Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Expresiones Regulares - Conceptos Avanzados - (35763 lectures)

Per **Daniel Rodriguez**, ***DaniRC*** (http://www.ibiza-beach.com/)

Creado el 18/07/2001 16:19 modificado el 18/07/2001 16:19

> *Todos hemos oido hablar o incluso usado alguna vez las expresiones regulares.*
> *Sabemos que son muy potentes, que son casi ininteligibles por los profanos y que con ellas hacemos cosas que los winusers no harian ni con una macro de 2000 lineas. Pero son un arma de doble filo si no se manejan de forma adecuada*

Las expresiones regulares se basan en el backtracking, lo que llamamos en castellano "la vuelta atras". Es un tipo de programación compleja que tiene siempre como ejemplo clasico la colocación de 8 reinas en un tablero de ajedrez. (otro ejemplo de uso de backtracking es el propio lenguaje PROLOG, los que lo hayais usado ya sabeis lo complejo que resulta obtener una solucion correcta con él, en los primeros dias de intentos)

No se si recordais en particular el ejemplo del tablero de ajedrez con 8 reinas, pero pensad que tiene unas 92 soluciones -ahora no lo recuerdo con exactitud- ... Ahora bien ... imaginaos que el reemplazo de una cadena en un texto medianamente largo tuviera tambien 92 soluciones.

El problema no radica en que tenga 92 soluciones, el problema radica en que tiene muchas mas segun cual sea el texto del documento, cual sea el patron de reemplazo y las demas variables.

Por norma cuando alguien empieza con las Expresiones Regulares se da por satisfecho con que le salga un resultado y lo considera como EL RESULTADO. No para a pensar en si ese es el verdadero resultado, si es el unico, si ademas de ese han quedado mas sin aparecer, ... etc.

Hay que conocer muy bien el funcionamiento interno de las expresiones regulares para no cometer errores infantiles cuando lo usamos. Saber que clase de condiciones implican backtracking y cuales no ... si una condicion va a consumir o no caracteres de la cadena mientras de comprueba su concordancia ... y en fin .. otras muchas cosas que no siempre aparecen en los habituales tutoriales de expresiones regulares.

Como no quiero atribuirme un mérito que no es mio, os voy a dar una traducción libre del articulo en inglés que me abrió los ojos. Podeis leerlo directamente en original aquí[1]. Hoy veremos de verdad como funcionan las expresiones regulares y las sorpresas que nos puede deparar su uso arbritario.

Backtracking - Vuelta atrás

Una caracteristica fundamental de las concordancias de patrones en las expresiones regulares lleva involucrada la noción de ***vuelta atrás*** Cuando es usada (porque es necesitada) en la expresion regular encontraremos estos cuantificadores *,*?,+,+?,{n,m}, y {n,m}?

Para que una expresion regular concuerde, la expresion regular *al completo* debe concordar, no basta con una parte de ella. Pero si el principio del patron contiene un cuantificador ocurre que la concordancia se logra de forma que se elige un camino y si posteriormente el patron deja de concordar se **vuelve atras** y se busca otro camino recalculando a partir de la parte inicial del patron que no falló --esto es lo que llamamos backtracking o vuelta atras.

He aqui un ejemplo de vuelta atras: Imagenemos que buscamos el texto "foo" en el string "Food is on the foo table."

```
$_ = "Food is on the foo table.";
  if ( /\b(foo)\s+(\w+)/i ) {
```

```
    print "$2 sigue a $1.\n";
  }
```

Cuando se ejecuta esto, la primera parte de la expresion regular (\b(foo)) encuentra una posible concordancia al principio del string, y asigna **$1** el valor "Foo". Ahora el programa de concordancias comprueba que detras del valor que tiene en $1 no hay un espacio y que esto ha sido un error y vuelve a empezar desde el ultimo caracter que habia comprobado antes de cometer el error. Esta vez sigue el camino hacia "foo", la expresion regular al completo concuerda esta vez, y ahora leeremos "table sigue a foo"

