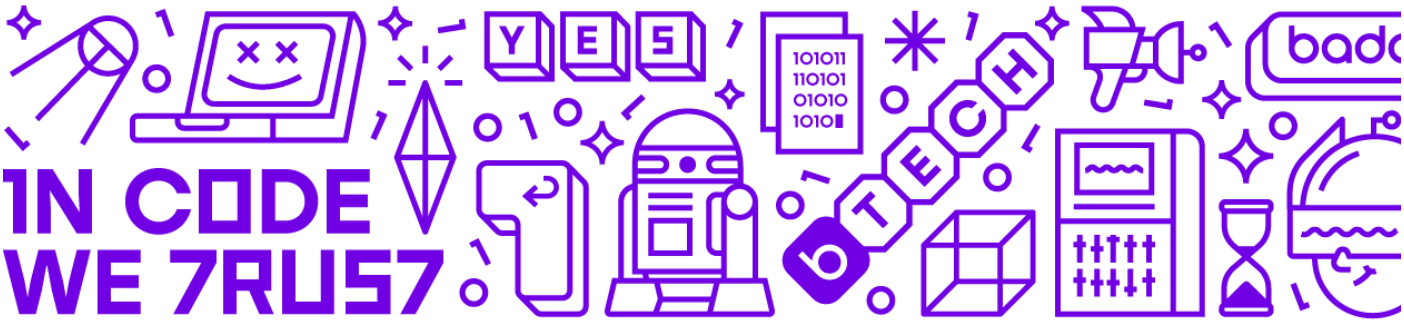


Badoo 282,99
Big Dating

nizkopal 27 ноября в 16:46

Регулярные выражения для самых маленьких

Регулярные выражения, Программирование, PHP, JavaScript, Блог компании Badoo

Привет, Хабр.

Под катом я расскажу основы работы с регулярными выражениями. На эту тему написано много теоретических статей. В этой ста решил сделать упор на количество примеров. Мне кажется, что это лучший способ показать возможности этого инструмента.

Некоторые из них для наглядности будут показаны на примере языков программирования PHP или JavaScript, но в целом они рабо независимо от ЯП.

Из названия понятно, что статья ориентирована на самый начальный уровень — тех, кто еще ни разу не использовал регулярные выражения в своих программах или делал это без должного понимания.

В конце статьи я в двух словах расскажу, какие задачи нельзя решить регулярными выражениями и какие инструменты для этого использовать.

Поехали!

Вступление

Регулярные выражения — язык поиска подстроки или подстрок в тексте. Для поиска используется паттерн (шаблон, маска), состоящий из символов и метасимволов (символы, которые обозначают не сами себя, а набор символов).

Это довольно мощный инструмент, который может пригодиться во многих случая — поиск, проверка на корректность строки и т.д Спектр его возможностей трудно уместить в одну статью.

В PHP работа с регулярными выражениями заключается в наборе функций, из которых я чаще всего использую следующие:

- preg_match (http://php.net/manual/en/function.preg-match.php)
- preg_match_all (http://php.net/manual/en/function.preg-match-all.php)
- preg_replace (http://php.net/manual/en/function.preg-replace.php)

Для работы с ними нужен текст, в котором мы будем искать или заменять подстроки, а также само регулярное выражение, описывающее правило поиска.

Функции на match возвращают число найденных подстрок или false в случае ошибок. Функция на replace возвращает измененную строку/массив или null в случае ошибки. Результат можно привести к bool (false, если не было найдено значений и true, если было использовать вместе с if или assertTrue для обработки результата работы.

В JS чаще всего мне приходится использовать:

- match (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match)
- test (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/test)
- replace (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace)

Все дальнейшие примеры предлагаю смотреть в https://regex101.com/. Это удобный и наглядный интерфейс для работы с регулярн выражениями.

Пример использования функций

В PHP регулярное выражение — это строка, которая начинается и заканчивается символом-разделителем. Все, что находится ме разделителями и есть регулярное выражение.

Часто используемыми разделителями являются косые черты “/”, знаки решетки “#” и тильды “~”. Ниже представлены примеры шаблонов с корректными разделителями:

- /foo bar/
- #^[^0-9]$#
- %[a-zA-Z0-9_-]%

Если необходимо использовать разделитель внутри шаблона, его нужно проэкранировать с помощью обратной косой черты. Если разделитель часто используется в шаблоне, в целях удобочитаемости, лучше выбрать другой разделитель для этого шаблона.

- /http:\/\//
- #http://#

В JavaScript регулярные выражения реализованы отдельным объектом RegExp и интегрированы в методы строк.

Создать регулярное выражение можно так:

```
let regexp = new RegExp("шаблон", "флаги");
```

Или более короткий вариант:

```
let regexp = /шаблон/; // без флагов
```

```
let regexp = /шаблон/gmi; // с флагами gmi (изучим их дальше)
```

Пример самого простого регулярного выражения для поиска:

```
RegExp: /o/
Text: hello world
```

В этом примере мы просто ищем все символы “o”.

