Регулярные выражения

Что такое регулярные выражения

Регулярные выражения (regular expressions, RegExp) — наборы символов, применяемых для поиска текстовых строк, соответствующих требуемым условиям. Результат применения регулярного выражения — подмножество данных, отобранное согласно логике, заложенной в выражении. Регулярные выражения применяются в любых задачах по поиску в множестве данных, для которых нужно получать выжимку по определенным правилам.

Зачем нужны регулярные выражения?

Сравнение с шаблоном: Регулярные выражения отлично помогают определять, соответствует ли строка тому или иному формату – например, телефонному номеру, адресу электронной почты или номеру кредитной карты.

Замена: При помощи регулярных выражений легко находить и заменять шаблоны в строке. Так, выражение text.replace(/\s+/g, " ") заменяет все пробелы в text, например, " \n\t ", одним пробелом.

Извлечение: При помощи регулярных выражений легко извлекать из шаблона фрагменты информации. Например, name.matches(/^(Mr|Ms|Mrs|Dr)\.?\s/i)извлекает из строки обращение к человеку, например, "Mr" из "Mr. Schropp".

- **Портируемость**: Почти в любом распространенном языке программирования есть своя библиотека регулярных выражений. Синтаксис в основном стандартизирован, поэтому вам не придется переучиваться регулярным выражениям при переходе на новый язык.
- **Код**: Когда пишете код, можно пользоваться регулярными выражениями для поиска информации в файлах; так, в Atom для этого предусмотрен find and replace, а в командной строке — ack.
- **Четкость и лаконичность**: Если вы с регулярными выражениями на «ты», то сможете выполнять весьма нетривиальные операции, написав минимальный объем кода.

Синтаксис регулярных выражений

Большинство символов в регулярных выражениях представляют сами себя, за исключением группы специальных символов «[ ] \ / ^ $. | ? * + ( ) { }». Если эти символы нужно представить в качестве символов текста, их следует экранировать обратной косой чертой «\».

Если эти спецсимволы встречаются без обратной косой черты, значит у них особенные значения в регулярных выражениях:

- «^» — каретка, циркумфлекс или просто галочка. Начало строки;
- «$» — знак доллара. Конец строки;
- «.» — точка. Любой символ;
- «*» – знак умножения, звездочка. Любое количество предыдущих символов;
- «+» – плюс. 1 или более предыдущих символов;
- «?» – вопросительный знак. 0 или 1 предыдущих символов;
- «( )» – круглые скобки. Группировка конструкций;
- «|» – вертикальная линия. Оператор «ИЛИ»;
- «[ ]» – квадратные скобки. Любой из перечисленных символов, диапазон. Если первый символ в этой конструкции – «^», то массив работает наоборот – проверяемый символ не должен совпадать с тем, что перечислено в скобках;
- «{ }» – фигурные скобки. Повторение символа несколько раз;
- «\» – обратный слеш. Экранирование служебных символов.

Также существуют специальные метасимволы, ими можно заменить некоторые готовые конструкции:

- \b — обозначает не символ, а границу между символами;
- \d — цифровой символ;
- \D — нецифровой символ;
- \s — пробельный символ;
- \S — непробельный символ;
- \w — буквенный или цифровой символ или знак подчеркивания;
- \W — любой символ, кроме буквенного или цифрового символа или знака подчеркивания.

Введение в синтаксис PCRE

Синтаксис шаблонов, используемых в функциях этого раздела, во многом похож на синтаксис, используемый в Perl. Выражение должно быть заключено в разделители, например, прямые слеши '/'. Разделителем могут выступать произвольные символы, кроме буквенно-цифровых, обратного слеша '\' и нулевого байта. Если символ разделителя встречается в шаблоне, его необходимо экранировать. В качестве разделителя доступны комбинации, используемые в Perl: (), {}, [] и <>.

