

Dao Regex Tutorial

Fu Limin (phoolimin@gmail.com)

September 19, 2009

Contents

1	Introduction	1
2	Character Class	1
3	Pattern Item	3
4	Grouping and Captures	3
5	String Methods for Regex	4
6	Examples	6

1 Introduction

A regular expression (*regex*

Starting from the latest release, the functionalities of regex can be accessed by string methods. The previous perl-style regex has been removed (as a way to simplify the language and implementation). The syntax of the new regex is similar to that of Lua, but most of the character class names are different.

2 Character Class

A character class is used to identify a set of characters.

- **x**: ordinary characters represent themselves, excluding magic characters `^$0%.[]*+~?{}<>`
;

- **.** : a dot represents any characters;
- **%a** : all alphabetic characters;
- **%s** : all white space characters;
- **%k** : all control characters;
- **%p** : all punctuation characters;
- **%d** : all digits;
- **%x** : all hexadecimal digits;
- **%c** : all lower case characters;
- **%w** : all alphabetic characters, digits and character '_';
- **%A** : non alphabetic characters, complement of **%a** ;
- **%S** : non white space characters, complement of **%s** ;
- **%K** : non control characters, complement of **%k** ;
- **%P** : non punctuation characters, complement of **%p** ;
- **%D** : non digits, complement of **%d** ;
- **%X** : non hexadecimal digits, complement of **%x** ;
- **%C** : upper case characters;
- **%W** : complement of **%w** ;
- **%x** : represents character **x** , where **x** is any non-alphanumeric character; **x** may also be an alphabetic character if it is not one of the character class symbols or **b** or **B** .
- **[set]** : represents the union of all characters in **set** ; a range of characters starting from a character **x** up to another character **y** can be included in **set** as **x-y** ; the above character classes can also be included in **set** ;
- **[^set]** : complement of **[set]** ;

3 Pattern Item

A pattern item can be

- a single character class;
- \wedge : match at the begin of a string;
- $\$$: match at the end of a string;
- $\%n$: match n -th captured sub string; n can be one or more digits;
- $\%bxy$: match a balanced pair of characters x and y ; here balance means, starting from the same matched position, the mached sub string should contain the same number and minimum number of x and y ; the same as that in Lua;
- $\%B\{\text{pattern1}\}\{\text{pattern2}\}$: match a balanced pair of patterns **pattern1** and **pattern2** ; here balance has the same meaning as in $\%bxy$;

A pattern item e can be optional skipped or matched repeatedly as indicated by:

- $e?$: match zero time or once;
- e^* : match zero time or any number of times;
- $e+$: match once or more;
- $e\{n\}$: match exactly n times;
- $e\{n,\}$: match at least n times;
- $e\{,n\}$: match at most n times;
- $e\{n,m\}$: match at least n times and at most m times;

4 Grouping and Captures

In a pattern, one or more pattern items can be grouped together by parenthesis to form sub patterns (group). Alternative patterns in a group can be separated by $|$, and the group could be optionally skipped if an empty alternative pattern is specified as $(|pattern)$ or $(pattern|)$. When a string is matched to a pattern, the sub strings that match the groups of sub patterns can be captured for other use. Captures are numbered according to their left parenthesis. For example, in pattern $(\%a+)\%s*(\%d+(\%a+))$, the first $(\%a+)$ will have group number 1, and $(\%d+(\%a+))$ will have group number 2, and the second $(\%a+)$ will have group number 3. For convenience, the whole pattern has group number 0.

In case there are multiple possible ways of matching a substring starting from the same position, the matching length is calculated as the sum of the lengths of the sub-matches of all groups (including number 0 group) in the pattern, and the matching giving maximum matching length is returned as the result. In this way, one can put a deeper nesting of parenthesis around a group, if one want that group has high priority to be matched. For example, when *1a2* is matched to patterh `(%d%w*)(%w*%d)` , there are two possible ways of macthing, namely, *1a* matching to `(%d%w*)` and *2* matching to `(%w*%d)` , or *1* matching to `(%d%w*)` and *a2* matching to `(%w*%d)` , but if an extra parenthesis is added to one of the group, for example, as `(%d%w*)((%w*%d))` , then the matching becomes unique, which is the second way of matching where letter *a* is matched in the last group.

5 String Methods for Regex

Like in Lua, the regular expression matching functionalities are accessed through various string methods. The regular expression patterns are stored in strings, and passed to these string methods. Each pattern string corresponds to an internal representation of a regular expression, which are compiled from the pattern string at the first time it is used. Though the strings that represent the same pattern can be passed multiple times to these methods, they are compiled once in one process (virtual machine process). So the overhead of compiling a regular expression can be normally ignored.

5.0.1 pfind(): find regular expression pattern

```
string.pfind( pt : string, index=0, start=0, end=0 )=>list<tuple<int,int> >
```

This method searches for the position(s) of substring(s) that match(es) to the pattern represented by *pt* . If *index* is greater than zero, search for the *index* -th occurrence of the matched substring, otherwise, search for all. The searching starts from position *start* , and ends at *end* if it is greater than zero. Zero value for *end* indicates searching until the end of the string.

5.0.2 match(): match substring

```
string.match( pt : string, start=0, end=0, substring=1 )  
=>tuple<start:int,end:int,substring:string>
```

Find a substring that matches to *pt* , starting from position *start* until position *end* . Also extract the matched substring if parameter *substring* is not zero;

5.0.3 extract(): extract substrings

```
string.extract( pt : string, matched=1, mask='', rev=0 )=>list<string>
```

If only parameter *pt* is presented, this method will extract the substrings that match to the pattern represented by string *pt* . If *matched* is positive, the matched substrings are returned; and if it is negative, the complement substrings of the matched regions are returned; and when *matched=0* , both the matched substrings and the complement substrings are returned.

One can also set the *mask* pattern, to restrict the searching of *pt* within the substrings that match to *mask* if *rev* is set to zero, otherwise, the searching will be restricted outside of *mask* -matched regions.

5.0.4 capture(): capture groups

```
string.capture( pt : string, start=0, end=0 )=>list<string>
```

Search for a substring that matches to *pt* , and capture the substrings that match to pattern groups. The captured substrings are returned as a list, where the *i*-th item is substring that matches to the *i*-th group.

5.0.5 change(): change substrings

```
string.change( pt : string, s : string, index=0, start=0, end=0 )=>int
```

Replace the substrings that match to *pt* with string *s* , which may contain reference to captured substrings for groups, for example, if *s = 'abc%1'* , it means replacing with a string which is a concatenation of *abc* and the substring that match to the first group of *pt* .

If *index* is positive, replace the *index* -th occurrence of the substring matching to *pt* . The substitution can also be restricted within a region starting from position *start* until position *end* .

The number of occurrence of substitution is returned.

6 Examples

```
s = 'abc123def456';

stdio.println( s.pfind( '%d+' ) );
stdio.println( s.match( '%d+' ) );
stdio.println( s.extract( '%d+' ) );
stdio.println( s.extract( '%d+', -1 ) );
stdio.println( s.extract( '%d+', 0 ) );
stdio.println( s.capture( '(%a+)(%d+)' ) );

s.change( '(%a+)', '==%1==' );
stdio.println( s );

stdio.println( s.extract( '%d+', 1, '==%d+==' ) );
```

Output

```
{ ( 3, 5 ), ( 9, 11 ) }
( 3, 5, 123 )
{ 123, 456 }
{ abc, def }
{ abc, 123, def, 456 }
{ abc123, abc, 123 }
==abc==123==def==456
{ 123 }
```