Бесплатная электронная книга

УЧУСЬ

Regular Expressions

Free unaffiliated eBook created from
Stack Overflow contributors.

#regex

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: regular-expressions

It is an unofficial and free Regular Expressions ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Regular Expressions.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с регулярными выражениями

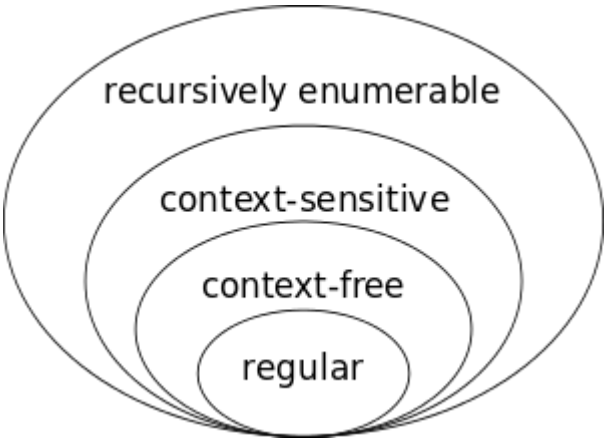
замечания

Для многих программистов *регулярное выражение* - это своего рода волшебный меч, который они бросают, чтобы решить любую ситуацию синтаксического анализа текста. Но этот инструмент не волшебный, и хотя он отлично подходит к тому, что он делает, он не является полнофункциональным языком программирования (*т. Е.* Он **не** является Turing-полным).

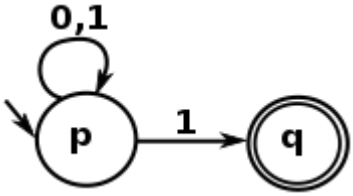
Что означает «регулярное выражение»?

Регулярные выражения выражают язык, определяемый *регулярной грамматикой,* которая может быть решена *недетерминированным конечным автоматом* (NFA), где согласование представлено состояниями.

Регулярная грамматика - самая простая грамматика, выраженная *Иерархией Хомского* .



Проще говоря, регулярный язык визуально выражается тем, что может выразить NFA, и вот очень простой пример NFA:



Язык *регулярных выражений* является текстовым представлением такого автомата. Этот последний пример выражается следующим регулярным выражением:

```
^[01]*1$
```

Которая соответствует любой строке, начинающейся с `0` или `1` , повторяя 0 или более раз, которая заканчивается на `1` . Другими словами, это регулярное выражение, чтобы соответствовать нечетным числам из их двоичного представления.

Являются ли все регулярные выражения *регулярной* грамматикой?

На самом деле это не так. Многие механизмы регулярных выражений улучшены и используют *push-down automata* , которые могут складываться и выгружать информацию по мере ее запуска. Эти автоматы определяют так называемые *контекстно-свободные* грамматики в иерархии Хомского. Наиболее типичным использованием в нерегулярном регулярном *выражении* является использование рекурсивного шаблона для сопоставления скобок.

Рекурсивным регулярным выражением, подобным приведенному ниже (что соответствует скобке), является примером такой реализации:

```
{((?>[^\(\)]+|(?R))*)}
```

(этот пример не работает с питона `re` двигателем, но с `regex` двигателя или с двигателем PCRE).

Ресурсы

Для получения дополнительной информации о теории регулярных выражений вы можете обратиться к следующим курсам, предоставленным MIT:

- Автоматы, вычислимость и сложность
- Регулярные выражения и грамматики
- Указание языков с регулярными выражениями и контекстно-свободные грамматики

Когда вы пишете или отлаживаете сложное регулярное выражение, есть онлайн-инструменты, которые могут помочь визуализировать регулярные выражения как автоматы, такие как сайт debuggex .