В PHP разница между preg_match и preg_match_all в том, что первая функция найдет только первый match и закончит поиск, в то вр
как вторая функция вернет все вхождения.

Пример кода на PHP:

```php
<?php
$text = ‘hello world’;
$regexp = ‘/o/’;
$result = preg_match($regexp, $text, $match);
var_dump(
    $result,
    $match
);
```

```
int(1)  // нам вернулось одно вхождение, т.к. после функция заканчивает работу
array(1) {
  [0]=>
  string(1) "o" // нам вернулось вхождение, аналогичное запросу, так как метасимволов мы пока не использовали
}
```

Пробуем то же самое для второй функции:

```php
<?php
$text = ‘hello world’;
$regexp = ‘/o/’;
$result = preg_match_all($regexp, $text, $match);
var_dump(
    $result,
    $match
);
```

```
int(2)
array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(1) "o"
    [1]=>
    string(1) "o"
  }
}
```

В последнем случае функция вернула все вхождения, которые есть в нашем тексте.

Тот же пример на JavaScript:

```javascript
let str = 'Hello world';
let result = str.match(/o/);
console.log(result);
```

```
["o", index: 4, input: "Hello world"]
```

Модификаторы шаблонов

Для регулярных выражений существует набор модификаторов, которые меняют работу поиска. Они обозначаются одиночной букв латинского алфавита и ставятся в конце регулярного выражения, после закрывающего “/”.

- i — символы в шаблоне соответствуют символам как верхнего, так и нижнего регистра.
- m — по умолчанию текст обрабатывается, как однострочная символьная строка. Метасимвол начала строки '^' соответствует т началу обрабатываемого текста, в то время как метасимвол конца строки '$' соответствует концу текста. Если этот модификат используется, метасимволы «начало строки» и «конец строки» также соответствуют позициям перед произвольным символо перевода и строки и, соответственно, после, как и в самом начале, и в самом конце строки.

Об остальных модификаторах, используемых в PHP, можно почитать тут.

В JavaScript — тут.

О том, какие вообще бывают модификаторы, можно почитать тут.

Пример предыдущего регулярного выражения с модификатором на JavaScript:

```
let str = "hello world \
             How is it going?"
let result = str.match(/o/g);
console.log(result);
```

```
["o", "o", "o", "o"]
```

Метасимволы в регулярных выражениях

Примеры по началу будут довольно примитивные, потому что мы знакомимся с самыми основами. Чем больше мы узнаем, тем бли реалиям будут примеры.

Чаще всего мы заранее не знаем, какой текст нам придется парсить. Заранее известен только примерный набор правил. Будь то п в смс, email в письме и т.п.

Первый пример, нам надо получить все числа из текста:

```
Текст: “Привет, твой номер 1528. Запомни его.”
```

Чтобы выбрать любое число, надо собрать все числа, указав “[0123456789]”. Более коротко можно задать вот так: “[0-9]”. Для все цифр существует метасимвол “\d”. Он работает идентично.

Но если мы укажем регулярное выражение “/\d/”, то нам вернётся только первая цифра. Мы, конечно, можем использовать модификатор “g”, но в таком случае каждая цифра вернется отдельным элементом массива, поскольку будет считаться новым вхождением.

Для того, чтобы вывести подстроку единым вхождением, существуют символы плюс “+” и звездочка “*”. Первый указывает, что на подойдет подстрока, где есть как минимум один подходящий под набор символ. Второй — что данный набор символов может быт может и не быть, и это нормально. Помимо этого мы можем указать точное значение подходящих символов вот так: “{N}”, где N — нужное количество. Или задать “от” и “до”, указав вот так: “{N, M}”.

Сейчас будет пара примеров, чтобы это уложилось в голове:

```
Текст: “Я хочу ходить на работу 2 раза в неделю.”
Надо получить цифру из тексте.
RegExp: “/\d/”
```

```
Текст: “Ваш пинкод: 24356” или “У вас нет пинкода.”
Надо получить пинкод или ничего, если его нет.
RegExp: “/\d*/”
```

```
Текст: “Номер телефона 89091534357”
Надо получить первые 11 символов, или FALSE, если их меньше.
RegExp: “/\d{11}/”
```

Примерно так же мы работает с буквами, не забывая, что у них бывает регистр. Вот так можно задавать буквы:

- [a-z]
- [a-zA-Z]
- [а-яА-Я]

С кириллицей указанный диапазон работает по-разному для разных кодировок. В юникоде, например, в этот диапазон не входит б “ё”. Подробнее об этом тут.

Пара примеров:

```
Текст: “Вот бежит олень” или “Вот ваш индюк”
Надо выбрать либо слово “олень”, либо слово “индюк”.
RegExp: “/[а-яА-Я]+/”
```

Такое выражение выберет все слова, которые есть в предложении и написаны кириллицей. Нам нужно третье слово.

Помимо букв и цифр у нас могут быть еще важные символы, такие как:

- \s — пробел
- ^ — начало строки
- $ — конец строки
- | — “или”

Предыдущий пример стал проще:

```
Текст: “Вот бежит олень” или “Вот бежит индюк”
Надо выбрать либо “олень”, либо “индюк”.
RegExp: “/[а-яА-Я]+$/”
```

Если мы точно знаем, что искомое слово последнее, мы ставим “$” и результатом работы будет только тот набор символов, после которого идет конец строки.

