Evgeny Vladimirovich

medium.com/nuances-of-programming/шпаргалка-по-регулярным-выражениям-в-примерах-53820a5f3435

6 января 2019 г.

Шпаргалка по регулярным выражениям. В примерах

Регулярные выражения (regex или regexp) очень эффективны для извлечения информации из текста. Для этого нужно произвести поиск одного или нескольких совпадений по определённому шаблону (т. е. определённой последовательности символов ASCII или unicode).

Области применения regex разнообразны, от валидации до парсинга/замены строк, передачи данных в другие форматы и Web Scraping'a.

Одна из любопытных особенностей регулярных выражени в их универсальности, стоит вам выучить синтаксис, и вы сможете применять их в любом (почти) языке программирования (JavaScript, Java, VB, C #, C / C++, Python, Perl, Ruby, Delphi, R, Tcl, и многих других). Небольшие отличия касаются только наиболее продвинутых функций и версий синтаксиса, поддерживаемых движком.

Давайте начнём с нескольких примеров.

Основы

Якоря — ^ и $

```
       соответствует строке, начинающейся с ->соответствует строке,
заканчивающейся на точное совпадение (начинается и заканчивается как
)соответствует любой строке, в которой есть текст
```

Квантификаторы — * + ? и {}

```
      соответствует строке, в которой после следует 0 или более символов->
соответствует строке, в которой после следует один или более символов
соответствует строке, в которой после следует 0 или один символ      соответствует
строке, в которой после следует 2 символа    соответствует строке, в которой после
следует 2 или более символов   соответствует строке, в которой после следует от 2
до 5 символовсоответствует строке, в которой после следует 0 или более
последовательностейсимволов соответствует строке, в которой после следует от 2 до
5 последовательностейсимволов
```

Оператор ИЛИ — | или []

```
соответствует строке, в которой после    как и в предыдущем примере
```

Символьные классы — \d \w \s и .

```
 соответствует одному символу, который является цифрой-> соответствует слову
(может состоять из букв, цифр и подчёркивания) ->  соответствует символу пробела
(включая табуляцию и прерывание строки)соответствует любому символу->
```

Используйте оператор `.` с осторожностью, так как зачастую класс или отрицаемый класс символов (который мы рассмотрим далее) быстрее и точнее.

У операторов `\d`, `\w` и `\s` также есть отрицания — `\D`, `\W` и `\S` соответственно.

Например, оператор `\D` будет искать соответствия противоположенные `\d`.

```
 соответствует одному символу, который не является цифрой->
```

Некоторые символы, например `^.[$()|*+?{\` , необходимо выделять обратным слешем `\` .

```
соответствует строке, в которой после символа ->
```

Непечатаемые символы также можно искать, например табуляцию `\t`, новую строку `\n`, возврат каретки `\r`.

Флаги

Мы научились строить регулярные выражения, но забыли о фундаментальной концепции — **флагах**.

Регулярное выражение, как правило, записывается в такой форме /`abc`/, где шаблон для сопоставления выделен двумя слешами `/`. В конце выражения, мы определяем значение флага (эти значения можно комбинировать):

- g (global) — не возвращает результат после первого совпадения, а продолжает поиск с конца предыдущего совпадения.
- m (multi line) — с таким флагом, операторы `^` и `$` вызовут совпадение в начале и конце строки ввода (line), вместо строки целиком (string).
- i (insensitive) — делает выражение регистронезависимым (например, соответствует).

Средний уровень

Скобочные группы — ()

```
a(bc)       создаём группу со значением bc -> теста(?:bc)*    оперетор ?:
отключает группу -> теста(?<foo>bc) так, мы можем присвоить имя группе -> тест
```

Этот оператор очень полезен, когда нужно извлечь информацию из строк или данных, используя ваш любимый язык программирования. Любые множественные совпадения, по нескольким группам, будут представлены в виде классического

массива: доступ к их значениям можно получить с помощью индекса из результатов сопоставления.

Если присвоить группам имена (используя `(?<foo>...)`), то можно получить их значения, используя результат сопоставления, как словарь, где ключами будут имена каждой группы.