Версии

PCRE

Версия	Вышел
2	2015-01-05
1	1997-06-01

Используется: PHP 4.2.0 (и выше), Delphi XE (и выше), Julia , Notepad ++

Perl

Версия	Вышел
1	1987-12-18
2	1988-06-05
3	1989-10-18
4	1991-03-21
5	1994-10-17
6	2009-07-28

.СЕТЬ

Версия	Вышел
1	2002-02-13
4	2010-04-12

Языки: C #

Джава

Версия	Вышел
4	2002-02-06
5	2004-10-04
7	2011-07-07
SE8	2014-03-18

JavaScript

Версия	Вышел
1.2	1997-06-11
1.8.5	2010-07-27

ПИТОН

Версия	Вышел
1.4	1996-10-25
2,0	2000-10-16
3.0	2008-12-03
3.5.2	2016-06-07

Oniguruma

Версия	Вышел
начальный	2002-02-25
5.9.6	2014-12-12
Onigmo	2015-01-20

Увеличение

Версия	Вышел
0	1999-12-14
1.61.0	2016-05-13

POSIX

Версия	Вышел
BRE	1997-01-01
ERE	2008-01-01

Языки: Bash

Examples

Руководство персонажа

Обратите внимание, что некоторые элементы синтаксиса имеют различное поведение в зависимости от выражения.

Синтаксис	Описание
`?`	Сопоставьте предыдущий символ или подвыражение 0 или 1 раз. Также используется для групп без захвата и названных групп захвата.
`*`	Сопоставьте предыдущий символ или подвыражение 0 или более раз.
`+`	Сопоставьте предыдущий символ или подвыражение 1 или более раз.
`{n}`	Сопоставьте предыдущий символ или подвыражение ровно *n* раз.
`{min,}`	Сопоставьте предыдущий символ или подвыражение *мин* или более раз.
`{,max}`	Совпадение предшествующего символа или подвыражения *макс* или меньше.
`{min,max}`	Сопоставьте предыдущий символ или подвыражение как минимум *мин.* Раз, но не более, чем `max` время.
`-`	При включении в квадратных скобках указывает на то, `to` ; например, [3-6] соответствует символам 3, 4, 5 или 6.
`^`	Начало строки (или начало строки, если задана опция многострочного `/m`), или отменяет список параметров (т. Е. В квадратных скобках `[]`)
`$`	Конец строки (или конец строки, если указан параметр multiline `/m`).
`( ... )`	Группирует подвыражения, захватывает совпадающее содержимое в специальных переменных (`\1` , `\2` и т. Д.), Которые могут использоваться позже в одном и том же регулярном выражении, например `(\w+)\s\1\s` соответствует повторению слова

Синтаксис	Описание
`(?<name> ... )`	Групповые подвыражения и захватывает их в именованной группе
`(?: ... )`	Групповые подвыражения без захвата
`.`	Соответствует любому символу, кроме разрывов строк (`\n` и обычно `\r`).
`[ ... ]`	Любой символ между этими скобками следует сопоставлять один раз. NB: `^` после открытой скобки отрицает этот эффект. `-` встречающийся внутри скобок позволяет задавать диапазон значений (если только это не первый или последний символ, и в этом случае он просто представляет собой обычную тире).
`\`	Убирает следующий символ. Также используется в мета-последовательностях - токены регулярного выражения со специальным значением.
`\$`	доллар (т. е. экранированный особый характер)
`\(`	открытая скобка (т. е. экранированный специальный символ)
`\)`	закрыть скобку (т. е. экранированный особый символ)
`\*`	звездочка (т. е. экранированный особый символ)
`\.`	точка (т. е. экранированный особый символ)
`\?`	вопросительный знак (т. е. экранированный специальный символ)
`\[`	левая (открытая) квадратная скобка (т. е. экранированный специальный символ)
`\\`	обратная косая черта (т. е. экранированный особый символ)
`\]`	правая (близкая) квадратная скобка (т.е. экранированный особый символ)
`\^`	карет (т. е. экранированный особый символ)
`\{`	левая (открытая) фигурная скобка / скобка (т.е. экранированный специальный символ)
`|`	труба (т. е. экранированный особый символ)
`\}`	правая (близкая) фигурная скобка / скобка (т.е. экранированный особый символ)

