

jenyay.net

Софт, исходники и фото

Поиск:

>>

[Домой](#) [Блог](#) [Контакты](#)[Печать](#) [Править](#)

Блог

Программки

OutWiker (rus)[Плагины](#)[Бета-версии](#)[Локализации](#)[Документация](#)[Предложения и](#)[баги](#)[Исходники](#)**OutWiker (en)**[Plug-ins](#)[Beta versions](#)[Translate](#)[Suggestions and](#)[bugs](#)[Source code](#)[Documentation](#)**Другие...****Программирование**[Python](#)[Rust](#)[.NET/C#](#)[C++](#)[PHP](#)[Алгоритмы](#)[Инструменты](#)[Остальное](#)**Обзоры книг**

Программирование скриптов для Vim. Часть 4. Работа со строками

Предыдущие части

[Часть 1. Запуск скриптов](#)[Часть 2. Переменные](#)[Часть 3. Работа со списками](#)

Оглавление

- [Создание строк](#)
- [Основные операции со строками](#)
- [Регулярные выражения](#)
- [Практика](#)
- [Комментарии](#)

[В прошлый раз](#) мы подробно разобрали работу со списками в Vim. На этот раз пришла очередь строк. В этой статье мы узнаем какие функции предоставляет Vim для работы со строками, рассмотрим операции, которые напоминают работу со списками, а также специфические для строк, после этого кратко пробежимся по регулярным выражениям в Vim, а завершит эту статью пример, близкий к практическому использованию.

Создание строк

Мы уже сталкивались с созданием строк, но на всякий случай напомним, что строки создаются с помощью кавычек. В Vim используются два типа кавычек: одинарные и двойные. Разница между ними состоит в том, что внутри

Студентам

Фото

Животные
Черно-белые
Пейзажи/Природа
Город
Закаты
Панорамы
Спорт
Репортаж
Разное

Контакты

двойных кавычек могут содержаться специальные символы вроде `\n` (перевод строки), `\t` (табуляция) и тому подобные, наверняка вам знакомые по другим языкам программирования. При создании строк с помощью одинарных кавычек добавить специальные символы не удастся, так как выражение `'\n'` будет интерпретироваться как строка, состоящая из двух символов: `\` и `n`. Чтобы добавить обратный слеш в строку, создаваемую с помощью двойных кавычек, его необходимо удвоить. Рассмотрим простой пример.

```
echo "Foo"
echo "Bar"

echo "Foo\tBar"
echo 'Foo\tBar'

echo "Foo\\Bar"
echo 'Foo\Bar'
```

[Исходник](#)

Результат выполнения этого скрипта (с помощью команды `:source %`) показан на следующем скриншоте:

```
Foo
Bar
Foo      Bar
Foo\tBar
Foo\Bar
Foo\Bar
Press ENTER or type command to continue
```

Выражение `\t` в двойных кавычках интерпретируется как символ табуляции, а в одинарных как два символа `\` и `t`. Чтобы вывести обратный слеш в двойных кавычках его пришлось удвоить, а в одинарных этого не понадобилось. Для знакомых с языком C# поведение двойных кавычек `"..."` так и остается без изменений, а вот аналогом одинарных кавычек `'...'` Vim в C# является запись вида `@"..."`.

Кроме того, одинарные и двойные кавычки удобно использовать совместно, когда требуется создать строку, содержащую внутри себя символ кавычек. Внутри двойных кавычек одинарные можно использовать напрямую, не заботясь о них, и наоборот двойные кавычки можно безбоязненно использовать внутри одинарных. Кроме того внутри двойных кавычек тоже можно использовать символ двойных кавычек, только перед ними нужно добавить символ обратного слеша `\`, потому что интерпретатор выражение `\`

преобразует в символ двойных кавычек. Эти случаи демонстрируются в следующем примере:

```
echo "'Foo Bar'"
echo '"Foo Bar"'
echo "\"Foo Bar\""
```

[Исходник](#)

Результат работы выглядит следующим образом:

```
'Foo Bar'
"Foo Bar"
"Foo Bar"
```

При желании можно сохранять строки в переменную точно так же, как это делается с другими типами с помощью оператора *let*:

```
let s:foo = "Foo"
let s:bar = "Bar"
```

[Исходник](#)

Основные операции со строками

Доступ к символам и проблемы с многобайтовыми кодировками

Строки во многом схожи со списками (списками символов, хотя это и не совсем так, о чем речь пойдет чуть позже), поэтому не удивительно, что многие операции у строк и списков выглядят одинаково. Например, с помощью квадратных скобок `[]` мы можем обращаться к одному из элементов строки. Я сознательно не говорю "символа", так как в Vim оператор индексирования возвращает один байт, а не символ, а если в строке содержится текст в многобайтовой кодировке (Unicode), то здесь возможны проблемы.