A veces los patrones minimalistas pueden ser muy utiles. Imaginemos que deseamos encontrar todo lo que hay entre "foo" y "bar". Inicialmente podriamos haber escrito lo siguiente.

```
$_ =  "The food is under the bar in the barn.";
   if ( /foo(.*)bar/ ) {
       print "obtienes <$1>\n";
   }

Lo que nos da una respuesta inesperada del tipo:

obtienes < d is under the bar in the >
```

Esto ocurre porque .* es **voraz** (es decir: busca el mayor trozo posible que concuerde)**,** por eso se obtiene todo lo que esta entre el primer foo y el ultimo bar. En este caso es mas efectivo usar un patron minimalista para estar seguro de que obtenemos lo que esta entre el primer "foo" y el primer "bar"

```
if ( /foo(.*?)bar/ ) { print "obtienes <$1>\n" } obtienes <d is under the >
```

Veamos ahora otro ejemplo: imaginemos que quieres que el numero esté al final y solo al final de la cadena, y ademas quieres guardar la primera parte de la cadena para un uso posterior.

```
$_ = "Yo tengo 2 numeros: 53147";
   if ( /(.*)(\d*)/ ) {                                  # Mal!
       print "El principio es <$1>, el numero es <$2>.\n";
   }
```

Esto no funciona del todo, porque (de nuevo) .* es voraz y se come la mayor parte de la cadena. Ademas \d* puede concordar con el string vacio y la expresion regular seguiría concordando.

```
El principio es <Yo tengo 2: 53147>, numero es <>.
```

Aqui abajo pongo un par de variantes, la mayoria de ellas tampoco funcionan.

```
$_ = "Yo tengo 2 numeros: 53147";
   @pats = qw{
       (.*)(\d*)
       (.*)(\d+)
       (.*?)(\d*)
       (.*?)(\d+)
       (.*)(\d+)$
       (.*?)(\d+)$
       (.*)\b(\d+)$
       (.*\D)(\d+)$
   };
   for $pat (@pats) {
       printf "%-12s ", $pat;
       if ( /$pat/ ) {
           print "<$1> <$2>\n";
       } else {
           print "FALLO\n";
       }
   }
```

Lo que daria las siguientes salidas:

```
(.*)(\d*)      <>
(.*)(\d+)      <7>
(.*?)(\d*)    <> <>
(.*?)(\d+)     <2>
(.*)(\d+)$     <7>
(.*?)(\d+)$    <53147>
(.*)\b(\d+)$   <53147>
(.*\D)(\d+)$   <53147>
```

Como ya os habreis dado cuenta esto puede ser un poco peliagudo. Es importante darse cuenta de que la expresion regular es meramente un conjunto de aserciones de cuya defiicion se deduce el exito o fracaso. Los resultados pueden ser 0,1 y existen multitud de caminos diferentes para que una definicion sea o no exitosa con un string en particular. Y justamente por eso, justamente porque hay muchas formas de hacer que un patron concuerde correctamente, por eso necesitamos comprender el backtracking para saber de que distintas formas se ha llegado al exito o al fracaso con nuestro patron.

Usando aserciones -afirmaciones, condiciones- del tipo "mira alrededor" y negaciones de las aserciones podemos caer en multiples trampas. Imaginemos que buscamos las secuencias no numericas no seguidas por "123"

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {                          # Mal!
    print "Uep!, no 123 en $_\n";
}
```

Pero esto no esta yendo en camino hacia la concordancia, no es el camino que estamos esperando. Esta diciendo que 123 no esta en el string, cuando esta claro que si que esta en el string!. Abajo aparece una figura clarificadora

```
$x = 'ABC123' ;
$y = 'ABC445' ;
print "1: obtienes $1\n" if $x =~ /^(ABC)(?!123)/ ;
print "2: obtienes $1\n" if $y =~ /^(ABC)(?!123)/ ;
print "3: obtienes $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: obtienes $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

Esto devuelve

```
1: obtienes ABC
2: obtienes ABC
3: obtienes AB
4: obtienes ABC
```