Синтаксис	Описание
`\+`	плюс (т. е. экранированный особый символ)
`\A`	начало строки
`\Z`	конец строки
`\z`	абсолютная строка
`\b`	слово (буквенно-цифровая последовательность)
`\1` , `\2` и т. Д.	обратные ссылки на ранее согласованные подвыражения, сгруппированные по `()` , `\1` означает первое совпадение, `\2` означает второе совпадение и т. д.
`[\b]`	backspace - когда `\b` находится внутри класса символов (`[]`) соответствует обратному пространству
`\B`	negated `\b` - соответствует любой позиции между двумя словами, а также в любой позиции между двумя символами, отличными от слова
`\D`	нецифровой
`\d`	цифра
`\e`	побег
`\f`	форма подачи
`\n`	line feed
`\r`	возврат каретки
`\S`	небелый-пространство
`\s`	бело-пространство
`\t`	табуляция
`\v`	вертикальная вкладка
`\W`	без слов
`\w`	слово (например, буквенно-цифровой символ)
`{ ... }`	набор символов
`|`	или же; т.е. определяет предыдущие и предыдущие варианты.

Прочитайте Начало работы с регулярными выражениями онлайн:
https://riptutorial.com/ru/regex/topic/259/начало-работы-с-регулярными-выражениями

глава 2: Lookahead и Lookbehind

Синтаксис

- **Положительный взгляд:** `(?=pattern)`
- **Отрицательный взгляд:** `(?!pattern)`
- **Положительный lookbehind** : `(?<=pattern)`
- **Отрицательный lookbehind** : `(?<!pattern)`

замечания

Не поддерживается всеми двигателями регулярных выражений.

Кроме того, многие механизмы регулярных выражений ограничивают шаблоны внутри lookbehinds строками фиксированной длины. Например, шаблон `(?<=a+)b aaab (?<=a+)b` должен соответствовать `b` в `aaab` но вызывает ошибку в Python.

Захватывающие группы разрешены и работают так, как ожидалось, включая обратные ссылки. Однако сам lookahead / lookbehind не является группой захвата.

Examples

ОСНОВЫ

Положительный lookahead `(?=123)` утверждает, что за текстом следует данный шаблон, без включения шаблона в совпадение. Аналогично, **положительный lookbehind** `(?<=123)` утверждает, что этому тексту предшествует данный шаблон. Заменив `=` с `!` отрицает утверждение.

Вход : `123456`

- `123(?=456)` соответствует `123` (*положительный результат*)
- `(?<=123)456` совпадений `456` (*положительный lookbehind*)
- `123(?!456)` не работает (*отрицательный результат*)
- `(?<!123)456` терпит неудачу (*отрицательный lookbehind*)

Вход : `456`

- `123(?=456)` не работает
- `(?<=123)456` не удается
- `123(?!456)` не работает

- `(?<!123)456` совпадений `456`

Использование lookbehind для тестирования окончаний

В конце шаблона можно использовать lookbehind, чтобы обеспечить его окончание или нет.

`([az ]+|[AZ ]+)(?<! )` соответствует последовательностям только строчных или только прописных слов, исключая конечные пробелы.

Имитация с переменной длиной lookbehind с \ K

Некоторые ароматы регулярных выражений (Perl, PCRE, Oniguruma, Boost) поддерживают только lookbehind с фиксированной длиной, но предлагают функцию `\K` , которая может использоваться для имитации зависания переменной длины в начале шаблона. При столкновении с `\K` согласованный текст до этой точки отбрасывается, и только текст, соответствующий части шаблона, *следующей за* `\K` , сохраняется в конечном результате.

```
ab+\Kc
```

Эквивалентно:

```
(?<=ab+)c
```

В общем, шаблон вида:

```
(subpattern A)\K(subpattern B)
```

Концы заканчиваются похожими на:

```
(?<=subpattern A)(subpattern B)
```

За исключением случаев, когда подшаблон B может совпадать с тем же текстом, что и подзаголовок A, вы можете получить незначительно разные результаты, потому что подшаблон A все еще потребляет текст, в отличие от истинного lookbehind.

Прочитайте Lookahead и Lookbehind онлайн:
https://riptutorial.com/ru/regex/topic/639/lookahead-и-lookbehind

глава 3: UTF-8: письма, знаки, знаки препинания и т. Д.