Допустим, у нас есть следующий скрипт:

```
let s:foo_en = "Abyrvalg"
let s:foo_ru = "Абырвалг"

echo s:foo_en
echo s:foo_ru

echo s:foo_en[3]
echo s:foo_ru[3]
```

[Исходник](#)

В зависимости от того в какой кодировке он был набран мы получим разный результат. Например, у меня в Windows XP кодировка для командной строки установлена как Windows-1251, поэтому если скрипт набран в этой же однобайтовой кодировке, то никаких проблем не возникает:

```
~
Abyrvalg
Абырвалг
r
p
Press ENTER or type command to continue
```

Если же скрипт набран в кодировке UTF-8, то будет выведена абракадабра (что не удивительно):

```
~
Abyrvalg
РћР±С<СЪРІР°Р»Рі
r
±
Press ENTER or type command to continue
```

Исправить вывод слова "Абырвалг" довольно легко, достаточно перед выводом преобразовать строку в кодировку Windows-1251 (это делается с помощью функции `iconv()`, о которой мы поговорим чуть позже), а вот как быть с доступом к отдельным символам, а не байтам, вопрос сложный. Можно, конечно, перед каждой операцией с элементами строки ее преобразовывать в однобайтовую кодировку, но не понятно как быть, если символ должен быть принципиально многобайтовым.

Ситуация усугубляется еще и тем, что если мы захотим узнать длину строки с помощью встроенных функций `len()` или `strlen()` (эти функции работают одинаково), то у нас возникнет другая проблема, связанная с тем, что перечисленные функции возвращают длину строки все в тех же байтах, а не в символах. Рассмотрим следующий пример. Его нужно набирать в кодировке UTF-8:

```
let foo = "абырвалг"
echo len (foo)
echo strlen (foo)
```

[Исходник](#)

В этом примере мы создаем строковую переменную `foo` со значением "абырвалг" и хотим узнать ее длину. Для этого

будем использовать функции `len()` и `strlen()`. Результат получается несколько неожиданным, если не знать, что наша строка многобайтовая:

```
16
16
```

Как быть? В документации предлагают перед расчетом длины строки заменить все символы каким-нибудь заведомо однобайтовым с помощью регулярного выражения:

```
echo len (substitute (foo, ".", "x", "g"))
```

[Исходник](#)

Функция `substitute()`, которая здесь применяется, заменяет в строке `foo` все найденные соответствия регулярному выражению, переданного вторым параметром (у нас это `"."`), на строку, переданную третьим параметром (у нас это `"x"`), а последний параметр `"g"` означает, что необходимо пройтись по всей строке `foo`. Но у меня и этот способ не заработал. Функция `substitute()` заменила на символ `x` все однобайтовые элементы строки, и в результате получилась строка той же длины 16 символов.

Можно перед расчетом длины строки преобразовать ее в однобайтовую кодировку:

```
let foo = "абырвалг"
echo len (iconv (foo, "utf-8", "windows-1251"))
```

[Исходник](#)

В этом случае мы получим правильный результат. В этом примере мы воспользовались функцией `iconv()`, которая как раз и преобразует строку из одной кодировки в другую. Пользоваться ей легко. Первый параметр - исходная строка, второй - исходная кодировка строки, третий - кодировка в которую хотим ее преобразовать. Функция возвращает преобразованную строку. Кодировку `"windows-1251"` можно обозвать так же `"cp1251"`, UTF-8 как `"utf8"`, функция `iconv()` поймет, какую именно кодировку вы имели в виду.

Но все-таки такой подход не очень удобен, если мы заранее не знаем нашей кодировки. Правда, можно воспользоваться одной хитростью. Если Vim не сможет

преобразовать какой-то символ, то он его заменит на знак "?", который является однобайтовым. Поэтому если предыдущий пример сохранить в кодировке Windows-1251, то функция `iconv()` вернет строку "????????", а функция `len()` определит ее длину правильно. Но это, конечно, шаманство.

Теперь, когда мы обсудили возможные проблемы с многобайтовыми кодировками, для дальнейших примеров, где это явно не оговорено, давайте считать, что мы обрабатываем однобайтный текст и сами скрипты написаны в однобайтовой кодировке.

Выделение подстрок

Возвращаемся к нашим баранам, а точнее к оператору `[]`. С помощью него можно не только получать один символ (строго говоря, байт) из строки, но и подстроку (опять же, строго говоря, последовательность байтов, а не символов). Для этого точно так же, как и у списков, используется запись вида

```
foo[start:end]
```

Здесь *foo* - исходная строка, *start* - номер первого байта извлекаемой подстроки, а *end* - номер последнего байта подстроки. Извлекаемая подстрока включает в себя и символ с индексом *start*, и символ с индексом *end*. И так же, как и у списков, параметр *start* можно пропускать, если он равен 0. Так же можно не писать значение *end*, если извлекаемая подстрока должна продолжаться до конца исходной строки. Все индексы начинаются с 0. К сожалению, для строк не работает запись вида `foo[-1]` для извлечения последнего символа. При извлечении подстрок также нужно быть аккуратнее с многобайтными кодировками.

Следующий пример демонстрирует основные операции, выполняемые с помощью квадратных скобок. Очень удобно для демонстрации интервальных операций использовать строку, состоящую из последовательности чисел - не надо высчитывать номер каждого символа.

```
let foo = "0123456789"
echo foo

echo foo[0]
echo foo[2]
```

```
echo foo[0:5]  
echo foo[:5]  
  
echo foo[5:]
```

[Исходник](#)

Еще раз напомним, что, как и при работе со списками, между именем переменной и открывающейся скобкой `[` не должно быть пробелов.

Результат работы этого скрипта:



```
0123456789  
0  
2  
012345  
012345  
56789  
Press ENTER or type command to continue
```

Сначала выводится вся строка, затем два символа, а затем три подстроки.

