C'è sempre l'eccezione alla "regular"

29 novembre 2007 A cura di Marco Beri

Le regular expression sono probabilmente uno dei concetti più antichi ma ancora vivi e vegeti dell'ambito informatico. Furono infatti implementate per la prima volta da Ken Thompson nel 1966 nel suo editor QED e già allora si basavano sui regular set definiti nel 1950 dal matematico Stephen Kleen. Se avete mai usato il glorioso comando di linea grep forse saprete già che il suo nome è un acronimo che deriva da loro: "search Globally for lines matching the Regular Expression, and Print them" (ricerca globale e visualizzazione di righe corrispondenti alle espressioni regolari).

Da allora le regular expression hanno prosperato fino ad oggi e oramai ogni linguaggio che si rispetti non si può definire tale se non ha una libreria che permetta l'uso delle regex, come sono per comodità abbreviate tra gli addetti ai lavori. Perfino Visual Basic, storicamente meno avanzato rispetto ad altri linguaggi moderni, ha dovuto cedere alle loro lusinghe e il framework .NET le abbraccia introducendo una sintassi proprietaria, come spesso accade per tecnologie adottate da Microsoft. L'importanza delle Regular expression è tale che in alcuni linguaggi, come Ruby, Perl, Javascript queste sono implementate nativamente nella loro sintassi.

Qualche tempo fa, per motivi che ancora adesso mi sfuggono, la casa editrice Apogeo mi ha chiesto di scrivere un libro sulle regular expression. Non c'è mezzo migliore che dover spiegare nei dettagli un tema per scoprire quanto poco ne conosciamo. Infatti, quando mi sono trovato a scrivere un'appendice sulle implementazioni delle regular expression, mi sono subito accorto dell'esotericità e della complessità dell'argomento. Con sempre maggior sgomento ho cercato di comprendere come una regular expression possa essere mappata con un automa a stati finiti, come esistano due tipi di automi a stati finiti, deterministici (DFA) e non deterministici (NFA), come i secondi, anche se più ambigui, siano molto più facilmente costruibili a partire da una regex…

Facendola breve tutti i motori di regular expressioni usano gli NFA ma con due scuole di pensiero differenti: chiamiamole quella dei multistati e quella del backtracking. Siccome i percorsi di un diagramma NFA possono presentare dei bivi dubbi (da qui la definizione "automa a stato finito non deterministico"), i motori multistato percorrono contemporaneamente tutte le strade possibili mentre i motori backtracking scelgono una strada e in presenza di un vicolo cieco tornano sui propri passi.

La particolarità di questi due approcci è che uno, quello del backtracking, è estremamente efficiente e più veloce nella stragrande maggioranza dei casi mentre l'altro, quello dei multistati, è normalmente molto più lento ma in alcuni casi rarissimi e del tutto particolari surclassa clamorosamente l'avversario. Praticamente tutti i linguaggi attuali adottano la scuola del backtracking anche perché quest'ultima permette l'uso di regular expression con caratteristiche più avanzate rispetto all'altra.

Se siete riusciti ad arrivare a leggere sino a qui senza addormentarvi sicuramente vi state domandando quello che anche io mi sono chiesto fino a qualche giorno fa: sì, va bene e allora? Le regex funzionano bene, sono comode e non mi importano per nulla tutte queste definizioni e tecnicismi. Cosa è successo qualche giorno fa che mi ha fatto cambiare idea? Il mio amico Carlo C8E Miron mi ha sottoposto un problema stranissimo che stava capitando a Ludovico Magnocavallo, il deus ex machina di BlogBabel. Il suo crawler di blog, dopo più di un anno di onorato servizio, più di 10.000 blog e quasi un milione e mezzo di articoli processati, improvvisamente aveva deciso di bloccarsi occupando per più di 8 ore il 100% della CPU del server senza dare segni di vita. Ludo aveva già abilmente circoscritto il problema alla regular expression che estraeva i link all'interno di un post e aveva dedotto si trattasse di un bug della libreria re di Python, il meraviglioso linguaggio da lui utilizzato per il cuore di BlogBabel. Per la cronaca i link estratti dal crawler erano già arrivati a più di 2.750.000, quindi cosa poteva mai essere successo se non un caso particolare che evidenziava un baco delle librerie? Ebbene no! Nessun baco ma proprio quello che state immaginando: uno di quei rarissimi casi in cui il motore a backtracking si comporta male, anzi, malissimo! Il crawler non era "piantato" ma girava avanti e indietro alla disperata ricerca di una via di uscita nel diagramma NFA corrispondente alla regex di ricerca dei link. Riscrivere una parte della regular expression per aggirare il caso particolare ha risolto il problema.