Examples

Соответствующие буквы в разных алфавитах

Примеры, приведенные ниже, приведены в Ruby, но такие же совпадения должны быть доступны на любом современном языке.

Скажем, у нас есть строка `"A𝔞Naïve"` , созданная Messy Artificial Intelligence. Он состоит из букв, но общий мануал `\w` не будет соответствовать многому:

```
▶ "A𝔞Naïve"[/\w+/]
#⇒ "A"
```

Правильный способ сопоставления букв Unicode с комбинированием меток заключается в использовании `\X` для указания кластера графем. Однако есть предостережение для Ruby. Onigmo, механизм регулярных выражений для Ruby, по-прежнему использует старое определение кластера графем . Он еще не обновлен до расширенного кластера Grapheme, как определено в стандарте Unicode Standard 29 .

Итак, для Ruby у нас может быть обходное решение: `\p{L}` будет делать почти отлично, за исключением того, что он не работает при комбинированном диакритическом акценте на `i` :

```
▶ "A𝔞Naïve"[/\p{L}+/]
#⇒ "A𝔞Nai"
```

Добавив в выражение символы «Mark», мы можем, наконец, сопоставить все:

```
▶ "A𝔞Naïve"[/[\p{L}\p{M}]+/]
#⇒ "A𝔞Naïve"
```

Прочитайте UTF-8: письма, знаки, знаки препинания и т. Д. онлайн: https://riptutorial.com/ru/regex/topic/1527/utf-8--письма--знаки--знаки-препинания-и-т--д-

глава 4: Атомная группировка

Вступление

Регулярные группы без захвата позволяют двигателю повторно вводить группу и пытаться сопоставить что-то другое (например, различное чередование или меньше символов, когда используется квантификатор).

Атомные группы отличаются от регулярных не-захватных групп тем, что откат запрещен. Как только группа выйдет, вся информация об отступлении отбрасывается, поэтому никаких альтернативных совпадений не может быть предпринята.

замечания

Притяжательный квантификатор ведет себя как атомная группа в том, что движок не сможет вернуться к токену или группе.

Следующие эквиваленты с точки зрения функциональности, хотя некоторые из них будут быстрее других:

```
a*+abc
(?>a*)abc
(?:a+)*+abc
(?:a)*+abc
(?:a*)*+abc
(?:a*)++abc
```

Examples

Группировка с (?>)

Использование атомной группы

Атомные группы имеют формат `(?>...)` с `?>` После открытого пара.

Рассмотрим следующий образец текста:

```
ABC
```

Регулярное выражение будет пытаться сопоставить начало в позиции 0 текста, который находится перед `A` в `ABC` .

Если бы использовалось нечувствительное к регистру выражение `(?>a*)abc` `A` `(?>a*)abc` , то

`(?>a*) A (?>a*)` бы символу 1 `A` , оставив

```
BC
```

как оставшийся текст для соответствия. Группа `(?>a*) A (?>a*)` завершается, а `abc` пытается `(?>a*)` оставшийся текст, который не соответствует.

Двигатель не может возвратиться в атомную группу, и поэтому текущий проход терпит неудачу. Двигатель переместится в следующую позицию в тексте, который будет находиться в позиции 1, которая находится после `A` и перед `B` из `ABC` .

Повторное выражение `(?>a*)abc A (?>a*)abc` снова выполняется, и `(?>a*) A (?>a*)` соответствует `A` 0 раз, оставляя

```
BC
```

как оставшийся текст для соответствия. Вызывается группа `(?>a*) A (?>a*)` и предпринимается попытка `abc` , которая терпит неудачу.

Опять же, двигатель не может вернуться в атомную группу, и поэтому текущий проход терпит неудачу. Регулярное выражение будет продолжать сбой, пока все позиции в тексте не будут исчерпаны.

Использование неатомной группы

Регулярные группы, не участвующие в захвате, имеют формат `(?:...)` с `?:` После открытого пара.