Для выделения подстрок также существует функция `strpart()`. Ее синтаксис выглядит следующим образом:

```
strpart(src, start[, len])
```

Здесь *src* - исходная строка, из которой нужно извлечь подстроку. *start* - номер байта, начиная с которого начинается извлекаемая подстрока, байт с номером *start* тоже включается в подстроку. *len* - необязательный параметр, который определяет сколько байт нужно "отсчитать" вправо от позиции *start*, чтобы определить где заканчивается подстрока, если этот параметр не задан, то подстрока заканчивается в конце основной строки *str*.

Если говорить более просто, то *start* определяет положение начала подстроки. При этом это значение может быть отрицательным, но это отрицательное значение не означает отсчет от конца строки, как у списка, а означает действительно отрицательную позицию. Как это использовать дальше станет ясно из примера. Чтобы узнать, где будет кончаться подстрока, прибавляем к значению *start* значение *len* - 1 и получаем положение конца подстроки. Теперь мы имеем начало и конец

подстроки, при этом они могут выходить за пределы исходной строки, например, `start` может быть отрицательным, а рассчитанный конец подстроки находится далеко за последним символом строки. В этом нет ничего страшного. Функция `strpart()` вернет подстроку, состоящую только из тех символов, которые находятся в пределах исходной строки.

Рассмотрим некоторые примеры. В комментариях будет написано, что выведет на экран Vim после той или иной команды.

```
" Выведет 34.  
echo strpart("0123456789", 3, 2)  
  
" Выведет 3456789  
echo strpart("0123456789", 3)  
  
" Выведет 012  
echo strpart("0123456789", -2, 5)  
  
" Выведет 56789  
echo strpart("0123456789", 5, 15)  
  
" Выведет 0123456789  
echo strpart("0123456789", -5, 20)  
  
" Выведет пустую строку  
echo strpart("0123456789", -5, 2)
```

[Исходник](#)

Еще раз обратите внимание, что `len` - в общем виде это не длина полученной подстроки, это значение только определяет положение конца подстроки. И помните, что параметры `start` и `len` задаются в байтах, а не в символах.

Другие операции

Одна из самых часто используемых строковых операций - это склеивание строк, или, говоря умными словами, конкатенация. Для этой операции в Vim предусмотрен специальный оператор `."`. Для добавления в конец строки другой строковой переменной можно использовать оператор `.="`. Не думаю, что здесь нужны особые комментарии, поэтому просто рассмотрим пример.

```
let s:foo = "Foo"  
let s:bar = "Bar"  
  
let s:spam = s:foo . "-" . s:bar  
  
" Будет выведено Foo-Bar  
echo s:spam  
  
let s:spam .= " spam spam spam"
```



```
" Будет выведено Foo-Bar spam spam spam  
echo s:spam
```

[Исходник](#)

Теперь давайте обсудим вопросы, связанные с преобразованием переменных численных типов в строку и наоборот. Сначала о преобразованиях в строки. Для этого существует функция *string()*, которая принимает в качестве своего единственного параметра какой-нибудь из известных Vim типов (это может быть строка, целое число, дробное число, список, словарь или указатель на функцию), а возвращает строку, соответствующую содержимому переданного параметра. И, как всегда пример, нас в данный момент интересуют только численные типы:

```
let s:foo = 123  
let s:bar = 3.1416  
  
let s:foo_str = string(s:foo)  
let s:bar_str = string(s:bar)  
  
echo s:foo_str "type:" type(s:foo_str)  
echo s:bar_str "type:" type(s:bar_str)
```

[Исходник](#)

В этом примере создаются две переменные (*s:foo* и *s:bar*), а затем они преобразуются в строки (соответственно *s:foo_str* и *s:bar_str*). После этого *s:foo_str* и *s:bar_str* выводятся на экран, а чтобы убедиться, что они действительно имеют строковый тип, также выводится и значение, возвращаемое функцией *type()* для этих переменных. Напомню, что функция *type()* возвращает значение 1, если ее параметр имеет строковый тип. Результат работы этого скрипта вы видите на следующем скриншоте.



```
123 type: 1  
3.1416 type: 1  
Press ENTER or type command to continue
```

Для обратного преобразования из строки в число существуют две функции: *str2nr()* для преобразования строки в целое число и *str2float()* для преобразования строки в число с плавающей точкой. *str2float()* имеет единственный параметр - строка, которую нужно преобразовать в число, а вот *str2nr()* имеет кроме того второй, необязательный параметр, описывающий систему счисления, в которой представлена строка. По умолчанию

используется десятичная система счисления, но могут быть использованы также и восьмеричная, и шестнадцатеричная системы. Для простоты мы будем использовать только десятичную систему счисления.

Следующий пример показывает использование функций *str2nr()* и *str2float()*.

```
let s:foo_str = "123"
let s:bar_str = "3.1416"

let s:foo = str2nr (s:foo_str)
let s:bar = str2nr (s:bar_str)

echo s:foo "type: " type (s:foo)
echo s:bar "type: " type (s:bar)
```

[Исходник](#)

Напомню, что функция *type()* для целых чисел возвращает значение 0, а для дробных значение 5. В этом вы можете убедиться на следующем скриншоте, который показывает результат работы скрипта.



```
123 type: 0
3.1416 type: 5
Press ENTER or type command to continue
```

Рассмотрим теперь тот же самый пример, но в случае, если строка на самом деле не представляет собой числа:

```
let s:foo_str = "абыр"
let s:bar_str = "абырвалг"

let s:foo = str2nr (s:foo_str)
let s:bar = str2nr (s:bar_str)

echo s:foo "type: " type (s:foo)
echo s:bar "type: " type (s:bar)
```