Demonos cuenta de que el test 3 falla porque es justo el que tiene la version mas general del test 1. La diferencia mas importante entre ellos es que el test 3 contiene el cuantificador (\D*) y este puede ser usado en backtracking y en el test 1 no hay backtracking. ¿Que esta ocurriendo te estaras preguntando? Es cierto que al principio de $x, siguen 0 o mas no digitos, y luego tienes algo que no es 123? Si el patron de concordancia tiene \D* expande "ABC", esa expansion causa que el patron falle (recordemos la voracidad del *). El algoritmo de busqueda inicialmente hace concordar \D* con ABC. Seguidamente intenta hacer concordar (?!123 pero eso, evidentemente, falla. Pero falla porque cuando usamos \D* en una expresion regular, el algoritmo de busqueda puede hacer vuelta atras y puede intentar un nuevo camino con la esperanza de hacer concordar la expresion regular al completo. (y no conseguimos que se pare con la primera solucion que es lo que nos interesaria en este caso, porque despues de todo ... la primera solucion aunque nos satisface a nosotros no satisface la totalidad de la expresion regular.)

Ahora bien, el patron realmente, *realmente* quiere concordar, para ello usa el metodo estandar de guarda la parte que concordó y vuelve a intentarlo, \D* expande solo "AB" esta vez. Ahora mira que lo que sigue a "AB" no sea "123" y efectivamente ahora lo que sigue a "AB" es "C123", lo que basta para afirmar la concordancia del patrón.

Se puede intentar arreglar esto usando ambas aserciones y negaciones. Queremos decir que la primera parte en $1 debe ser seguida por un 1 digito, y en efecto, necesita se seguida por algo que no sea "123". Recordemos que las expresiones de "miraalrededor" son expresiones que pueden ser vacias, que simplemente "miran", pero que no consumen ningun parte del string en su prueba de concordancia. Por tanto reescribiendo en este sentido mostramos lo que se nos podria haber ocurrido. Sin embargo el caso 5 falla y el caso 6 es correcto:

```
print "5: obtiene $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: obtiene $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;
5: obtiene AB
6: obtiene ABC
```

En otras palabras, las 2 aserciones de tamaño cero -que no consumen string en su consulta- despues de cada una de los otras tareas funcionan a modo de AND, y si usas alguna asercion incorporada en el lenguaje como :/^$/ esto concordara solo si esta al principio de la linea Y (AND) al final de la linea simultaneamente. El profundo mensaje que encierra este parrafo es que las expresiones regulares siempre esperan el AND, excepto cuando uno explicitamente les dice que desea OR o bien usando la barra vertical. /ab/ espera encontrar una "a" Y (AND) seguidamente espera encontrar una "b". Esto es susceptible de muchas concordancias en posiciones diferentes porque "a" no es una asercion de longitud 0, sino una asercion de longitud 1. (es decir .. que consume un caracter)

Un AVISO: las expresiones regualres particularmente complicadas puden tomar un tiempo exponencial en su ejecucion debido al inmenso numero de soluciones y caminos que pueden llegar a encontrarse cuando de usa backtracking. Por ejemplo lo siguiente puede tomar un largo tiempo en ejecutarse

```
/((a{0,5}){0,5}){0,5}/
```

Y si ademas usaramos *'s sobre el limite entre 0 y 5 concordancias, esto podria literalmente durar toda la vida --o mas comunmente hasta que se acabe la memoria para la pila de tu ordenador.

No os perdais para mas información la pagina oficial de PERL:

- La Perl home page[2].

Translated by: Daniel R.C.
Nota: No pretendo llenar Bulma de articulos traducidos de otras páginas, pero hay ciertos materiales que me parecen de sumo interes para toda la comunidad. Creo que Bulma debe ofrecer buenos articulos de alto contenido tecnico y prefiero hacer una traduccion que intentar reescribir cosas que no domino ni la mitad de lo que lo domina el autor original.

Lista de enlaces de este artículo:

1. http://language.perl.com/all_about/back.html
2. http://bulma.net/index.html
3. mailto:tchrist@perl.com

E-mail del autor: danircJUBILANDOSEbulma.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=736