Замена строк

Применение регулярных выражений в конструкциях Perl

Регулярные выражения применяются во многих конструкциях.

- В функции split() первым параметром может использоваться регулярное выражение, которое будет служить для поиска разделителей при разделении строки на части.

Так, например, можно разбить строку на подстроки по любому из пробельных символов:

```
@substrings = split /\s+/, $text; # разбить на части
```

Регулярные выражения часто применяются в функциях, работающих с массивами для фильтрации нужных элементов.

- Функция grep()

Синтаксис:

```
grep EXPR,LIST
```

похожа на map, но в первом параметре проверяется выполнение некоторого условия, и если оно выполняется, то соответствующий элемент массива попадает в итоговый массив:

Использование без рег. выражения:

@numbers = (1,-2,-3,4,5);

@square = grep($_>0, @numbers);

print "@square "; # печатает 1 4 5

Использование с рег. выражением:

Функция grep() возвратит список элементов массива, соответствующих указанному образцу:

```
@result = grep /$pattern/, @source; # отобрать элементы
```

- С помощью функции map можно применить операцию замены в соответствии с регулярным выражением ко всем элементам массива.

Пример:

```
@hrefs = ('http://regex.info', 'http://regexp.ru');
map s{http://}{}, @hrefs;        # убрать 'http://' из ссылок
```

Пример:

Допустим нужно разделить записи на блоки, разделенные четырьмя пробелами:

```
@probel = map m!\s{4}!, split /\n/, $test;
```

==========

Модификаторы поиска ======= /x

/x - использовать расширенный синтаксис регулярных выражений (eXtended) -- Режим свободного форматирования.

В этом режиме пропуски за пределами символьных классов в основном игнорируются. Учитываются пропуски внутри символьных классов (кроме java.util.regex), а между # и концом строки могут находиться комментарии.

/x позволяет записать регулярное выражение в виде:

```
$text =~ s{
      \b
      # Сохранить адрес в $1...
      (
            имя_пользователя
            \@
            имя_хоста
      )
      \b
}{<a href="mailto:$1">$1</a>}gix;
```

Модификатор /x завершает этот фрагмент (вместе с модификаторами /g и /i) и обеспечивает две простые, но чрезвычайно полезные возможности. Во-первых, «посторонние» пропуски игнорируются, что позволяет представить регулярное выражение в формате, удобном для чтения. Во-вторых, в регулярное выражение могут включаться комментарии, начинающиеся с префикса #.

Выражаясь точнее, модификатор /x превращает пропуски в игнорируемые метасимволы, а символ # – в метасимвол, который означает: «игнорируйте меня и весь текст до следующего символа новой строки»

Модификаторы поиска ======= /s

```
$str =~ m/regexp/modifier;
```

/s - рассматривать текст как одну строку (Single-line) -- режим совпадения метасимвола точки со всеми символами (многострочный поиск); обычно "точка" не совпадает с \n ;

Пример.

```
$str="The aim is future.\n And who are  you?";

if($str=~/. A/s)
{
print " OK!\n";
}
else
{
print "NO\n";
}
```

Модификаторы поиска ======= /m

/m - расширенный режим привязки к границам строк (Multi-line) с учетом \n ;

Будем считать, что конец абзаца обозначается пустой строкой. Существует несколько способов идентификации пустой строки. Сначала может возникнуть идея использовать конструкцию вида

$text =~ s/^$/<p>/g;

Фактически условие поиска означает: «Найти позицию начала строки, сразу же после которой следует позиция конца строки». Действительно, подобное решение годится для программ, которые всегда работают с текстом, состоящим из одной логической строки.

- Метасимволы ^ и $ обычно относятся не к *логическим строкам*, а к абсолютным позициям начала и конца рассматриваемого *текста*. Следовательно, если целевая строка содержит несколько логических строк, придется поискать решение работы со строками.

К счастью, в большинстве языков с поддержкой регулярных выражений существует простое решение – *расширенный режим привязки*, в котором метасимволы ^ и $ применяются именно к *логическим строкам*. В языке Perl этот режим активизируется модификатором /m:

```
$text =~ s/^$/<p>/mg;
```

Обратите внимание на слияние модификаторов /m и /g (при указании нескольких модификаторов вы можете объединять их в произвольном порядке).