[Исходник](#)

Результат работы:



```
0 type: 0
0.0 type: 5
Press ENTER or type command to continue
```

Как видите, если строку не удалось преобразовать в число, то функция *str2nr()* возвращает значение 0, а *str2float()* значение 0.0. Так как эти функции при ошибках не бросают исключения (при исключениях мы

поговорим как-нибудь в другой раз), то в случае нулевого результата узнавать удачно ли произошло преобразование или нет может быть немного неудобно. Как вариант, в случае нулевого результата можно вручную проверять действительно ли в строке кроме нулей и знака минуса (а для функции `str2float()` и символов ".", "e" и "E") ничего нет. Но это тоже не самый лучший вариант проверки из-за одной особенности интерпретатора Vim, которая показана в следующем примере:

```
let s:foo_str = "10абыр"
let s:bar_str = "3.1416абырвалг"

let s:foo = str2nr(s:foo_str)
let s:bar = str2float(s:bar_str)

echo s:foo "type: " type(s:foo)
echo s:bar "type: " type(s:bar)
```

[Исходник](#)

Здесь переменные `s:foo_str` и `s:bar_str` начинаются с числа, а затем идет строковый мусор. А теперь посмотрите на результат:

```
10 type: 0
3.1416 type: 5
Press ENTER or type command to continue
```

Vim не растерялся и просто обрезал весь мусор в конце. Насколько такое поведение оправдано судить не берусь, но про такое поведение интерпретатора тоже не следует забывать.

Несмотря на то, что мы до сих пор не рассматривали операторы вроде `if/else` (про них мы поговорим в ближайших частях), хотелось бы рассмотреть операции сравнения строк.

Во-первых, все операторы сравнения (в Vim это `"=="` - равно, `"!="` - не равно, а также `">"`, `"<"`, `">="` и `"<="`) возвращают целочисленные значения: 1, если условие истинно, и 0, если ложно. Пока остановимся только на операторах `"=="` и `"!="`. Для строк возникает резонный вопрос: а как происходит сравнение строк, с учетом регистра или нет? Ответ: или так, или так :) Дело в том, что в Vim есть настройка `ignorecase`, которая как раз и определяет как будут сравниваться строки с помощью операторов `"=="` и `"!="`. То есть, при сравнении строк

регистр не будет учитываться, если в файле `.vimrc` или `_vimrc` есть строка

```
set ignorecase
```

[Исходник](#)

И наоборот, эта настройка выключается с помощью команды.

```
set noignorecase
```

[Исходник](#)

По умолчанию настройка `ignorecase` выключена. Давайте посмотрим как влияет эта команда на сравнение строк.

```
let s:foo = "abyrvalg"
let s:bar = "ABYRVALG"

set noignorecase
echo s:foo == s:bar

set ignorecase
echo s:foo == s:bar

unlet s:foo s:bar
```

[Исходник](#)

В этом примере мы создаем две строки (`s:foo` и `s:bar`), различающиеся только регистром букв. После этого сначала отключаем настройку `ignorecase` (с помощью строки `set noignorecase`) и сравниваем строки. Затем включаем настройку `ignorecase` (с помощью строки `set ignorecase`) и снова сравниваем те же строки. Результат довольно ожидаем:

```
0
1
```

То же самое можем проделать, заменив оператор `"=="` на `"!="`:

```
let s:foo = "abyrvalg"
let s:bar = "ABYRVALG"

set noignorecase
echo s:foo != s:bar

set ignorecase
echo s:foo != s:bar

unlet s:foo s:bar
```

[Исходник](#)

Теперь результат будет противоположный:

```
1
0
```

Казалось бы, удобно по ходу работы скрипта переключать опцию *ignorecase* то туда, то сюда. Но на самом деле это не лучшая идея. Во-первых, потому что это влияет на глобальные настройки редактора, а скрипт в большинстве случаев после себя должен вернуть настройки к тому виду, как они были до его вмешательства. А, во-вторых, разработчики Vim предусмотрели еще два оператора сравнения строк, один из них всегда учитывает регистр (оператор "`==#`"), а другой регистр всегда игнорирует (оператор "`==?`"). На эти операторы настройка *ignorecase* не действует. Давайте в этом убедимся:

```
let s:foo = "abyrvalg"
let s:bar = "ABYRVALG"

set noignorecase
echo s:foo ==# s:bar

set ignorecase
echo s:foo ==# s:bar

set noignorecase
echo s:foo ==? s:bar

set ignorecase
echo s:foo ==? s:bar

unlet s:foo s:bar
```

[Исходник](#)

В этом примере каждый из операторов "`==#`" и "`==?`" выполняется при разных настройках *ignorecase*, но на результат это не влияет, и на экран будут выведены следующие строки:

```
0
0
1
1
```

Регулярные выражения

Поиск по регулярному выражению

Вот мы и подошли к самому интересному. Дело в том, что в Vim в большинстве строковых операций можно использовать регулярные выражения, например, в функции поиска подстроки или замены. Больше того, в Vim нет аналогичных

