Regex Debugging

Debugging von Regulären Ausdrücken

In Ausgabe 7 hat Thomas Fahle schon eine Einführung in den Debugger für Perl-Programme gegeben – an dieser Stelle nun eine Einführung in das Debugging von Regulären Ausdrücken. Ein paar hilfreiche Module in Sachen "Reguläre Ausdrücke" wurden bereits in Ausgabe 4 vorgestellt.

Eine Sache die man nicht gerne macht, ist das Debuggen von Regulären Ausdrücken. Bei einfachen RegEx lässt sich das noch gut "per Hand" machen, aber sobald es etwas komplexer wird, ist man verloren.

Die RegEx-Engine von Perl hat einen Debug-Modus, mit dem man sich anschauen kann, was bei einem Regulären Ausdruck für einen Eingabestring passiert.

Allerdings muss man sagen, dass Regex-Debugging nicht die einfachste Aufgabe ist. Den Debugger kann man mit `-Mre=debug` im Aufruf des Perl-Interpreters anschalten:

```
perl -Mre=debug -e '"foobar" =~ /regex/'
```

Fangen wir ganz einfach an - Listing 1.

Gehen wir das ganze Schritt für Schritt durch:

Als erstes wird ein alter Reguläre Ausdruck verworfen, danach wird der zu testende Reguläre Ausdruck kompiliert. Die Regex-Engine ist sehr komplex. Für Perl 5.10 wurde die komplette Regex-Engine neu geschrieben und das Konzept dahinter wurde umgeworfen. Damit sind jetzt auch komplexere Reguläre Ausdrücke schneller.

```
'foobar' =~ /(.)b/; # ein beliebiges Zeichen vor 'b'

Freeing REx: `","'
Compiling REx `(.)b'
size 8 Got 68 bytes for offset annotations.
first at 3
   1: OPEN1(3)
   3:   REG_ANY(4)
   4: CLOSE1(6)
   6: EXACT <b>(8)
   8: END(0)
anchored "b" at 1 (checking anchored) minlen 2
Offsets: [8]
        1[1] 0[0] 2[1] 3[1] 0[0] 4[1] 0[0] 5[0]
Guessing start of match, REx "(.)b" against "foobar"...
Found anchored substr "b" at offset 3...
Starting position does not contradict /^/m...
Guessed: match at offset 2
Matching REx "(.)b" against "obar"
  Setting an EVAL scope, savestack=3
   2 <fo> <obar>          |  1:  OPEN1
   2 <fo> <obar>          |  3:  REG_ANY
   3 <foo> <bar>          |  4:  CLOSE1
   3 <foo> <bar>          |  6:  EXACT <b>
   4 <foob> <ar>          |  8:  END
Match successful!
Freeing REx: `"(.)b"'
```

Listing 1

Die Größenange zeigt, wie groß die kompilierte Form des Regulären Ausdrucks ist. In der Regel ist die Einheit 4-byte-Worte. Danach wird die Größe der Offset/Längen-Tabelle für den Regex ausgegeben. Auf diese Tabelle komme ich gleich noch einmal zu sprechen.

`first at 3` gibt an, welcher Knoten des Regulären Ausdrucks der erste Knoten ist, der einen Match versucht. Die Knoten werden dann in den Zeilen dargestellt:

```
1: OPEN1(3)
3:   REG_ANY(4)
4: CLOSE1(6)
6: EXACT <b>(8)
8: END(0)
```

Der Aufbau ist wie folgt:

```
ID: TYP(NÄCHSTER_KNOTEN)
```

Ich werde nicht auf alle Typen eingehen, die es gibt. Die sind in der Dokumentation `perldebguts` beschrieben. Auf das Beispielprogramm werde ich dennoch eingehen. `OPEN#` zeigt an, dass hier eine Gruppierung vorliegt, die die Treffer auch speichert. Die Nummer gibt an, die wievielte Gruppierung das ist. Würde eine "non-capturing group" verwendet werden, also `(?:.)` würde das `OPEN1` und das `CLOSE1` komplett wegfallen.

In der Klammer nach dem Typ wird die ID des Knotens gespeichert, zu dem gesprungen werden soll, wenn der Ursprungsknoten matcht. In diesem Beispiel ist das immer der nächste Knoten. Ich werde nachher noch ein Beispiel zeigen, bei dem das nicht der Fall ist.

`REG_ANY` bedeutet, dass ein beliebiges Zeichen gematcht werden soll. Nach dem Typ `EXACT` wird noch angegeben, welcher String literal gematcht werden soll.

Nach der Darstellung des kompilierten Programms kommen ein paar Angaben zu Optimierungen und Heuristiken - Listing 2.