Пример:

```
$str="The our is future.\n And who are  you?";  ) {
if($str=~/^ /) {
print " OK!\n";
}
else {
print "NO\n";
}
```

Пример на /m и /g. Рассмотрим пример команды с несколькими парами сохраняющих скобок.

Предположим, весь текст почтового ящика Unix хранится в одной переменной, содержимое которой разбито на логические строки вида:

```
alias Jeff          jfriedl@regex.info
alias Perlbug       perl5_porters@perl.org
alias Prez          president@whitehouse.gov
```

Для извлечения псевдонимов и адресов из одной логической строки можно воспользоваться командой

`m/^alias\s+(\S+)\s+(.+)/m (без модификатора /g).`

В списковом контексте команда вернет список из двух элементов ('Jeff', 'jfriedl@regex.info'). Чтобы найти все подобные пары, мы добавим модификатор /g. Команда вернет список вида:

```
('Jeff', 'jfriedl@regex.info', 'Perlbug',
'perl5_porters@perl.org', 'Prez', 'president@whitehouse.gov' )
```

Если возвращаемые элементы образуют пары «ключ/значение», как в приведенном примере, результат можно присвоить ассоциативному массиву (хешу). После выполнения команды

```
my %alias = $text =~ m/^alias\s+(\S+)\s+(.+)/mg;
```

полный адрес Jeff доступен через элемент хеша `$alias{Jeff}`.

Модификаторы поиска ======= /o

/o - один раз откомпилировать регулярное выражение (Once);

Пример.

Для формирования регулярного выражения можно использовать переменные Perl. Как, например, в следующем примере:

```
$sign='[+-]?';

$digits='\d+';

$decimal ='\.?';

$more_digits='\d*';

$number="$sign$digits$decimal$more_digits";

$num=10.23;

if($num=~m/^$number$/o)
{
print "OK!\n";
}
else
{
print "No match\n";
}
```

Интерактивный поиск – скалярный контекст с модификатором /g

1) Скалярный контекст m/…/g представляет собой специальную конструкцию, заметно отличающуюся от других ситуаций.

- Как и обычный оператор m/…/, он находит только одно совпадение, но
- по аналогии со списковым оператором m/…/g запоминает позицию предыдущего совпадения.
- При каждом выполнении m/…/g в скалярном контексте находится «следующее» совпадение.
- Когда очередной поиск завершится неудачей, следующая проверка снова начинается с начала строки.

Простой пример:

```
$text = "WOW! This is a SILLY test.";

$text =~ m/\b([a-z]+\b)/g;

print "The first all_lowercase word: $1\n"; # The first all_lowercase word: is

$text =~ m/\b([A-Z]+\b)/g;

print "The subsequent all_uppercase word: $1\n"; # ... word: SILLY
```

В обоих случаях поиска используется скалярный контекст с модификатором /g.

Операции поиска в скалярном контексте с модификатором /g связаны друг с другом:

- первая операция (`m//`) устанавливает «текущую позицию» за совпавшим словом, записанным строчными буквами,
- а вторая (`/g`) продолжает поиск и находит первое слово, записанное прописными буквами, начиная с этой позиции.

Модификатор /g указывается при каждой операции, чтобы в процессе поиска учитывалась «текущая позиция», поэтому если хотя бы в одной из команд отсутствует модификатор /g, вторая команда найдет слово ‘WOW’ (напоминание:

\w - то же, что и [a-zA-Z0-9_]

\W - противоположность символа \w, то есть [^\w]

\b граница слова: позиция между \w и \W или \W и \w

Т.е. символ ! входит в \W, а символ W входит в \w, и \b устанавливается между ними).

- Эту конструкцию удобно использовать в условии цикла while.