функций, но без регулярных выражений, поэтому их приходится применять даже для простых операций. Думаю, что вы знаете что такое регулярные выражения и с чем их едят, поэтому говорить об этом я не буду. Также, не будем говорить про их составление, по этой теме пишут целые книги. Давайте лучше рассмотрим использование регулярных выражений и их особенности в Vim. Разбираться с ними будем на примере самой простой функции, их использующей, - функции `match()`. Она осуществляет поиск подстроки, соответствующей шаблону регулярному выражению. `match()` возвращает номер байта, где найдено совпадение. Она имеет следующий синтаксис:

```
match(expr, pattern[, start[, count]])
```

- *expr* - строка, в которой ищется подстрока. На самом деле здесь может быть список, но об этом случае мы поговорим отдельно.
- *pattern* - строка, описывающая регулярное выражение.
- *start* - номер байта в строке, с которого нужно начинать поиск подстроки (по умолчанию используется 0)
- *count* - номер совпадения с шаблоном, который нас интересует. То есть, если `count == 1` (значение по умолчанию), то функция `match()` вернет положение первой же найденной подстроки, а если `count == 2`, то функция вернет положение (в байтах) второй найденной подстроки и так далее.

Если функция `match()` не найдет совпадений, то она вернет значение -1.

При поиске по регулярному выражению для определения того нужно ли учитывать регистр также используется настройка `ignorecase` / `noignorecase`. Рассмотрим пример:

```
let s:foo = "qqqWWWeeRRRtttYYYQQQ"
set ignorecase
echo match (s:foo, "QQQ")

set noignorecase
echo match (s:foo, "QQQ")

unlet s:foo
```

[Исходник](#)

Здесь мы ищем подстроку "QQQ". В результате работы этого примера сначала будет выведено число 0 (так как "qqq" ==? "QQQ"), затем 18.

Параметры регулярных выражений

Как уже говорилось выше, трогать пользовательские настройки в большинстве случаев нехорошо, поэтому регулярные выражения могут содержать некоторые установки, которые влияют только на конкретное выражение. С помощью одной из таких установок можно определить будет ли учитываться регистр. Установка называется `\c`, если регистр не надо учитывать) и `\C`, если надо. Параметры для регулярного выражения можно писать куда угодно, хоть в начало выражения, хоть в конец, хоть в середину. Но нельзя забывать, что при использовании двойных кавычек Vim после слеша ожидает какой-нибудь специальный символ вроде перевода строки (`"\n"`) или табуляции (`"\t"`), поэтому, чтобы в строке содержался слеш, его необходимо удвоить. Это видно на следующем примере:

```
let s:foo = "qqqWWWeeeRRRtttYYYQQQ"

set ignorecase
echo match (s:foo, "\\cQQQ")
echo match (s:foo, "\\CQQQ")

unlet s:foo
```

[Исходник](#)

Здесь мы устанавливаем настройку `ignorecase`, а затем ищем подстроку, сначала с параметрами `\c`, а затем `\C`. В результате на экран будет выведено:

```
0
18
```

Поиск по регулярному выражению можно чуть компактнее переписать с использованием одинарных кавычек:

```
echo match (s:foo, '\cQQQ')
echo match (s:foo, '\CQQQ')
```

[Исходник](#)

А можно переписать и вот так:

```
echo match (s:foo, 'QQQ\c')
```

```
echo match (s:foo, 'Q\CQQ')
```

[Исходник](#)

Здесь параметр `\с` перенесен в конец выражения, а `\C` в середину. Последнее явно не пойдет на пользу читаемости, а вот где писать параметры: в начале или в конце - это дело вкуса.

Теперь поиграемся с необязательными параметрами функции `match()`:

```
let s:foo = "qqqWWWeeeRRRtttYYYQQQ"
echo match (s:foo, '\cQQQ', 0, 2)
echo match (s:foo, '\cQQQ', 5, 1)
unlet s:foo
```

[Исходник](#)

Первый вызов `match()` ищет второе вхождение регулярного выражения, но поиск начинается с нулевой позиции. Второй вызов начинает поиск с пятой позиции, но возвращает первое вхождение. В результате обеих этих строк будет найдено выражение "QQQ", которое начинается на 18-й позиции.

Теперь мы попытаемся отыскать третье вхождение, начиная с нулевой позиции или второе, начиная с пятой позиции:

```
let s:foo = "qqqWWWeeeRRRtttYYYQQQ"
echo match (s:foo, '\cQQQ', 0, 3)
echo match (s:foo, '\cQQQ', 5, 2)
unlet s:foo
```

[Исходник](#)

В результате ничего похожего найдено не будет и функция `match()` в обоих случаях вернет значение -1.

Нотации регулярных выражений

Не смотря на то, с формальной точки зрения мы уже использовали регулярное выражение "QQQ", но вы же понимаете, что этим возможности Vim не ограничиваются. Поэтому пришло время разобраться с тем как записываются полноценные регулярные выражения. Начнем с самых простых и всем известных команд:

- `"."` - любой символ

- "?" - ноль или одно совпадение предыдущего выражения
- "*" - любое количество предыдущих выражений.

Допустим, мы хотим найти в предыдущей строке не просто "QQQ", а последовательность из символа, длины не меньше 2. Во многих языках это выражение выглядело бы как "QQ*", а в Vim возможны варианты. Дело в том, что в этом редакторе есть некий магический параметр с именем *magic*, который определяет как Vim будет отличать управляющие символы регулярного выражения от обычных печатных символов (литералов), которые надо воспринимать как буквы, цифры и знаки препинания.