То же самое с началом строки:

```
Текст: “Олень вкусный” или “Индюк вкусный”
Надо выбрать либо “олень”, либо “индюк”.
RegExp:  “/^[а-яА-Я]+/”
```

Прежде, чем знакомиться с метасимволами дальше, надо отдельно обсудить символ “^”, потому что он у нас ходит на две работы (это чтобы было интереснее). В некоторых случаях он обозначает начало строки, но в некоторых — отрицание.

Это нужно для тех случаев, когда проще указать символы, которые нас не устраивают, чем те, которые устраивают.

Допустим, мы собрали набор символов, которые нам подходят: “[a-z0-9]” (нас устроит любая маленькая латинская буква или циф теперь предположим, что нас устроит любой символ, кроме этого. Это будет обозначаться вот так: “[^a-z0-9]”.

Пример:

```
Текст: “Я люблю кушать суп”
Надо выбрать все слова.
RegExp: “[^\s]+”
```

Выбираем все “не пробелы”.

Итак, вот список основных метасимволов:

- \d — соответствует любой цифре; эквивалент [0-9]
- \D — соответствует любому не числовому символу; эквивалент [^0-9]
- \s — соответствует любому символу whitespace; эквивалент [\t\n\r\f\v]
- \S — соответствует любому не-whitespace символу; эквивалент [^ \t\n\r\f\v]
- \w — соответствует любой букве или цифре; эквивалент [a-zA-Z0-9_]
- \W — наоборот; эквивалент [^a-zA-Z0-9_]
- . — (просто точка) любой символ, кроме перевода “каретки”

Операторы [] и ()

По описанному выше можно было догадаться, что [] используется для группировки нескольких символов вместе. Так мы говорим, нас устроит любой символ из набора.

Пример:

```
Текст: “Не могу перевести I dont know, помогите!”
Надо получить весь английский текст.
RegExp: “/[A-Za-z\s]{2,}/”
```

Тут мы собрали в группу (между символами []) все латинские буквы и пробел. При помощи {} указали, что нас интересуют вхожде где *минимум* 2 символа, чтобы исключить вхождения из пустых пробелов.

Аналогично мы могли бы получить все русские слова, сделав инверсию: “[^A-Za-z\s]{2,}”.

В отличие от [], символы () собирают отмеченные выражения. Их иногда называют “захватом”.

Они нужны для того, чтобы передать выбранный кусок (который, возможно, состоит из нескольких вхождений [] в результат выда

Пример:

```
Текст: ‘Email you sent was ololo@example.com Is it correct?’
Нам надо выбрать email.
```

Существует много решений. Пример ниже — это *приближенный* вариант, который просто покажет возможности регулярных выражений. На самом деле есть RFC, который определяет правильность email. И есть “регулярки” по RFC — вот примеры.

Мы выбираем все, что не пробел (потому что первая часть email может содержать любой набор символов), далее должен идти сим @, далее что угодно, кроме точки и пробела, далее точка, далее любой символ латиницы в нижнем регистре...

Итак, поехали:

- мы выбираем все, что не пробел: “[^\s]+”
- мы выбираем знак @: “@”
- мы выбираем что угодно, кроме точки и пробела: “[^\s\.]+”

- мы выбираем точку: “\.” (обратный слеш нужен для экранирования метасимвола, так как знак точки описывает любой символ - выше)
- мы выбираем любой символ латиницы в нижнем регистре: “[a-z]+”

Оказалось не так сложно. Теперь у нас есть email, собранный по частям. Рассмотрим на примере результата работы preg_match в

```
<?php
$text = ‘Email you sent was ololo@example.com. Is it correct?’;
$regexp = ‘/[^\s]+@[^\s\.]+\.[a-z]+/’;
$result = preg_match_all($regexp, $text, $match);
var_dump(
    $result,
    $match
);
```

```
int(1)
array(1) {
  [0]=>
  array(1) {
    [0]=>
    string(13) "ololo@example.com"
  }
}
```

Получилось! Но что, если теперь нам надо по отдельности получить домен и имя по email? И как-то использовать дальше в коде? Е нам поможет “захват”. Мы просто выбираем, что нам нужно, и оборачиваем знаками (), как в примере:

Было:

```
/[^\s]+@[^\s\.]+\.[a-z]+/
```

Стало:

```
/([^\s]+)@([^\s\.]+\.[a-z]+)/
```

Пробуем:

```
<?php
$text = ‘Email you sent was ololo@example.com. Is it correct?’;
$regexp = ‘/([^\s]+)@([^\s\.]+\.[a-z]+)/’;
$result = preg_match_all($regexp, $text, $match);
var_dump(
    $result,
    $match
);
```

```
int(1)
array(3) {
  [0]=>
  array(1) {
    [0]=>
    string(13) "ololo@example.com"
  }
  [1]=>
  array(1) {
    [0]=>
    string(5) "ololo"
  }
  [2]=>
  array(1) {
```

```
    [0]=>
    string(7) "example.com"
  }
}
```

