match pas (en arrière)

14/19

Failing attempts

Capture vide, et groupe non participant

- `(q?)b\1` match b
- `q?` match le vide, le groupe capture vide, b match b, \1 doit matcher vide c'est ok.
- `(q)?b\1` ne match pas b
- `(q)` match le vide (optionnel), le groupe ne capture rien, car il ne participe pas à l'expression. b match b, \1 va échouer car le groupe n'a pas participé.

16/19

Que se passe-t-il pour la capture si

- J'utilise `azerty([0-9])+`
- La capture `([0-9])` risque d'intervenir plusieurs fois.
- `\1` ne se souviendra que de la dernière capture.

Utilisons une capture avant de la définir

- Fonctionne avec PCRE
- `(\2deux|un)+` va matcher : `unundeux`
- Au début, `\2` échoue à matcher `u`, la seconde alternative `"un"` réussit (`un` est capturé)
- Deuxième itération `\2` match `"un"` et `deux` match `deux`.

Quelques options

PCRE

- On précise l'option après le `#` final.
- `i` : case insensitive
- `s` : match aussi le retour à la ligne
- `U` : ungreedy par défaut
- `x` : permet d'écrire ses motifs sur plusieurs lignes, avec des commentaires (`#`, il faut utiliser un autre délimiteur (ex. `/`))