Допустим, параметр *magic* включен с помощью следующей команды

```
set magic
```

[Исходник](#)

Если ничего не менять в настройках `.vimrc/_vimrc`, то по умолчанию этот параметр так и считается включенным. В этом случае в спорных ситуациях в большинстве случаев приоритет отдается символу регулярного выражения, а не литере. А, если в выражении подразумевается литера, то перед ней нужно ставить обратный слеш. Такое поведение можно отключить с помощью команды

```
set nomagic
```

[Исходник](#)

Тогда приоритет будет отдаваться литере, а для того, чтобы показать, что символ является управляющим в регулярном выражении, перед ним нужно ставить обратный слеш. Но такое поведение работает не для всех управляющих символов, поэтому надо быть аккуратнее и почаще заглядывать в справку, там часто описывается как тот или иной управляющий символ записывается при разных режимах.

Пусть у нас есть следующее выражение:

```
123.
```

Если установлен параметр *magic*, то оно будет интерпретироваться как "последовательность литер 1, 2 и 3, за которым идет любой символ". Если установлен параметр *nomagic*, то это же выражение обозначает "последовательность литер 1, 2 и 3, за которым идет литера точка". Чтобы изменить интерпретацию точки в обоих случаях достаточно перед ней поставить обратный слеш:

```
123\.
```

[Исходник](#)

Чтобы не гадать какой режим установлен у пользователя Vim, регулярные выражения поддерживают также следующие параметры:

- `\m` - включает режим *magic*.
- `\M` - включает режим *nomagic*.

Вернемся к нашему предыдущему примеру.

```
let s:foo = "qqqWWWeeeRRRtttYYYQQQ"
echo match (s:foo, '\c\mQQ*')
echo match (s:foo, '\c\MQQ*')
unlet s:foo
```

[Исходник](#)

Первый поиск по регулярному выражению со включенным параметром *magic* найдет совпадение на нулевой позиции, а второй поиск с выключенной магией не найдет ничего, так как там значок "*" будет интерпретироваться буквально, как литера "звездочка".

Кроме параметров `\m` и `\M`, существует еще два режима регулярных выражений, которые нельзя включить через настройки `.vimrc/_vimrc`:

- `\v` - Очень магический режим, в котором все символы за исключением цифр, английских букв и символа подчеркивания интерпретируются как управляющие.
- `\V` - Совсем немагический режим, в котором символы всегда интерпретируются как литералы, а перед каждым управляющим символом должен стоять обратный слеш.

Подробнее про эти режимы с некоторыми дополнительными примерами вы можете прочитать в справке, выполнив следующую команду:

```
:help magic
```

В дальнейшем мы будем использовать иногда и эти режимы.

Давайте на этом остановимся при рассмотрении записи регулярных выражений и пройдемся по остальным возможностям Vim, которые с ними связаны. Чтобы узнать побольше про возможности Vim для построения регулярных выражений, введите команду

```
:help pattern
```

Поиск по списку

При первом упоминании функции `match()` было сказано, что в качестве первого ее параметра может выступать список строк. В этом случае будет возвращен номер первого элемента списка, в котором найдена подстрока, удовлетворяющая регулярному выражению. Причем заметьте, что с регулярным выражением не обязательно должна совпадать вся строка целиком. Рассмотрим пример. Пусть у нас есть список дат в формате "ДД.ММ.ГГГГ", и мы хотим узнать индекс элемента, содержащий дату, относящуюся к 19 веку:

```
let s:dates = ['12.09.2008', '25.01.1975', '03.03.1812',  
              '06.08.2100', '16.10.1983', '30.10.1856']  
  
echo match(s:dates, '\m\d\d\.\d\d\.\d\d\d\d')  
  
unlet s:dates
```

[Исходник](#)

В этом примере мы использовали выражение `\d`, обозначающее любое число от 0 до 9. Этот пример выведет на экран число 2.

Здесь мы полностью описывали формат даты, но можем и упростить выражение, если будем искать только год, перед которым стоит точка:

```
let s:dates = ['12.09.2008', '25.01.1975', '03.03.1812',  
              '06.08.2100', '16.10.1983', '30.10.1856']
```

```
echo match (s:dates, '\m\d\d\.\d\d\.\d\d\.\d\d')
echo match (s:dates, '\m\.\d\d\.\d\d\.\d\d')

unlet s:dates
```

[Исходник](#)

Обратите внимание, что мы используем магический ражим, поэтому для того, чтобы точка воспринималась как литерал, а не "любой символ", перед ней мы ставим обратный слеш.

Другие функции для работы с регулярными выражениями

Функция `matchstr()`

Эта функция тоже осуществляет поиск с помощью регулярного выражения, но, в отличие от `match()`, возвращает не номер первого найденного совпадения, а подстроку, которая соответствует выражению. Все параметры этой функции в точности совпадают с параметрами функции `match()`. Например, следующий скрипт ищет дату в строке:

```
let s:foo = 'Это произошло 15.06.2058, когда...'
echo matchstr (s:foo, '\m\d\{2\}\.\d\{2\}\.\d\{4\}')
unlet s:foo
```

[Исходник](#)

Здесь мы использовали выражение `\{n\}`, обозначающее, что предыдущий символ (в общем случае выражение) должно повториться ровно *n* раз. В результате выполнения этого скрипта мы увидим строку

```
15.06.2058
```

То же самое мы можем переписать в очень магическом режиме (с параметром `\v`), тогда обратные слешы перед фигурными скобками нужно будет убрать, и будет заметно нагляднее:

```
let s:foo = 'Это произошло 15.06.2058, когда...'
echo matchstr (s:foo, '\v\d{2}\.\d{2}\.\d{4}')
unlet s:foo
```

[Исходник](#)

Результат работы скрипта тот же самый.