In diesem Fall weiß die Regex-Engine, dass es ein literales `b` in dem String mit dem Offset 1 geben muss und dass der zu durchsuchende String mindestens 2 Zeichen lang sein muss.

Danach wird die Offset/Längen-Tabelle angezeigt. Zuerst wird die Anzahl der Element in der Tabelle angegeben; hier sind es 8 Elemente. Dann folgt der Inhalt der Tabelle. Die Zählung der Elemente fängt bei 1 an und der Index ist gleich der ID der Knoten im kompilierten Programm. Die Elemente werden in der Schreibweise `offset[länge]` notiert.

Ist ein Element `0[0]`, dann existiert kein Knoten dafür. Wir haben in der kompilierten Form gesehen, dass es keinen Knoten mit der ID 2 gibt, also ist das zweite Element der Tabelle ein `0[0]`.

Das erste Element (hier: `1[1]`) repräsentiert also den Knoten mit der ID 1 (`OPEN1`). Der Offset gibt an, an welcher Stelle im originalen (nicht-kompilierten) Regulären Ausdruck dieser Knoten beginnt und die Länge gibt an, wieviele Zeichen der Knoten repräsentiert. Ich denke, es ist klar ersichtlich, dass das "(" am Anfang des Regulären Ausdrucks (`(.)b`) steht und 1 Zeichen lang ist.

Nach dieser Tabelle werden hier noch angezeigt, dass die Regex-Engine versucht zu raten, wo der Reguläre Ausdruck in dem zu durchsuchenden String anfängt. Hier findet die Engine heraus, dass "b" der vierte Buchstabe des Strings ist ("offset 3") und rät, dass der Treffer beim dritten Buchstaben anfängt ("offset 2").

Bis zu dieser Stelle waren Ausgaben, die de Kompile-Zeit des Regulären Ausdrucks betreffen.

Ab jetzt folgen Ausgaben, die die Laufzeit des Regulären Ausdrucks betreffen. Sollte so eine Ausgabe fehlen, bedeutet das, dass der Reguläre Ausdruck erst gar nicht ausgeführt wird, z.B. weil der String zu kurz ist oder ein fester Substring erst gar nicht im String enthalten ist (z.B. bei `'hallo' =~ /es/`.

```
anchored "b" at 1 (checking anchored) minlen 2
Offsets: [8]
        1[1] 0[0] 2[1] 3[1] 0[0] 4[1] 0[0] 5[0]
Guessing start of match, REx "(.)b" against "foobar"...
Found anchored substr "b" at offset 3...
Starting position does not contradict /^/m...
Guessed: match at offset 2
```

Listing 2

Aber zurück zum Beispiel:

```
Matching REx "(.)b" against "obar"
  Setting an EVAL scope, savestack=3
    2 <fo> <obar>          |  1:  OPEN1
    2 <fo> <obar>          |  3:  REG_ANY
    3 <foo> <bar>          |  4:  CLOSE1
    3 <foo> <bar>          |  6:  EXACT <b>
    4 <foob> <ar>          |  8:  END
```

Es wird angezeigt, gegen welchen String der Regex ausgeführt wird. Hier sieht man auch, welche Auswirkungen das Raten der Engine hat. Es wird nämlich nicht "foobar" sondern "obar" genommen. Das spart Zeit.

Danach sieht man den Ablauf des Regulären Ausdrucks. Die Zeilen sind wie folgt aufgebaut:

```
STRING_OFFSET <PRE-STRING> <POST-STRING>
                                  | ID: TYP
```

`STRING_OFFSET` ist die Stelle im zu durchsuchenden String, `PRE-STRING` der schon abgearbeitete Teil des Strings und `POST-STRING` der noch zu durchsuchende Teil. Danach kommt die Angabe, welcher Knoten aus dem kompilierten Regulären Ausdrucks an dieser Stelle genommen wird.

Backtracking

Backtracking ist ein Thema, das nicht wirklich leicht ist und man kann leicht in die Backtracking-Falle tappen. Es ist am einfachsten, Backtracking an einem Beispiel zu zeigen. Dazu verwenden wir wieder das Debugging für Reguläre Ausdrücke.

Was ist Backtracking?