Скобочные выражения — []

[abc] соответствует строке, которая содержит либо символ **a** или **a b** или **a c** -> такой же эффект от **a|b|c** -> **тест[a-c]** то же, что и выше**[a-fA-F0-9]** строка, представляющая одну шестнадцатеричную цифру без учёта регистра -> **тест[0-9]%** строка, содержащая символ от 0 до 9 перед знаком %**[^a-zA-Z]** строка, которая не имеет буквы от a до z или от A до Z. В этом случае ^ используется как отрицание в выражении -> **тест**

Помните, что внутри скобочных выражений все специальные символы (включая обратную косую черту `\`) теряют своё служебное значение, поэтому нам ненужно их экранировать.

Жадные и ленивые сопоставления

Квантификаторы (`* + {}`) — это «жадные» операторы, потому что они продолжают поиск соответствий, как можно глубже — через весь текст.

Например, выражение `<.+>` соответствует `<div>simple div</div>` в `This is a <div> simple div</div> test`. Чтобы найти только тэг `div` — можно использовать оператор `?`, сделав выражение «ленивым»:

<.+?> соответствует любому символу, один или несколько раз найденному между **<** и **>**, расширяется по мере необходимости -> **тест**

Обратите внимание, что хорошей практикой считается не использовать оператор `.`, в пользу более строгого выражения:

<[^<>]+> соответствует любому символу, кроме **< или >**, один или более раз встречающемуся между **<** и **>** -> **тест**

Продвинутый уровень

Границы слов — \b и \B

\babc**\b** выполняет поиск слова целиком -> **тест**

`\b` — соответствует границе слова, наподобие якоря (он похож на `$` и `^`), где предыдущий символ — словесный (например, `\w`), а следующий — нет, либо наоборот, (например, это может быть начало строки или пробел).

`\B` — соответствует несловообразующей границе. Соответствие не должно обнаруживаться на границе `\b`.

\Babc**\B** соответствует, только если шаблон полностью окружён словами -> **тест**

Обратные ссылки — \1

([abc]**)\1 \1** соответствует тексту из первой захватываемой группы -> **тест**([abc])**(**[de]**)\2**\1 можно использовать **\2** (\3, \4, и т.д.) для определения порядкового номера захватываемой группы -> **тест(?<foo>**[abc]**)\k<foo>** мы присвоили имя **foo** группе, и теперь ссылаемся на неё используя — (**\k<foo>**). Результат, как и в первом выражении -> **тест**

Опережающие и ретроспективные проверки — (?=) and (?<=)

d**(?=**r**)** соответствует **d,** только если после этого следует **r**, но **r** не будет входить в соответствие выражения -> **тест(?<=**r**)**d соответствует **d,** только если перед этим есть **r**, но **r** не будет входить в соответствие выражения -> **тест**

Вы можете использовать оператор отрицания !

d**(?!**r**)** соответствует **d,** только если после этого нет **r**, но **r** не будет входить в соответствие выражения -> **тест(?<!**r**)**d соответствует **d,** только если перед этим нет **r**, но **r** не будет входить в соответствие выражения -> **тест**

Заключение

Как вы могли убедиться, области применения регулярных выражений разнообразны. Я уверен, что вы сталкивались с похожими задачами в своей работе (хотя бы с одной из них), например такими:

- Валидация данных (например, правильно ли заполнена строка time)
- Сбор данных (особенно веб-скрапинг, поиск страниц, содержащих определённый набор слов в определённом порядке)
- Обработка данных (преобразование сырых данных в нужный формат)
- Парсинг (например, достать все GET параметры из URL или текст внутри скобок)
- Замена строк (даже во время написания кода в IDE, можно, например преобразовать Java или C# класс в соответствующий JSON объект, заменить “:” на “,”, изменить размер букв, избегать объявление типа и т.д.)
- Подсветка синтаксиса, переименование файла, анализ пакетов и многие другие задачи, где нужно работать со строками (где данные не должны быть текстовыми).

Перевод статьи Jonny Fox: Regex tutorial — A quick cheatsheet by examples