Рассмотрим следующий фрагмент:

```
while ($ConfigData =~ m/^(\w+)=(.*)/mg) {
      my($key, $value) = ($1, $2);
      ...
}
```

- В цикле будут найдены все совпадения, но
- между совпадениями (вернее, после каждого совпадения) будет выполняться тело цикла.
- После того как очередная попытка поиска завершится неудачей, результат окажется ложным и цикл while завершится.
- После неудачи также сбрасывается состояние /g, поэтому следующий поиск с модификатором /g начнется с начала строки.

Сравните фрагменты:

```
$text = "64.156.215.240";
while ($text =~ m/(\d+)/) { # Опасно!
      print "found: $1\n";
}
```

и

```
while ($text =~ m/(\d+)/g) {
      print "found: $1\n";
}
```

Они отличаются только присутствием модификатора /g, но это очень существенное различие. Скажем, если переменная $text содержит IP адрес из предыдущего примера, второй фрагмент выведет именно то, что нужно:

```
found: 64
found: 156
found: 215
found: 240
```

С другой стороны, первый фрагмент будет снова и снова выводить строку «found: 64». Без модификатора /g команда просто находит «первое вхождение (\d+) в $text», а это всегда ‘64’, независимо от количества проверок. Использование модификатора /g в конструкции со скалярным контекстом изменяет ее смысл – в новом варианте команда находит «следующее вхождение (\d+) в $text» и поэтому последовательно выводит все числа.

2) Начальная позиция поиска и функция pos()

С каждой строкой в Perl ассоциируется «начальная позиция поиска», откуда начинается поиск. Начальная позиция является атрибутом строки и не ассоциируется с каким-либо конкретным регулярным выражением.

- После создания или модификации строки поиск начинается с самого начала, но после того как будет найдено успешное совпадение, начальная позиция перемещается в конец найденного совпадения.
- При следующем поиске с модификатором /g механизм приступает к анализу строки от текущей начальной позиции.

Для работы с начальной позицией поиска в Perl используется функция pos(). Рассмотрим следующий ***пример***:

```
my $ip = "64.156.215.240";
while ($ip =~ m/(\d+)/g) {
      printf "found '$1' ending at location %d\n", pos($ip);
}
```

Результат:

```
found: '64' ending at location 2
found: '156' ending at location 6
found: '215' ending at location 10
found: '240' ending at location 14
```

Вспомним: индексация в строках начинается с нуля, поэтому «позиция 2» находится перед третьим символом.

После успешного поиска с модификатором /g значение $+[0] (первый элемент массива @+, см. Лек12) совпадает со значением, возвращаемым pos для целевой строки.

Функция pos() использует тот же аргумент по умолчанию, что и оператор поиска – переменную $_.

3) Предварительная настройка начальной позиции поиска

- Настоящая сила функции pos() заключается в том, что ей можно присвоить значение.
- Тем самым вы указываете механизму регулярных выражений позицию, с которой должен начинаться поиск (разумеется, если в нем используется модификатор /g).

Пример. Допустим журналы веб-сервера хранятся в специализированном формате; каждая запись содержит 32-байтовый заголовок, за которым следует запрашиваемая страница и прочая информация. Чтобы извлечь из записи данные о странице, можно воспользоваться конструкцией ^.{32} и пропустить заголовок фиксированной длины:

```
if ($logline =~ m/^.{32}(\S+)/) {
      $RequestedPage = $1;
}
```

Механизму регулярных выражений приходится выполнять работу по пропуску первых 32 байтов. Такое решение и менее эффективно, и менее наглядно, чем ручной перевод начальной позиции:

```
pos($logline) = 32; # Страница начинается после 32 символа
# начать поиск с этой позиции...
if ($logline =~ m/(\S+)/g) {
      $RequestedPage = $1;
}
```

Такое решение лучше, но оно не эквивалентно прежнему. Поиск начинается с нужной позиции, но в отличие от оригинала второе решение не требует обязательного совпадения в этой позиции. Если по какой-то причине 33-й символ не совпадет с \S, в первом варианте поиск завершится неудачей, а во втором варианте, не привязанном к определенной позиции в строке, механизм продолжит поиск после смещения. Таким образом, он может вернуть совпадение \S+, начиная с более поздней позиции в строке. К счастью, у этой проблемы имеется простое решение, описанное в следующем разделе.