При использовании одного и того же текста примера, но вместо выражения с нечувствительным к регистру `(?:a*)abc` будет происходить совпадение, так как может произойти обратное отслеживание.

Сначала `(?:a*) A (?:a*)` будет использовать букву `A` в тексте

```
ABC
```

уход

```
BC
```

как оставшийся текст для соответствия. Группа `(?:a*) A (?:a*)` завершается, а `abc` пытается `(?:a*)` оставшийся текст, который не соответствует.

Двигатель возвращается в группу `(?:a*) A (?:a*)` и пытается совместить 1 меньшее число символов: вместо сопоставления 1 символа `A` он пытается совместить символы 0 `A` , и

группа `(?:a*) A (?:a*)` завершается. Это оставляет

```
ABC
```

как оставшийся текст для соответствия. Регулярное выражение `abc` теперь может успешно сопоставлять оставшийся текст.

Другой пример текста

Рассмотрим этот образец текста как с атомными, так и с неатомными группами (опять же, без учета регистра):

```
AAAABC
```

Регулярное выражение будет пытаться сопоставить начало в позиции 0 текста, которое находится перед первым `A` в `AAAABC` .

Образец, использующий атомную группу `(?>a*)abc A (?>a*)abc` , не **сможет** сравниться, ведя почти одинаково с приведенным выше примером атомной `ABC` : все 4 из символов `A` сначала сопоставляются с `(?>a*) A (?>a*)` (оставляя `BC` как оставшийся текст соответствует), а `abc` не может соответствовать этому тексту. Группа **не** может быть повторно введена, поэтому совпадение не выполняется.

Образец, использующий неатомную группу `(?:a*)abc A (?:a*)abc` , **сможет** сопоставляться, действуя аналогично приведенному выше `ABC` примеру `ABC` : все 4 из символов `A` сначала сопоставляются с `(?:a*) A (?:a*)` (оставляя `BC` поскольку оставшийся текст соответствует), и `abc` не может соответствовать этому тексту. Группа может быть повторно введено, так один меньше попытка: 3 `A A` символы совпадают вместо 4 (оставляя `ABC` как остальной текст , чтобы соответствовать), и `abc` способны успешно соответствовать по этому тексту.

Прочитайте Атомная группировка онлайн: https://riptutorial.com/ru/regex/topic/8770/атомная-группировка

глава 5: Граница Word

Синтаксис

- Стиль POSIX, конец слова: `[[:>:]]`
- Стиль POSIX, начало слова: `[[:<:]]`
- Стиль POSIX, граница слова: `[[:<:][:>:]]`
- SVR4 / GNU, конец слова: `\>`
- SVR4 / GNU, начало слова: `\<`
- Perl / GNU, граница слова: `\b`
- Tcl, конец слова: `\M`
- Tcl, начало слова: `\m`
- Tcl, граница слова: `\y`
- Portable ERE, начало слова: `(^|[^[:alnum:]_])`
- Portable ERE, конец слова: `([^[:alnum:]_]|$)`

замечания

Дополнительные ресурсы

- Глава POSIX о регулярных выражениях
- Документация по регулярному выражению Perl
- Страница руководства Tcl re_syntax
- Выражения обратной ссылки GNU grep
- BSD re_format
- Больше информации

Examples

Матч полного слова

```
\bfoo\b
```

будет соответствовать полному слову без алфавитно-цифровых и _ предшествующих или последующих им.

Взятие из regularexpression.info

> Существуют три разных позиции, которые квалифицируются как границы слов:
>
> 1. Перед первым символом в строке, если первый символ является символом

слова.
2. После последнего символа в строке, если последний символ является символом слова.
3. Между двумя символами в строке, где один является символом слова, а другой не является символом слова.

Термин « *слово»* здесь означает любое из следующих

1. Алфавит (`[a-zA-Z]`)
2. Номер (`[0-9]`)
3. Подчеркивание _

Короче говоря, *символ слова* = `\w` = `[a-zA-Z0-9_]`

Поиск шаблонов в начале или в конце слова

Изучите следующие строки:

```
foobarfoo
bar
foobar
barfoo
```

- `bar` регулярного выражения будет соответствовать всем четырем строкам,
- `\bbar\b` будет соответствовать только 2,
- `bar\b` будет соответствовать 2-й и 3-й строкам и
- `\bbar` будет соответствовать 2-й и 4-й строкам.