В массиве match нулевым элементом всегда идет полное вхождение регулярного выражения. А дальше по очереди идут “захваты”

В PHP можно именовать “захваты”, используя следующий синтаксис:

```
/(?<mail>[^\s]+)@(?<domain>[^\s\.]+\.[a-z]+)/
```

Тогда массив матча станет ассоциативным:

```
<?php
$text = ‘Email you sent was ololo@example.com. Is it correct?’;
$regexp = ‘/(?<mail>[^\s]+)@(?<domain>[^\s\.]+\.[a-z]+)/’;
$result = preg_match_all($regexp, $text, $match);
var_dump(
    $result,
    $match
);
```

```
int(1)
array(5) {
  [0]=>
  array(1) {
    [0]=>
    string(13) "ololo@example.com"
  }
  ["mail"]=>
  array(1) {
    [0]=>
    string(5) "ololo"
  }
  ["domain"]=>
  array(1) {
    [0]=>
    string(7) "example.com"
  }
}
```

Это сразу +100 к читаемости и кода, и регулярки.

Примеры из реальной жизни

Парсим письмо в поисках нового пароля:

Есть письмо с HTML-кодом, надо выдернуть из него новый пароль. Текст может быть либо на английском, либо на русском:

```
Текст: “пароль: <b>f23f43tgt4</b>” или “password: <b>wh4k38f4</b>”
RegExp: “(password|пароль):\s<b>([^<]+)<\/b>”
```

Сначала мы говорим, что текст перед паролем может быть двух вариантов, использовав “или”.
Вариантов можно перечислять сколько угодно:

```
(password|пароль)
```

Далее у нас знак двоеточия и один пробел:

```
:\s
```

Далее знак тега b:

```
<b>
```

А дальше нас интересует все, что не символ “<”, поскольку он будет свидетельствовать о том, что тег b закрывается:

```
([^<]+)
```

Мы оборачиваем его в захват, потому что именно он нам и нужен.
Далее мы пишем закрывающий тег b, проэкранировав символ “/”, так как это спецсимвол:

```
<\/b>
```

Все довольно просто.

Парсим URL:

В PHP есть клевая функция, которая помогает работать с урлом, разбирая его на составные части:

```
<?php
$URL = "https://hello.world.ru/uri/starts/here?get_params=here#anchor";
$parsed = parse_url($URL);
var_dump($parsed);
```

```
array(5) {
  ["scheme"]=>
  string(5) "https"
  ["host"]=>
  string(14) "hello.world.ru"
  ["path"]=>
  string(16) "/uri/starts/here"
  ["query"]=>
  string(15) "get_params=here"
  ["fragment"]=>
  string(6) "anchor"
}
```

Давай сделаем то же самое, только регуляркой? :)

Любой урл начинается со схемы. Для нас это протокол http/https. Можно было бы сделать логическое “или”:

```
[http|https]
```

Но можно схитрить и сделать вот так:

```
http[s]?
```

В данном случае символ “?” означает, что “s” может есть, может нет…

Далее у нас идет “://”, но символ “/” нам придется экранировать (см. выше):

```
“:\/\/”
```

Далее у нас до знака “/” или до конца строки идет домен. Он может состоять из цифр, букв, знака подчеркивания, тире и точки:

```
[\w\.-]+
```

Тут мы собрали в единую группу метасимвол “\w”, точку ”\.” и тире ”-”.

Далее идет URI. Тут все просто, мы берем все до вопросительного знака или конца строки:

```
[^?$]+
```

Теперь знак вопроса, который может быть, а может не быть:

```
[?]?
```

Далее все до конца строки или начала якоря (символ #) — не забываем о том, что этой части тоже может не быть:

```
[^#$]+
```

Далее может быть #, а может не быть:

```
[#]?
```

Дальше все до конца строки, если есть:

```
[^$]+
```

Вся красота в итоге выглядит так (к сожалению, я не придумал, как вставить эту часть так, чтобы Habr не считал часть строки — комментарием):

```
/(?<scheme>http[s]?):\/\/(?<domain>[\w\.-]+)(?<path>[^?$]+)?(?<query>[^#$]+)?[#]?(?<fragment>[^$]+)?/
```