Qual è la morale? Possiamo forse riassumerla con una storiella: un tizio aveva un problema e disse "So come risolverlo: userò una regular expression!" e così in un colpo solo ebbe due problemi… Scherzi a parte non è certo così, ma una cosa la si può davvero dire: non c'è niente di scontato nell'informatica, nemmeno nelle ultracinquantenarie regular expression.

Approfondimenti

La sintassi proprietaria introdotta da .NET riguarda i gruppi con nome. Normalmente questi sono definiti con `(?P<nome>regex)` e riusati con `(?P=nome)`. .NET invece usa per gli stessi scopi `(?<nome>regex)` e `\k<nome>` e inoltre introduce una seconda sintassi `(?'nome'regex)` e `\k'nome'`.

Un articolo che affronta in dettaglio con grafici e misurazioni la questione delle due scuole e descrive cosa siano gli automi DFA e NFA è stato scritto da Russ Cox e può essere consultato all'indirizzo http://swtch.com/~rsc/regexp/regexp1.html.

La storiella è in realtà una citazione di una famosa frase di *Jamie Zawinski*: «"Some people, when confronted with a problem, think «I know, I'll use regular expressions.» Now they have two problems".» La frase è diventata praticamente una meta-frase in cui al posto di "regular expressions" di volta in volta si inserisce "XML", "threads" e via dicendo a seconda di cosa si vuole criticare. L'origine della frase è controversa e il primo riferimento ad essa risale al 1998 in un messaggio di comp.lang.python.

La regex incriminata è la seguente:

```
<a.*?href=["\'](.*?)["\'].*?>(.*?)</a'
```

Con queste poche righe in Python possiamo riprodurre il problema incontrato da Ludo:

```
import re, timeit
r= re.compile( r'<a.*?href=["\'](.*?)["\'].*?(.*?)</a',
     re.M | re.S | re.I | re.U)
def cerca(testo):
  r.findall(testo)
for i in range(1, 20):
  n = max(1, 10 * i)
  testo = ( """<a id="blah"></a>""" +
   """<v:imagedata href=" http://example.org?blablah.png" />"""
```


10 consigli per interagire
geek di famiglia

Amazon lancia il Kindle i

IBM rilascia la versione 9
Express-C

Considerazioni sulla scar
Italia

COMMENTI RECENTI

Leonardo su 10 consigli
il proprio geek di famigli

Gilton su 10 consigli per
proprio geek di famiglia

Lorenzo su La fluidità de

Lorenzo su 10 consigli pe
proprio geek di famiglia

Mr.Price su 10 consigli p
proprio geek di famiglia

CATEGORIE

Applicazioni & OS (13)

Editoriali (11)

Gadget (7)

IT Business (25)

Legge & Privacy (8)

Libri (6)

Networking & Security (1

Programmazione (104)

Siamo tutti geek (32)

Software Engineering (9)

Tlc & Internet (17)

TAG POPOLARI

agile algoritmi amazon ap
conferenze django ev
programming firefox framew
internet ironruby italia java
linguaggi linux lisp m
microsoft mozilla netw
Programmazione project
python rails rest

```
    * n )
  t = timeit.Timer(
        "cerca('%s ')" % testo,
        "from __main__ import cerca")
  print  "% 4d% 5d%.2f " % (n, len(testo), t.timeit(1))
```

Eseguendole ecco il risultato:

Passo	Lunghezza	Secondi
10	547	0.01
20	1077	0.09
30	1607	0.40
40	2137	1.23
50	2667	3.10
60	3197	6.43
70	3727	12.07
80	4257	19.74
90	4787	32.52
100	5317	48.20
110	5847	70.44
120	6377	98.54
130	6907	136.39
140	7437	181.57
150	7967	239.89
160	8497	308.50
170	9027	400.57
180	9557	526.81
190	10087	636.33
200	10617	866.52

Tabella 1. Tempi di esecuzione

La stringa del passo 200 è lunga 10617 caratteri mentre la stringa che Blogbabel ha cercato di processare era lunga 61883 caratteri. Potete vedere qui sotto il grafico risultante, il quale lascia intuire che probabilmente avrebbe fatto prima il sole a spegnersi…