Am besten kann man Backtracking mit einem Bildlichen Vergleich beschreiben: Man stelle sich eine Schnitzeljagd durch die Gegend vor. Es geht von einem Punkt zum nächsten. Es gibt eine Beschreibung, wie es nach jedem Punkt weiter geht. Ein Regulärer Ausdruck ist diese Beschreibung. Bei machen Zwischenpunkten auf der Schnitzeljagd wird es nur eine Möglichkeit geben, wie man zu dem nächsten Zwischenziel kommt, bei anderen wird es verschiedene Möglichkeiten geben. Ziel ist es natürlich, bis ans Ende zu kommen. Befindet man sich an einem Zwischenziel mit mehreren Möglichkeiten, muss man sich erst für eine Möglichkeit entscheiden. Diesen Weg geht man dann. Unterwegs stellt man fest, dass man so nicht mehr weiter kommt. Also geht man zurück bis zum letzten erfolgreichen Zwischenziel und versucht die nächste Möglichkeit. Das nennt man Backtracking. Wenn man bei einem Zwischenziel alle Möglichkeiten ausgeschöpft hat und nicht mehr weiterkommt, muss man eventuell auch zwei oder mehr Zwischenschritte zurückgehen. Das kann Zeit kosten.

Als Beispiel

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

1. Beginne mit dem ersten Buchstaben im String ('a')

2. Probiere die erste Alternative in der ersten Gruppe ('abd')

3. Matche 'a' gefolgt von einem 'b'. Soweit so gut.

4. Das 'd' im Regulärer Ausdruck matcht nicht das 'c' im String - eine Sackgasse. Also gehe zwei Buchstaben zurück (Backtracking) und probiere die zweite Alternative in der ersten Gruppe ('abc').

5. Matche 'a' gefolgt von einem 'b' gefolgt von einem 'c'. Wir sind im Plan und haben die erste Gruppe erledigt. Setze `$1` auf 'abc'.

6. Gehe zur zweiten Gruppe und probiere die erste Alternative ('df').

7. Matche das 'd'.

8. Das 'f' im Regulären Ausdruck match nicht das 'e' im String, also eine Sackgasse. Gehe einen Buchstaben zurück (Backtracking) und probiere die zweite Alternative in der zweiten Gruppe ('d').

9. Das 'd' matcht. Die zweite Gruppe ist erfolgreich, also setze `$2` auf 'd'.

10. Wir sind am Ende des Regulären Ausdrucks, also sind wir fertig! Wir haben 'abcd' im String 'abcde' gematcht.

Das Beispiel in Listing 3 zeigt nochmal in der Debug-Ausgabe, wobei die Regex-Engine dabei schon einiges optimiert und Strings zusammenfasst:

```
C:\>perl -Mre=debug -e "'abcde' =~ /(abd|abc)(df|d|de)/;"
Freeing REx: `","'
Compiling REx `(abd|abc)(df|d|de)'
size 24 Got 196 bytes for offset annotations.
first at 3
   1: OPEN1(3)
   3:   BRANCH(6)
   4:     EXACT <abd>(9)
   6:   BRANCH(9)
   7:     EXACT <abc>(9)
   9: CLOSE1(11)
  11: OPEN2(13)
  13:   BRANCH(16)
  14:     EXACT <df>(22)
  16:   BRANCH(19)
  17:     EXACT <d>(22)
  19:   BRANCH(22)
  20:     EXACT <de>(22)
  22: CLOSE2(24)
  24: END(0)
minlen 4
Offsets: [24]
        1[1] 0[0] 1[1] 2[3] 0[0] 5[1] 6[3] 0[0] 9[1] 0[0] 10[1] 0[0] 10[1] 11[2]
 0[0] 13[1] 14[1] 0[0] 15[1] 16[2] 0[0] 18[1] 0[0] 19[0]
Matching REx "(abd|abc)(df|d|de)" against "abcde"
  Setting an EVAL scope, savestack=3
   0 <> <abcde>            |  1:  OPEN1
   0 <> <abcde>            |  3:  BRANCH
  Setting an EVAL scope, savestack=14
   0 <> <abcde>            |  4:    EXACT <abd>
                               failed...
   0 <> <abcde>            |  7:    EXACT <abc>
   3 <abc> <de>            |  9:    CLOSE1
   3 <abc> <de>            | 11:    OPEN2
   3 <abc> <de>            | 13:    BRANCH
  Setting an EVAL scope, savestack=25
   3 <abc> <de>            | 14:      EXACT <df>
                                  failed...
   3 <abc> <de>            | 17:      EXACT <d>
   4 <abcd> <e>            | 22:      CLOSE2
   4 <abcd> <e>            | 24:      END
Match successful!
Freeing REx: `"(abd|abc)(df|d|de)"'
```

Listing 3

Hier sieht man schon, dass die Regex-Engine viel arbeiten muss. In diesem Beispiel war es noch ein relativ einfacher Regulärer Ausdruck.

Renée Bäcker