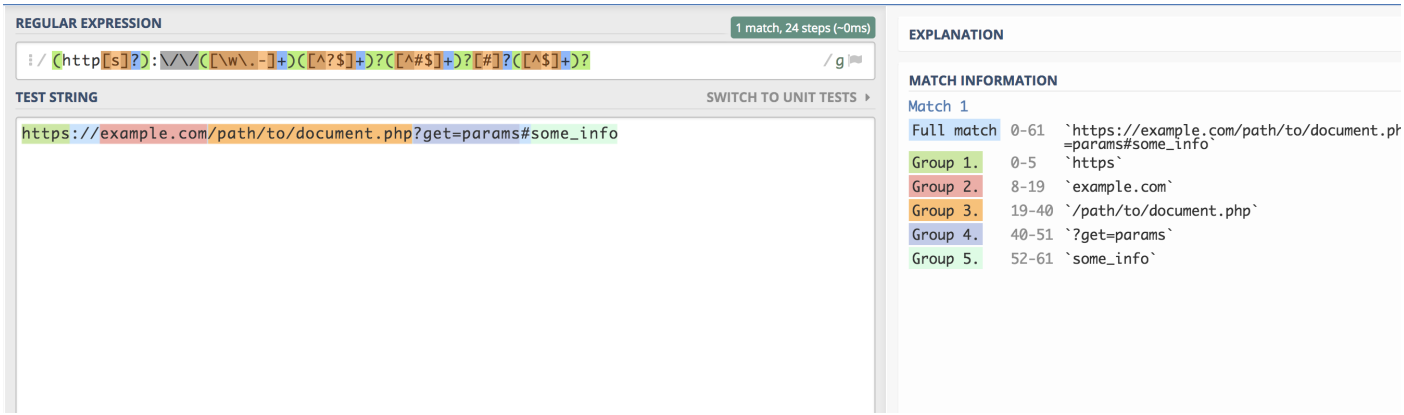
Главное не моргать! :)

```
<?php
$URL = "https://hello.world.ru/uri/starts/here?get_params=here#anchor";
$regexp = “/(?<scheme>http[s]?):\/\/(?<domain>[\w\.-]+)(?<path>[^?$]+)?(?<query>[^#$]+)?[#]?(?<fragment>[^$]+)?/”;
$result = preg_match($regexp, $URL, $match);
var_dump(
    $result,
    $match
);
```

```
array(11) {
  [0]=>
  string(61) "https://hello.world.ru/uri/starts/here?get_params=here#anchor"
  ["scheme"]=>
  string(5) "https"
  ["domain"]=>
  string(14) "hello.world.ru"
  ["URI"]=>
  string(16) "/uri/starts/here"
  ["params"]=>
  string(15) "get_params=here"
```

```
  ["anchor"]=>
  string(6) "anchor"
}
```

Получилось примерно то же самое, только своими руками.



Какие задачи не решаются регулярными выражениями

На первый взгляд кажется, что регулярными выражениями можно описать и распарсить любой текст. Но, к сожалению, это не так.

Регулярные выражении — это подвид формальных языков, который в иерархии Хомского принадлежат 3-ому типу, самому простс
Об этом тут.

При помощи этого языка мы не можем, например, парсить синтаксис языков программирования с вложенной грамматикой. Или Н
код.

Примеры задач:

У нас есть span, внутри которых много других span и мы не знаем сколько. Надо выбрать все, что находится внутри этого span:

```
<span>
    <span>ololo1</span>
    <span>ololo2</span>
    <span>ololo3</span>
    <span>ololo4</span>
    <span>ololo5</span>
    <...>
</span>
```

Само собой, если мы парсим HTML, где есть не только этот span. :)

Суть в том, что мы не можем начать с какого-то момента “считать” символы span и /span, подразумевая, что открывающих и закрывающих символов должно быть равное количество. И “понять”, что закрывающий символ, для которого ранее не было пары - самый закрывающий, который обосабливает блок.

То же самое с кодом и символами {}.

Например:

```
function methodA() {
    function() {<...>}
    if () { if () {<...>} }
}
```

В такой структуре мы не сможем при помощи только регулярного выражения отличить закрывающую фигурную скобку внутри код той, которая завершает начальную функцию (если код состоит не только из этой функции).

Для решение таких задач используются языки более высокого уровня.

Заключение

Я постарался довольно подробно рассказать об азах мира регулярных выражений. Конечно невозможно в одну статью уместить в Дальнейшая работа с ними — вопрос опыта и умения гуглить.

Спасибо за внимание.

Метки: regexp, javascript, junior developer, php

Комментарии 43

saver 27.11.17 в 17:08

Очень подробная статья и много примеров, да еще и все в одном месте, спасибо. Для новичка действительно может быть полезным стартс

Sirion 27.11.17 в 17:35

Не надо регулярно выражаться при самых маленьких.

redfs 27.11.17 в 18:06

> Мы выбираем все, что не пробел (потому что первая часть email может содержать любой набор символов)

... включая пробел.
`"name with spaces"@example.com`
Вообще, imho не надо учить новичков проверять email regexp-ами.

nizkopal 27.11.17 в 18:14

Да, Вы совершенно правы, спасибо. В таком случае имя пользователя должен находиться в кавычках, насколько я помню. С этим пример наша регулярка не справится.

Но, как я и написал в статье, это лишь тестовый пример, просто чтобы разбирать более или менее приближенные к реалиям кейзы. Мне показалось, что без них представление о том, какие задачи в целом решаются регулярными выражениями будет не полной.