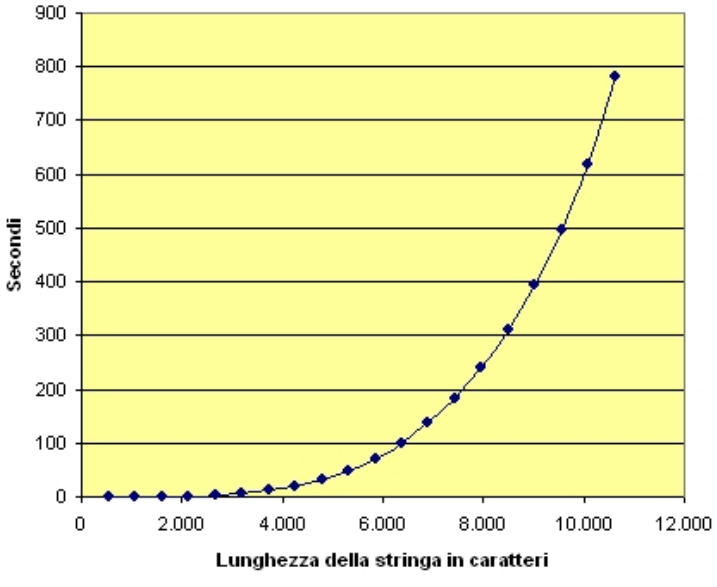

Figura 1. Grafico dei tempi di esecuzione

La versione leggermente modificata della regex che riesce a elaborare istantaneamente la stringa lunga 61883 caratteri è questa:

```
<a[^>]+href=(["\'])(.*?)\1(?:\s+[^>]*)?>([^<]*)</a
```

Il problema viene risolto grazie a questi due "pezzetti" di regex che prevengono il continuo backtracking:

```
[^>]*
```

```
[^<]*
```

Inoltre con il gruppo di backreference `\1` si evita che dei link come il seguente, delimitati da doppie virgolette e contenenti un apice, non vengano individuati correttamente:

```
<a href="http://example.org?query='>123">pippo</a>
```

Filed Under: Programmazione Tagged With: blogbabel, regex

About Marco Beri

Marco Beri si laurea in Scienze dell'Informazione nel 1990, periodo oramai definibile come la preistoria del settore. Il computer è prima di tutto un suo hobby e anche per questo si innamora di Python a prima vista nel lontano 1999, dopo aver sperimentato una ventina di altri linguaggi. Fa di tutto, riuscendoci, per portarlo nella sua azienda, la Link I.T. spa, dove dal 1997 occupa il ruolo di amministratore e responsabile dello sviluppo software. Riesce perfino a intrufolarsi come amministratore nella fondazione dell'associazione Python Italia. Incredibilmente pubblica anche diversi libri per Apogeo/Feltrinelli.

Comments

Folletto says:
17 dicembre 2007 at 01:24

Ok, ero proprio curioso di sapere cosa avesse causato il problema che citava Ludo. 🙂

Grazie per l'approfondimento. 😉

Simbul says:
17 dicembre 2007 at 09:49

Piuttosto tecnico e circoscritto ad un caso molto particolare, ma nondimeno estremamente interessante 😉

Si conferma la pericolosità del .* nelle regexp 😀

Marcob says:
17 dicembre 2007 at 10:09

Simbul: eh già. Comunque il vero pitfall in questo caso è stato .*?

Fabrizio says:


Carmine Noviello


Claudio Cicali (RS


Alberto Revelli (R


Roberto De Ioris (


Davide Casali (RS


Franco Lombardo


Nicola Larosa (RS


Lorenzo Bolognin


Massimiliano Mir


Emmanuele Somi


Francesco Matara


Marco De Paoli (R


Massimo Scamar

22 dicembre 2007 at 14:46

Articolo interessante. Uso le regular expression saltuariamente per risolvere casi semplici. Regex è complesso perchè può fare cose complesse.
Alla fine spacco il problema in ricerche più mirate. E' vero che faccio lavorare la macchina 2 volte sulla stessa stringa. Ma almeno capisco che diavolo fa.

PS: Voglio fare un complimento a Stacktrace (al team e a tutti quelli che ci stanno attorno) siete proprio bravi. Complimenti!!

Marcob says:
22 dicembre 2007 at 17:04

Fabrizio: grazie! L'impegno di tutti è stato (ed è) massimo, per cui ci fa davvero molto piacere vedere che qualcuno lo riconosce.

Policy per i commenti: Apprezzo moltissimo i vostri commenti, critiche incluse. Per evitare spam e troll, e far rimanere il discorso civile, i commenti sono moderati e prontamente approvati poco dopo il loro invio.