Функция `matchlist()`

Еще одна полезная функция называется `matchlist()`, она возвращает список, состоящий из найденной подстроки (нулевой элемент) и внутренних подвыражений, которые разделяются с помощью символов группировки `\(... \)` (в магическом режиме). Если в предыдущем примере мы заменим функцию `matchstr()` на `matchlist()` и немного изменим регулярное выражение, то сможем выделить отдельно всю дату, день, месяц и год.

```
let s:foo = 'Это произошло 15.06.2058, когда...'
echo matchlist(s:foo, '\m\(\d\{2\}\)\.\(\d\{2\}\)\.\(\d\{4\}\)\')
unlet s:foo
```

[Исходник](#)

Результат работы скрипта может быть несколько неожиданным:

```
['15.06.2058', '15', '06', '2058', , , , , , ]
```

Может возникнуть вопрос откуда взялись пустые строки. Здесь на самом деле все просто. Дело в том, что Vim поддерживает только 9 переменных, которые хранят значения объединенных с помощью `\(... \)` подвыражений. Эти переменные имеют имена `\1`, `\2` и так далее до `\9`. Их можно использовать в регулярных выражениях, когда требуется, чтобы какая-то его часть повторялась, но не подряд, или использовать при замене. А функция `matchlist()` создает список, в котором содержатся значения всех таких переменных, поэтому неиспользованные переменные хранят пустые строки.

Вас может смутить, то что можно выделить всего 9 переменных, а ведь скобки используются и для группировки выражений, когда не требуется их значения сохранять. Например, если нужно, чтобы регулярное выражение повторилось несколько раз, то после такой группы может стоять символ `*`. Если нам не надо сохранять значение подвыражений, то лучше использовать конструкцию группировки `\%(... \)` вместо `\(... \)`. Следующий пример ищет последовательности из двух цифр, причем

группы из двух цифр должны быть разделены пробелом или табуляцией.

```
let s:foo = '6987 12 12 52 8754 236 295744 25877'
echo matchlist (s:foo, '\m[[:blank:]]^\%(\\d\\d[[:blank:]]$\\)\{1,\\}')
unlet s:foo
```

[Исходник](#)

Буквально выражение можно прочитать следующим образом: "Выражение начинается с символа пробела, табуляции (`[:blank:]`) как раз и обозначает пробел или табуляцию) или начала строки (^). Затем идет группа из двух чисел и пробела, символа табуляции или конца строки (\$). Группа должна повториться 1 или более раз (`\{1,\\}`)".

В нашем примере этому выражению удовлетворяет строка "12 12 52 ". Если мы запустим этот скрипт, то получим список, нулевым элементом которого будет эта самая строка, а остальные 9 элементов будут пустыми строками. То есть мы смогли использовать группировку, но при этом не потратили ни одной переменной \1 - \9.

Функция `substitute()`

До сих пор мы только искали текст с помощью регулярного выражения, но, разумеется, в Vim есть средства для замены. Собственно для этого и существует функция `substitute()`. Синтаксис ее выглядит следующим образом:

```
substitute(expr, pattern, sub, flags)
```

- *expr* - Строка, в которой нужно произвести замену.
- *pattern* - шаблон регулярного выражения.
- *sub* - Выражение, на которое будет заменена подстрока, подходящая под регулярное выражение.
- *flags* - флаги настройки. Может быть либо "g", если нужно произвести замену во все строке, либо пустая строка "", если нужно провести только первую замену.

Эта функция возвращает строку, в которой были произведены замены, если совпадения с регулярным выражением были найдены, и исходную строку без замен, если совпадений не было найдено.

Начнем с простого примера:

```
let s:foo = "qqqWWWeeeRRRtttYYYQQQ"
echo substitute(s:foo, '\cQQQ', "ZZZ", "g")
unlet s:foo
```

[Исходник](#)

Результат довольно ожидаемый - обычная замена выражений без учета регистра (благодаря настройке \c):

```
ZZZWWWeeeRRRtttYYYZZZ
```

Теперь продемонстрируем для чего могут понадобиться переменные \1, ..., \9. Пусть у нас есть список дат в формате ДД.ММ.ГГГГ, а нам нужно оставить от каждой даты только год. Это делает следующий скрипт:

```
let s:foo = '12.09.2008 25.01.1975 03.03.1812 06.08.2100'
echo substitute(s:foo, '\v\d{2}\.\d{2}\.(\d{4})', '\1', "g")
unlet s:foo
```

[Исходник](#)

В очень магическом режиме (\v) мы используем скобки (...), чтобы значение, найденное внутри них было сохранено в переменной \1. В качестве строки для замены мы указываем в кавычках как раз эту переменную, чтобы функция *substitute()* заменила найденное регулярное выражение целиком на его отдельную часть, взятую в скобки, то есть в нашем случае на год. В результате такой операции мы получим следующий результат:

```
2008 1975 1812 2100
```