В дальнейшем, конечно, такие вещи, как работа с email или URL стоит гуглить, интересоваться стандартами и типовыми решениями.

redfs 27.11.17 в 18:37

Понятно, что это лишь пример. Просто новички часто берут примеры сразу в работу, не читая особо текст с предупреждениями о RFC т.п... Случай email вообще — особенный. Казалось бы он достаточно нагляден, но

> В дальнейшем, конечно, такие вещи, как работа с email или URL стоит гуглить, интересоваться стандартами и типовыми решениями

А правильные типовые решения скорее всего такие:
Невозможно проверить адрес e-mail на допустимость с помощью регулярных выражений
или
Прекратите проверять Email с помощью регулярных выражений!

nizkopal 27.11.17 в 20:48

Раз уж завязалась дискуссия, стоит определиться с целью, которую мы преследуем, проверяя email.

Если мы делаем это, чтобы вычислить 100% возможных вариантов (например, с целью заспамить онные), то нам действительно сто позаботиться о том, чтобы не пропустить даже те, необычные, которые с пробелом.

Если же мы работаем на массовую аудиторию и наша задача — подсказать пользователю, что он, *возможно*, ошибся с введением своего email, то эту задачу мы вполне себе решим без чрезмерного уровня дотошности.

Тот же Gmail не даст так просто зарегистрировать email с пробелом, вот его сообщение об ошибке:

> Некорректное имя почтового ящика. Допустимо использовать только латинские буквы, цифры, знак подчеркивания («_»), точку («.»), минус («-»)

А вот ответ Yandex:

> Логин может состоять из латинских символов, цифр, одинарного дефиса или точки. Он должен начинаться с буквы, заканчиватьс буквой или цифрой и содержать не более 30 символов.

Ответ Mail:

> Некорректное имя почтового ящика. Допустимо использовать только латинские буквы, цифры, знак подчеркивания («_»), точку («.»), минус («-»)

Ответ Yahoo:

> Имя пользователя может содержать только буквы, цифры, точки (.) и символы подчеркивания (_).

Даже если к нам придет пользователь с некорректным email, который он себе как-то сделал, мы напишем ему, что следует завести другой email ради нашего сервиса. Но это менее 0.(0)1%.

Зато мы заранее предупредим остальных 99.(9)% пользователей о возможной опечатке, чтобы им не пришлось проходить этап регистрации дважды.

Я это к тому, что в погоне за «абсолютной правильностью» важно понимать, какую задачу мы решаем. И не подменять ее на другу абстрактную.

redfs 27.11.17 в 22:42

> Я это к тому, что в погоне за «абсолютной правильностью» важно понимать, какую задачу мы решаем. И не подменять ее на другую, абстрактную.

Вы в статье рассматриваете абстрактные примеры валидации email, и именно в этом абстрактном контексте я и беседую. Поэто надо в рамках данной дискуссии подменять абстрактную задачу на какую-то конкретную.

Поясню. Примеры с Яндексом и т.д. совсем не в тему. Это примеры систем со **своими** требованиями и ограничениями к формат email адреса и именно эти свои требования они проверяют. Это пример как раз конкретной задачи с определенными ограничени

Поэтому если бы в вашей статье пример выглядел так:
«Давайте проверим email пользователя Яндекс — логин может состоять из латинских символов, цифр, одинарного дефиса или то Он должен начинаться с буквы, заканчиваться буквой или цифрой и содержать не более 30 символов.»
то мне был бы понятен ваш аргумент, однако в таком случае и этой дискуссии бы не было.

Вы же в статье пишите о RFC

> На самом деле есть RFC, который определяет правильность email. И есть “регулярки” по RFC — вот примеры.

и якобы (примеры по ссылке — тоже работают неправильно) существующей возможности проверки email адреса регулярным выражением. Без какой-то конкретики о постановке задачи. Абстрактно. Я вам указал на то, что это — ошибка и что тему «валидация email регулярным выражением» было бы правильно пометить в главу «Какие задачи не решаются регулярными выражениями».

nizkopal 27.11.17 в 23:49

Соглашусь. В конечном итоге Вы все равно правы.
Еще раз спасибо за дополнение. :)

POPSuL 28.11.17 в 02:42

Можно же `.+@.+` =)

Sovetnikov 27.11.17 в 23:49

Каждый раз глядя на своё программное решение с регулярными выражениями для разбора строк, чувствую, что единственное полезное выражение это «что-то (\d+)» и ничего другого лучше не писать.

Очень сложно поддерживать регулярные выражения в программе, всегда в первоначальный «простой» вариант вносятся изменения и усовершенствования. В итоге получается мини-монстр.
Как ящик пандоры — такие надежды всегда, а получается не совсем то.

На мой взгляд сама история появления регулярных выражений определяет их пределы использования. Они же родились в научной среде к некое теоретическое изыскание вылившееся в такой вот язык (утрирую конечно) и на практике оно впервые появилось в текстовом редакт чтобы можно было делать более сложные текстовые замены. И это стихия регулярных выражений.
Ты сделал замену регулярным выражением и сразу оценил результат, если он тебя устроил то регулярное выражение может кануть в забь т.к. оно выполнило свою роль. Это действительно бывает полезно.

nizkopal 27.11.17 в 23:50

В одной старой шутке говорится: если у вас есть проблема, и вы собираетесь решать ее с использованием регулярных выражений, то у есть две проблемы.