Границы слов

Метасимвол `\b`

Чтобы упростить поиск целых слов, мы можем использовать метасимвол `\b` . Он отмечает **начало** *и* **конец** буквенно-цифровой последовательности *. Кроме того, поскольку он служит только для обозначения этих местоположений, он фактически не соответствует ни одному символу.

**: Обычно называют буквенно-цифровую последовательность словом, так как мы можем поймать ее символы с `\w` (класс слов символов). Это может ввести в заблуждение, так как `\w` также включает числа и, в большинстве случаев, подчеркивание.*

Примеры:

Regex	вход	Матчи?
`\bstack\b`	`stackoverflow`	**Нет** , поскольку нет никакого смысла **всего** слова `stack`
`\bstack\b`	`foo stack bar`	**Да** , поскольку нет ничего раньше или после `stack`
`\bstack\b`	`stack!overflow`	**Да** : перед `stack` нет ничего `!` не является символом слова
`\bstack`	`stackoverflow`	**Да** , поскольку перед `stack` ничего нет
`overflow\b`	`stackoverflow`	**Да** , поскольку после `overflow` ничего нет

Метасимвол `\B`

Это противоположно `\b` , соответствующему местоположению каждого неграничного символа. Подобно `\b` , поскольку он соответствует местоположениям, он не соответствует ни одному символу. Это полезно для поиска *не* целочисленных слов.

Примеры:

Regex	вход	Матчи?
`\Bb\B`	`abc`	**Да** , поскольку `b` не окружен границами слов.
`\Ba\B`	`abc`	**Нет** , `a` имеет границу слова с левой стороны.
`a\B`	`abc`	**Да,** не имеет границ слова на его правой стороне. `a`
`\B,\B`	`a,,,b`	**Да** , он соответствует второй запятой, потому что `\B` также будет соответствовать пробелу между двумя символами без слова (следует отметить, что есть левая граница слева от первой запятой и справа от второй).

Сделайте текст короче, но не сломайте последнее слово

Чтобы сделать длинный текст длиной не более N символов, но оставить неизменным последнее слово, используйте `.{0,N}\b` pattern:

```
^(.{0,N})\b.*
```

Прочитайте Граница Word онлайн: https://riptutorial.com/ru/regex/topic/1539/граница-word

глава 6: Группы захвата

Examples

Основные группы захвата

Группа представляет собой раздел регулярного выражения, заключенного в круглые скобки `()` . Это обычно называется «суб-выражением» и служит двум целям:

- Это делает субэкспрессией атомом, т. Е. Он будет либо соответствовать, либо терпеть неудачу, либо повторять в целом.
- Часть текста, которую он сопоставляет, доступна в остальной части выражения и в остальной части программы.

Группы пронумерованы в двигателях регулярных выражений, начиная с 1. Традиционно максимальное число групп равно 9, но многие современные ароматы регулярных выражений поддерживают более высокие групповые подсчеты. Группа 0 всегда соответствует всему шаблону, так же, как и окружающее все регулярное выражение с помощью скобок.

Порядковый номер увеличивается с каждой открывающей скобкой независимо от того, размещены ли группы один за другим или вложенные:

```
foo(bar(baz)?) (qux)+|(bla)
   1   2       3      4
```

групп и их номеров

После того, как выражение достигнет общего соответствия, все его группы будут использоваться - независимо от того, удалось ли какой-то конкретной группе сопоставить что-либо или нет.

Группа может быть необязательной, например `(baz)?` выше или в альтернативной части выражения, которое не использовалось для совпадения, например `(bla)` выше. В этих случаях несогласованные группы просто не будут содержать никакой информации.

Если квантификатор помещен за группой, как и в `(qux)+` выше, общий групповой счет выражения остается неизменным. Если группа соответствует более одного раза, ее содержимое будет последним совпадением. Тем не менее, современные ароматы регулярных выражений позволяют получить доступ ко всем событиям, связанным с повторением матчей.