Далее перечислены все доступные на сегодняшний день модификаторы. Имя, взятое в круглые скобки, указывает внутреннее PCRE-имя для данного модификатора. Пробелы и переводы строк в модификаторах игнорируются, другие символы вызывают ошибки.

i (*PCRE_CASELESS*)

Если этот модификатор используется, символы в шаблоне соответствуют символам как верхнего, так и нижнего регистра.

m (*PCRE_MULTILINE*)

По умолчанию PCRE обрабатывает данные как однострочную символьную строку (даже если она содержит несколько разделителей строк). Метасимвол начала строки '^' соответствует только началу обрабатываемого текста, в то время как метасимвол "конец строки" '$' соответствует концу текста, либо позиции перед завершающим текст переводом строки (в случае, если модификатор *D* не установлен). В Perl ситуация полностью аналогична. Если этот модификатор используется, метасимволы "начало строки" и "конец строки" также соответствуют позициям перед произвольным символом перевода и строки и, соответственно, после, как и в самом начале и в самом конце строки. Это соответствует Perl-модификатору /m. В случае, если обрабатываемый текст не содержит символов перевода строки, либо шаблон не содержит метасимволов '^' или '$', данный модификатор не имеет никакого эффекта.

s (*PCRE_DOTALL*

Если данный модификатор используется, метасимвол "точка" в шаблоне соответствует всем символам, включая перевод строк. Без него - все символы, кроме переводов строк. Этот модификатор эквивалентен записи /s в Perl. Класс символов, построенный на отрицании, например [^a], всегда соответствует переводу строки, независимо от наличия этого модификатора.

x (*PCRE_EXTENDED*)

Если используется данный модификатор, неэкранированные пробелы, символы табуляции и пустой строки будут проигнорированы в шаблоне, если они не являются частью символьного класса. Также игнорируются все символы между неэкранированным символом '#' (если он не является частью символьного класса) и символом перевода строки (включая сами символы '\n' и '#'). Это эквивалентно Perl-модификатору /x, и позволяет размещать комментарий в сложных шаблонах. Замечание: это касается только символьных данных. Пробельные символы не фигурируют в служебных символьных последовательностях, к примеру, в последовательности '(?(', открывающей условную подмаску.

e (*PREG_REPLACE_EVAL*)

Если используется данный устаревший модификатор, preg_replace() после выполнения стандартных подстановок в заменяемой строке интерпретирует ее как PHP-код и использует результат для замены искомой строки. Одинарные и двойные кавычки, обратные слеши (\) NULL-символы будут проэкранированы обратными слешами в подставляемых обратных ссылках.

A (*PCRE_ANCHORED*)

Если используется данный модификатор, соответствие шаблону будет достигаться только в том случае, если он "заякорен", то есть соответствует началу строки, в которой производится поиск. Того же эффекта можно достичь подходящей конструкцией с вложенным шаблоном, которая является единственным способом реализации этого поведения в Perl.

D (*PCRE_DOLLAR_ENDONLY*)

Если используется данный модификатор, метасимвол $ в шаблоне соответствует только окончанию обрабатываемых данных. Без этого модификатора метасимвол $ соответствует также позиции перед последним символом, в случае, если им является перевод строки (но не распространяется на любые другие переводы строк). Данный модификатор игнорируется, если используется модификатор *m*. В языке Perl аналогичный модификатор отсутствует.

S

В случае, если планируется многократно использовать шаблон, имеет смысл потратить немного больше времени на его анализ, чтобы уменьшить время его выполнения. В случае, если данный модификатор используется, проводится дополнительный анализ шаблона. В настоящем это имеет смысл только для "незаякоренных" шаблонов, не начинающихся с какого-либо определенного символа.

U (PCRE_UNGREEDY

Этот модификатор инвертирует жадность квантификаторов, таким образом они по умолчанию не жадные. Но становятся жадными, если за ними следует символ ?. Такая возможность не совместима с Perl. Его также можно установить с помощью (?U) установки модификатора внутри шаблона или добавив знак вопроса после квантификатора (например, .*?).

Замечание:

В нежадном режиме обычно невозможно совпадение символов превышающих pcre.backtrack_limit.

X (PCRE_EXTRA)

Этот модификатор включает дополнительную функциональность PCRE, которая не совместима с Perl: любой обратный слеш в шаблоне, за которым следует символ, не имеющий специального значения, приводят к ошибке. Это обусловлено тем, что подобные комбинации зарезервированы для дальнейшего развития. По умолчанию же, как и в Perl, слеш со следующим за ним символом без специального значения трактуется как опечатка. На сегодняшний день это все возможности, которые управляются данным модификатором

J (PCRE_INFO_JCHANGED)

Модификатор (?J) меняет значение локальной опции *PCRE_DUPNAMES* - подшаблоны могут иметь одинаковые имена. Модификатор *J*поддерживается с версии PHP 7.2.0.

u (PCRE_UTF8)