4) Якорный метасимвол \G

Якорный метасимвол \G обозначает «позицию, в которой завершилось предыдущее совпадение». Именно это нам и требуется для решения проблемы, описанной в предыдущем разделе:

```
pos($logline) = 32; # Страница следует после 32 символа,
# поэтому поиск следует начинать с этой позиции.
if (Slogline =~ m/\G(\S+)/g) {
      $RequestedPage = $1;
}
```

Наличие метасимвола \G фактически означает: «не смещать текущую позицию в этом регулярном выражении – если совпадение не находится от начальной позиции, немедленно сообщить о неудаче».

В Perl метасимвол \G надежно работает лишь в том случае, если он находится в самом начале регулярного выражения, а само выражение не содержит высокоуровневой конструкции выбора.

5) Поиск с модификаторами /gc

Обычно в результате неудачной попытки поиска m/…/g позиция pos возвращается в начало целевого текста. Но если к модификатору /g добавить модификатор /c, неудача не будет приводить к сбросу начальной позиции поиска.

- Модификатор /c никогда не используется без /g, поэтому будем использовать общую запись /gc.

Конструкция m/…/gc чаще всего используется в сочетании с \G для создания «лексеров», разбирающих строку на компоненты. Ниже приведен простой пример разбора кода HTML в переменной $html:

```
while (not $html =~ m/\G\z/gc) # Продолжать до конца текста...
# \z совпадает только в конце строки без учета возможных символов новой строки
{
      if ($html =~ m/\G( <[^>]+> )/xgc) { print "TAG: $1\n" }
      elsif ($html =~ m/\G( &\w+; )/xgc) {print "NAMED ENTITY: $1\n" }
      elsif ($html =~ m/\G( &\#\d+; )/xgc) {print "NUMERIC ENTITY: $1\n" }
      elsif ($html =~ m/\G( [^<>&\n]+ )/xgc) {print "TEXT: $1\n" }
      elsif ($html =~ m/\G \n /xgc) {print "NEWLINE\n" }
      elsif ($html =~ m/\G( . )/xgc) {print "ILLEGAL CHAR: $1\n" }
      else {
            die "$0: oops, this shouldn't happen!";
      }
}
```

В каждом регулярном выражении имеется часть, совпадающая с одним из типов конструкций HTML (выделена жирным шрифтом).

- Все проверки осуществляются последовательно, начиная с текущей позиции (из-за /gc), однако совпадение может начинаться только с этой позиции (из-за \G).
- Регулярные выражения перебираются до тех пор, пока цикл не опознает текущую лексему и не выведет информацию о ней.
- В результате позиция pos для переменной $html перемещается в начало следующей лексемы, которая обрабатывается на следующей итерации цикла.

Цикл завершается тогда, когда находится совпадение для m/\G\z/gc, т.е. когда текущая позиция \G перейдет в конец строки (\z).

Важная особенность приведенного решения заключается в том, что при каждой итерации один из вариантов должен совпадать. Если совпадение не будет найдено (и цикл не будет прерван), произойдет зацикливание, поскольку ничто не приведет к продвижению или сбросу позиции pos для строки $html.

В пример включена завершающая секция else; в представленной версии управление никогда не передается в эту секцию, но в процессе редактирования в программу могут быть внесены ошибки, поэтому присутствие секции else вполне оправдано. Если данные содержат незапланированные последовательности (например, ‘<>’), представленная версия выдает одно предупреждение для каждого непредвиденного символа.

У такого решения имеется еще один важный аспект – порядок проверок. Например, выражение \G(.) проверяется в последнюю очередь.

Предположим, мы хотим расширить приложение так, чтобы оно опознавало блоки <script>:

$html =~ m/\G (<script[^>]*>.*?</script>)/xgcsi

(целых пять модификаторов!) Чтобы программа работала правильно, новую проверку необходимо вставить перед проверкой <[^>]+>, которая сейчас находится на первом месте, или <[^>]+> совпадет с открывающим тегом <script> и перехватит совпадение.