Регулярные выражения - это очень большая тема, поэтому мы рассмотрели далеко не всё и не все функции, которые предоставляет Vim для работы с ними. Но рассмотренные функции, на мой взгляд, будут использоваться наиболее часто, а про остальные вы сможете узнать из справки. Советую набрать в командной строке Vim следующую команду, там много всего интересного:

```
:help function-list
```

Практика

И в завершении статьи рассмотрим более практичный пример. А будет он делать следующее. Дело в том, что эту статью я пишу с использованием нотации движка [pmWiki](#), в ней каждый крупный подраздел начинается с оператора "!!", обозначающий заголовок, за которым идет название раздела (аналог тега H2). Давайте напишем скрипт, который будет автоматически создавать оглавление для статьи, которое вы видели в самом начале. Для ссылки оглавления перед каждым крупным заголовком я добавляю якорь (anchor), который задается следующим образом:

```
[[#anchor]]
```

[Исходник](#)

Якорь располагается на строку выше заголовка. Например, заголовок этого раздела выглядит следующим образом:

```
[[#example]]
!! Практика
```

[Исходник](#)

Давайте облегчим мне жизнь и сделаем скрипт, который по готовой статье будет создавать оглавление, которое должно содержать по одной строке для каждого раздела в следующем виде (на примере этого раздела):

```
* [[#example | Практика]]
```

[Исходник](#)

Скрипт, который все это делает выглядит следующим образом (надеюсь, что комментариев в его коде будет достаточно для понимания):

```
" Регулярное выражение для нахождения якоря
let s:pattern_anchor = '\m^\[\#\([a-zA-Z0-9_]*\)\]\]'

" Регулярное выражение для нахождения заголовка
let s:pattern_title = '\m^!!\s*\(.*)$'

" Количество строк в файле
let s:count = line("$")

" Сюда будут добавляться строки оглавления
let s:result = []

" Проходимся по всем строкам в буфере.
for n in range(1, s:count)
    " Получим номер строки по ее номеру
    " Строки в буфере нумеруются с 1
    let s:currline = getline(n)
```



```

" Для каждой строки проверим соответствует ли она шаблону
с якорем
let s:anchor = matchlist (s:currline, s:pattern_anchor)

if len (s:anchor) != 0
    " Если строка соответствует, то список не будет пустым

    " Теперь проверим соответствует ли следующая строка
шаблону заголовка
let s:nextline = getline (n + 1)
let s:title = matchlist (s:nextline, s:pattern_title)

    if len (s:title) != 0
        " Если и заголовок найден, создадим строку
оглавления
let s:resline = printf ("* [[#%s | %s]]",
s:anchor[1], s:title[1])
call add (s:result, s:resline)
    endif
endif

endfor

" Получить положение курсора в виде списка:
" [номер буфера,
" номер строки,
" номер столбца,
" параметр при использовании опции virtualedit]
let s:cursor = getpos(".")

" Вставим полученное оглавление в ту строку, где сейчас стоит
курсор
call append (s:cursor[1], s:result)

" Удалим все переменные
unlet s:pattern_anchor s:pattern_title s:count
unlet! s:resline s:nextline s:title s:anchor s:currline

```

[Исходник](#)

В результате мы получим вот такое оглавление:

```

* [[#create | Создание строк]]
* [[#operators | Основные операции со строками]]
* [[#regexp | Регулярные выражения]]
* [[#example | Практика]]
* [[#example | Практика]]

```

[Исходник](#)

Последние строки удвоились, так как заголовок действительно написан дважды - один раз в начале раздела, а второй как пример заголовка. Но это не страшно :)

Теперь я могу сохранить этот скрипт, допустим, под именем *titles.vim*, поставить курсор туда, где должно быть оглавление и выполнить команду

```
:source titles.vim
```

Все, оглавление будет создано автоматически.

Итак, эта статья оказалась довольно большой, но, надеюсь, полезной. В ближайших частях мы будем

рассматривать работу со словарями и создание функций, после чего можно будет приступить к программированию полноценных плагинов для Vim.

Продолжение следует.

[Часть 5. Операции ветвления и функции](#)

Вы можете подписаться на новости сайта через [RSS](#), [Группу Вконтакте](#) или [Канал в Telegram](#).



Рейтинг 4.7/5. Всего 33 голос(а, ов)

☐ Плохо ☐ Так себе ☐ Неплохо ☐ Хорошо ☐ Отлично

sKwa 31.07.2009 - 05:11

Ошибка

В главе "*Выделение подстрок*" последняя картинка не соответствует примеру. Вообще очепяток много. Как вариант, если правописание важно, можно поставить систему Орфус или найди редактора, чтоб diff возможно было делать, а о каждой запоминать местоположение и писать о ней - сизифов труд.

sKwa 31.07.2009 - 05:14

Сорри

Сорри - это не последняя картинка, а единственная в этой главе 🙄

sKwa 31.07.2009 - 05:26

Регулярные выражения

Ошибка в оформлении - там бокс захватил лишнее

Jenyay 31.07.2009 - 09:32

Спасибо, сейчас исправлю, что найду. Ведь проверял.

Jenyay 31.07.2009 - 10:10

Орфус тоже поставил, еще раз спасибо за наводку.

sKwa 31.07.2009 - 13:21

Спасибо тебе


Спасибо тебе за статьи, а очепятки - это так мелочи 😊


poljak181 21.11.2010 - 02:46

спасибо

Женуа 21.11.2010 - 09:20
Да, действительно. Исправил, спасибо.







228

Послать