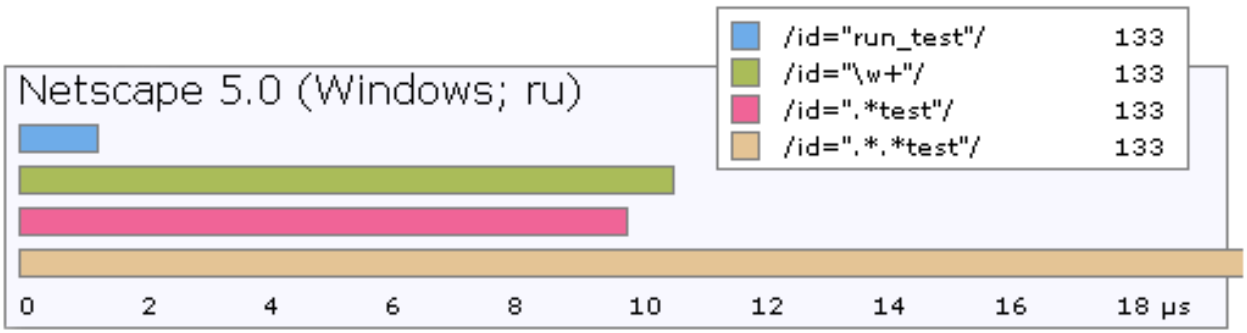
Этот модификатор включает дополнительную функциональность PCRE, которая не совместима с Perl: шаблон и целевая строка обрабатываются как UTF-8 строки. Недопустимая целевая строка приводит к тому, что функции preg_* ничего не находят, а неправильный шаблон приводит к ошибке уровня E_WARNING. Пятый и шестой октеты UTF-8 последовательности рассматриваются недопустимыми с PHP 5.3.4 (согласно PCRE 7.3 2007-08-28); ранее они считались допустимыми.

"Жадные" и не "жадные" регулярные выражения.

Жадные (greedy) квантификаторы

В современных регулярных выражениях есть несколько разновидностей замыканий. *Stephen Cole Kleene*, который ввел это понятие, описал два таких: * и +. Как было описано выше, поведение их «жадное» — они пытаются покрыть все что можно — до конца строки. Но дальше в нашем выражении идет следующий оператор или символ, а мы уже в конце строки. Тут парсер откручивает наш квантификатор обратно по точкам возврата, пока не выполнится условие последующего подвыражения.

Очевидно что подобное поведение легко порождает проблемы с производительностью. Вот время выполнения для нескольких вариантов:



Последний случай с двумя звёздочками на самом деле отрабатывает на порядок медленнее. Это связано с особенностью работы парсера. Как было сказано, выражение «любой символ много раз» выполняется дословно и фактически парсер сначала покрывает этим выражением всю строку, сохраняя на каждом символе точку возврата. Увидев что наше выражение не закончено, парсер возвращается обратно, пока не найдет совпадение. Наличие двух звездочек увеличивает

количество точек возврата на порядок, трех — еще на порядок. Легко увидеть что такой путь может «простое выражение» сделать ощутимо медленным.

Есть несколько способов улучшить эффективность:

интервал со стоп-символом.

Например, если мы ищем теги от '<' до '>', то можно указать интервал вместо произвольного символа: /<[^>]+>/

Парсер остановится, увидев символ вне диапазона и сразу же сработает последующий литеральный символ '>'.

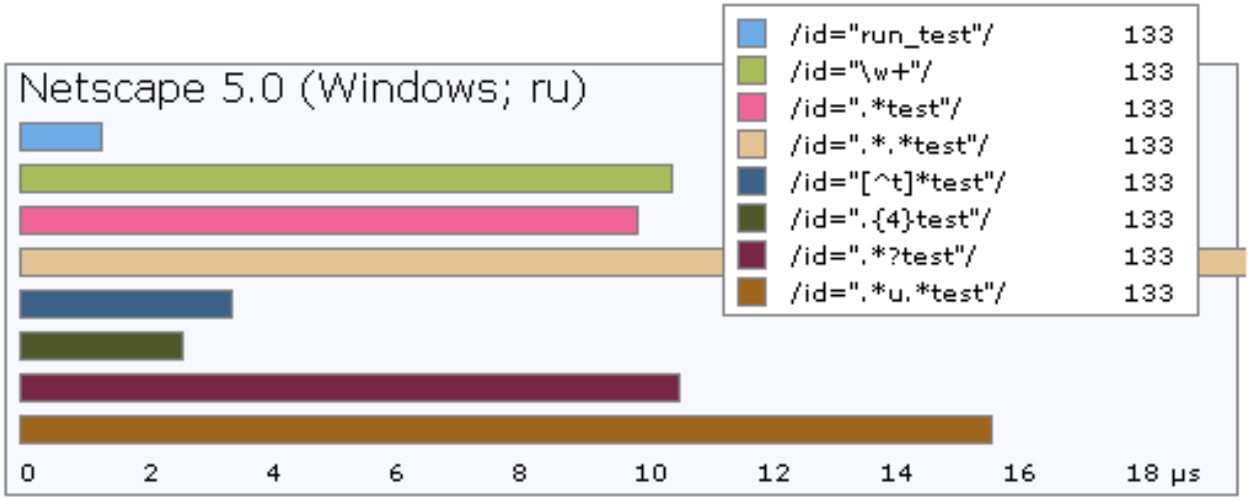
использовать интервал повторений {min,max}