AstarothAst 28.11.17 в 09:52

Какая же это шутка?

jamepock 28.11.17 в 15:54

Звучит эта шутка так: «Если у вас есть проблема, и вы собираетесь решать ее с использованием регулярных выражений, то у вас е две проблемы.» :)

И это не шутка.

AstarothAst 29.11.17 в 09:08

Вообще я имел ввиду, что это нифига не шутка, а суровая правда жизни, а вовсе не просил рассказать мне эту шутку :)

netmels 28.11.17 в 00:19

На собеседовании как-то дали задание определить, что все фигурные скобки в тексте (кусок кода) расположены по правилам и закрыты. Н смог...

nizkopal 28.11.17 в 00:20

Определить, используя регулярные выражения?

ArVaganov 28.11.17 в 08:07

Выглядит как эта задача.
www.hackerrank.com/challenges/ctci-balanced-brackets/problem
Регулярные выражения там может и применимы, но решается проблема насколько я помню через хешмап.

aleksandy 28.11.17 в 08:47

Задачу можно решить через инкремент/декремент целого числа, если на выходе не равно нулю, то скобки несбалансированы. Слу когда закрывающая идёт раньше открывающей, обрабатывается условием, что декрементировать 0 нельзя.

ArVaganov 28.11.17 в 09:58

В случае одного вида скобок — да это будет супер элегантным решением.
Если несколько видов скобок, то по вашему алгоритму пройдет конструкция ([ )], хотя она несбалансированая.

aleksandy 28.11.17 в 11:08

Если видов скобок несколько, то, естественно, алгоритм не сработает.

Но в комментарии стояла задача найти

> все фигурные скобки в тексте

, т.е. только '{' и '}'.

akryukov 28.11.17 в 11:56

Для разных видов скобок будет полезно применить стек, в который помещать открывающие скобки. При закрытии соответствующей скобки вынимать ее из стека. Выдавать ошибку, если закрывается не та скобка, которая верхняя в стеке.

netmels 28.11.17 в 12:23

Можно было решать любым способом, но, почему-то, стал думать в этом направлении...

GlukKazan 28.11.17 в 13:41

Это не очень удачное направление. Регулярные выражения (если не рассматривать некоторые расширения) непригодны для нормальной обработки рекурсивно вложенных структур.

pyJloH 28.11.17 в 13:42

Если с ходу сложно вспомнить регулярки (а я всегда перед их использованием гуглю), то на собеседовании нужно было переключи на другой способ решения и попробовать его. Важен не только конечный решенный вариант, но поведение при сложностях с задач

netmels 28.11.17 в 14:54

Спасибо! Да, вернуться бы на несколько лет назад и сказать это себе...
Подобные статьи по области применения регулярок очень полезны новичкам и не только, как мне кажется.

vadimturkov 28.11.17 в 11:56

Задачи со скобками как правило решаются с помощью стека же

bopoh13 28.11.17 в 01:45

> Все дальнейшие примеры предлагаю смотреть в `https://regex101.com/`.

Два аргумента за использование regexr.com
1. На теле по ошибке добавить p — попадёшь на вирусняк с редиректом
2. Не работает в FF с выключенным параметром `dom.workers.enabled` (как и приват в avito)

aleksandy 28.11.17 в 08:45

> `[password|пароль]`

Неправильная регулярка. Квадратные скобки должны быть заменены на круглые.

```
>['password', 'пароль', 'passwпароль', 'рольпа', 'drowssap'].map(w => /[password|пароль]/.test(w));
Array [ true, true, true, true, true ]
```

nizkopal 28.11.17 в 12:16

Верно, спасибо! Исправлено.

deniss-s 28.11.17 в 11:56

1. Регулярные выражения надо знать.
2.

> Some people, when confronted with a problem, think «I know, I'll use regular expressions.» Now they have two problems.

--Jamie Zawinsk

Olehor 28.11.17 в 12:31

Некоторые люди во время решения некой проблемы думают: «Почему бы мне не использовать регулярные выражения?». После этого у ни
две проблемы...
Jamie Zawinski