Хорошо работает, если нам известно сколько должно быть символов, например при первичной проверке uid или md5 сигнатур.

Нежадные (non-greedy) или ленивые (lazy) квантификаторы

Такой квантификатор действует наоборот — покрывает минимальный набор символов и расширяет его, если последующие сцепленные выражения не выполняются.

С точки зрения производительности очень хорошо работает для вложенных необязательных выражений, но также как и жадный квантификатор может вызвать существенное замедление, поскольку если в хорошем случае (когда наше выражение выполняется) все более-менее быстро, то в плохом случае парсер пытается увеличить покрытие для нежадного квантификатора, пока не дойдет до очевидного конца. Что-бы избежать этого, желательно максимально сузить покрываемые символы интервалом и сразу после него поставить простое выражение — фиксированные символы, например.



У ленивых «звездочек» и «плюсов» есть еще один недостаток — они могут и очень часто покрывают слишком мало символов, если границу не обозначить последующим подвыражением. Например, если вы разбираете слова так: \w+? то можете обнаружить что без последующего литерального символа (в конце большого выражения), это комбинация покроет только одну букву и в данном случае эффективнее «жадный» вариант. Также жадные эффективнее если четко известно что следующим будет другой символ \w+ так можно описать слово или параметр, ленивый тут просто менее эффективный.

Поиск и замена с применением регулярных выражений

Регулярные выражения мощнейшее средство для поиска в тексте нужных фрагментов и замены их на другие фрагменты. Этот модуль посвящён: подвыражениям и обратным ссылкам; модификаторам регулярных выражений; объекту RegExp; методам объекта String для работы с регулярными выражениями; "экономному" поиску Подвыражения и обратные ссылки.

Мы узнали, что круглые скобки () применяются для группирования последовательности символов в регулярном выражении в единый фрагмент, который после этого начинает вести себя как один символ, и к нему можно будет применить любой квалификатор. (Отметим, что то же самое касается скобок, в которых заключается языковая конструкция вида <<<последовательность 1>>> <<<последовательность 2>>>.) Такой единый фрагмент носит название подвыражения. Подвыражения имеют две очень полезные особенности: 1. Фрагмент текста, совпадающий с подвыражением, запоминается в оперативной памяти, и впоследствии его можно извлечь для использования в сценарии. (Как это сделать, будет рассказано далее.) 2. Мы можем указать, что в

определённом месте регулярного выражения должен присутствовать фрагмент, совпадающий с ранее указанным в том же регулярном выражении подвыражением. Для этого используются так называемые обратные ссылки, которые записываются следующим образом: \<<<порядковый номер подвыражения>>>

Подвыражения нумеруются в порядке слева направо. // Регулярное выражение, совпадающее со строкой, которая включает: // * последовательность из, как минимум, одной латинской буквы a-z, заключённую // в символы < и >, то есть открывающий HTML-тег (<[a-z]+>). // Эта последовательность заключена в круглые скобки и, таким образом, превращена // в подвыражение; // * последовательность из любого количества любых символов (.*); // * последовательность символов, совпадающая с первым подвыражением и также // заключённая в символы </ и >, то есть парный закрывающий HTML-тег (<\/\1>). // Для обращения к первому подвыражению использовалась обратная ссылка \1. var r = /<([a-z]+)>.*<\/\1>/; // Проверяем, совпадает ли это регулярное выражение со строкой " абзац ". // Есть совпадение! var f = r.test(" абзац "); dcument.write(f); // Результат true // А со строкой " абзац " это регулярное выражение не совпадает f = r.test(" абзац "); dcument.write(f); // Результат false.