Londoner 28.11.17 в 15:31

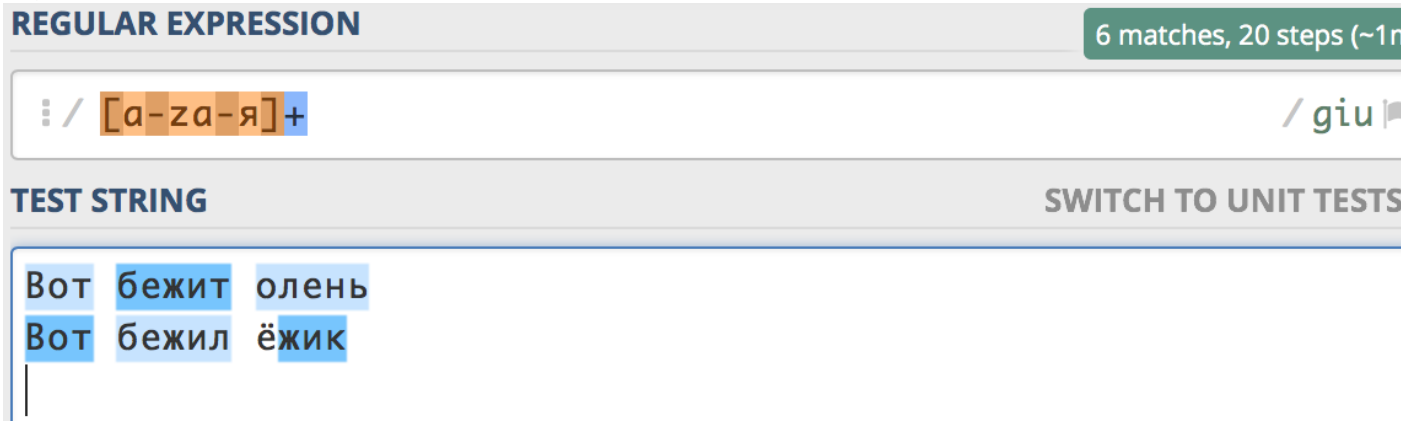
Когда-то на хабре видел статью про билибиотеку, строющую регулярные выражения из вызовов вроде regexp=new
RegExp().startsWith(«aaa»).oneOf(«bbb»,«ccc»).oneOrMore(digit).endsWith(endOfLine);
Но не могу теперь найти. Кто помнит, как она называлась?

happyproff 28.11.17 в 16:25

https://github.com/gherkins/regexpbuilderphp или https://github.com/VerbalExpressions/PHPVerbalExpressions

ezh 28.11.17 в 16:01

Я просто оставлю это здесь



[^\s]+ эквивалентно \S+

nizkopal 28.11.17 в 16:05

Спасибо. Да, я писал, что буква “ё” не входит в диапазон [а-я].

> С кириллицей указанный диапазон работает по-разному для разных кодировок. В юникоде, например, в этот диапазон не входит буква
> “ё”. Подробнее об этом тут.

Drag13 28.11.17 в 16:21

Стоит добавить что флаг /g может вызвать весьма необычное поведение

```
const reg = /^[1-9]+\d*$/g;
reg.test('11'); // true;
reg.test('11'); //false;
```

nizkopal 28.11.17 в 16:32

Да, это особенность JS:

> The RegExp object keeps track of the lastIndex where a match occurred, so on subsequent matches it will start from the last used index, instea
> 0.

Спасибо, интересный момент.

redmanmale 28.11.17 в 16:56

Классика про парсинг HTML регулярками.

Inerren 28.11.17 в 17:09

Для начала: спасибо за подробную статью, сохраню, чтобы выдавать интересующимся для ознакомления.

Позволю себе только немного уточнить (здесь, т.к. это как раз может повлиять на новичков, которые часто берут примеры сразу в работу)

> Но если мы укажем регулярное выражение “/\d/”, то нам вернётся только первая цифра. Мы, конечно, можем использовать модификатор “i”...

Видимо модификатор: «g»?

> Или задать “от” и “до”, указав вот так: “{N, M}”.

Которые также можно задавать по отдельности вида “{N,}” или “{,M}”. Тоже полезная функция.

И ещё я бы упомянул про пассивные группы "(?:test)", которые не попадают в результат позволяют ограничить его только искомыми «полезными» группами.

Ну и мне очень часто помогает вот этот файлик.

nizkopal 28.11.17 в 17:10

Спасибо. Про «g» исправил. Поспешил, видимо, когда писал.

Остальное — тоже полезное дополнение. Нарушать структуру повествования уже поздно, но тут они будут ждать своего внимательного читателя. :)

XNoNAME 29.11.17 в 08:50

ololo@mail.ru — хозяину этого адреса сейчас икается, наверно. А статья годная, спасибо.

nizkopal 29.11.17 в 15:54

Да, стоило быть избирательнее. Вдруг человек этот email использует для бизнеса.
Исправил, спасибо!

GGybarev 29.11.17 в 10:52

Спасибо, как начинающему front-end`y — отличное пособие!

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

«Хочешь быть системным архитектором? Там только свет и чистота...»

+65 21,5k 122 67

У CDROM'а век не долог?

+18 14k 19 170

Как мы делали гигантский объём связи для немаленькой охранной структуры

+32 8k 42 15

Как ты реализуешь аутентификацию, приятель?

+68 20,3k 289 53

«Автоматический детектор спама». Или «О чем предупреждали Хемингуэй, Хаксли и Постман?»

+32 7,5k 57 9

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Да, PVS-Studio умеет выявлять утечки памяти

+7 107 1 0

Чем токен Hamster Marketplace отличается от токенов других проектов?

+6 235 2 0

Opensource нам строить и жить помогает: интервью разработчика Yii Framework Александра Макарова

+7 630 6 4

Впечатления разработчиков и дизайнеров от iPhone X — и от выреза

+9 2,4k 11 13

Avito Product Analytics Meetup 9 декабря: анонс

+14 429 6 0