Если вы хотите получить дату и уровень ошибки в записи журнала, как этот:

```
2012-06-06 12:12.014 ERROR: Failed to connect to remote end
```

Вы можете использовать что-то вроде этого:

```
^(\d{4}-\d{2}-\d{2}) \d{2}:\d{2}.\d{3} (\w*): .*$
```

Это позволит извлечь дату записи журнала `2012-06-06` качестве группы захвата 1 и `ERROR` уровня ошибки в качестве группы захвата 2.

Обратные ссылки и группы, не связанные с захватом

Поскольку группы «пронумерованы», некоторые двигатели также поддерживают соответствие тому, что группа ранее сопоставляла снова.

Предполагая, что вы хотите сопоставить что-то, где два равны строкам длины три, делятся на `$` вы использовали:

```
(.{3})\$\1
```

Это будет соответствовать любой из следующих строк:

```
"abc$abc"
"a b$a b"
"af $af "
"   $   "
```

Если вы хотите, чтобы группа не была пронумерована движком, вы можете объявить ее не захваченной. Не захватывающая группа выглядит следующим образом:

```
(?:)
```

Они особенно полезны для повторения определенной картины любое количество раз, так как группа также может использоваться как «атом». Рассматривать:

```
(\d{4}(?:-\d{2}){2} \d{2}:\d{2}.\d{3}) (.*)[\r\n]+\1 \2
```

Это будет соответствовать двум записям журнала в смежных строках, которые имеют одну и ту же метку времени и одну и ту же запись.

Именованные группы захвата

Некоторые ароматы регулярных выражений позволяют *назвать группы захвата* . Вместо числового индекса вы можете ссылаться на эти группы по имени в следующем коде, то есть в обратных ссылках, в шаблоне замены, а также в следующих строках программы.

Числовые индексы меняются по мере изменения количества или расположения групп в

выражении, поэтому они более хрупкие в сравнении.

Например, чтобы сопоставить слово (`\w+`), заключенное в одинарные или двойные кавычки (`['"]`), мы могли бы использовать:

```
(?<quote>['"])\w+\k{quote}
```

Это эквивалентно:

```
(['"])\w+\1
```

В простой ситуации, подобной этой, регулярная, пронумерованная группа захвата не имеет никаких обратных связей.

В более сложных ситуациях использование названных групп сделает структуру выражения более очевидной для читателя, что улучшит ремонтопригодность.

Синтаксический анализ файла журнала является примером более сложной ситуации, которая выигрывает от имен групп. Это общий формат журнала Apache (CLF):

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

Следующее выражение захватывает части в названные группы:

```
(?<ip>\S+) (?<logname>\S+) (?<user>\S+) (?<time>\[[^]]+\]) (?<request>"[^"]+") (?<status>\S+)
(?<bytes>\S+)
```

Синтаксис зависит от вкуса, общие из них:

- `(?<name>...)`
- `(?'name'...)`
- `(?P<name>...)`

Обратные ссылки:

- `\k<name>`
- `\k{name}`
- `\k'name'`
- `\g{name}`
- `(?P=name)`

В стиле .NET вы можете иметь несколько групп, имеющих одно и то же имя, они будут использовать блоки захвата .

В PCRE вы должны явно включить его с помощью модификатора `(?J)` (`PCRE_DUPNAMES`) или с помощью группы сброса ветвей `(?|)` . Тем не менее, будет доступно только последнее зафиксированное значение.

```
(?J)(?<a>...)(?<a>...)
(?|(?<a>...)|(?<a>...))
```

Прочитайте Группы захвата онлайн: https://riptutorial.com/ru/regex/topic/660/группы-захвата

глава 7: Жадные и ленивые кванторы

параметры

Кванторы	Описание
`?`	Сопоставьте предыдущий символ или подвыражение 0 или 1 раз (предпочтительно 1).
`*`	Сопоставьте предыдущий символ или подвыражение 0 или более раз (как можно больше).
`+`	Сопоставьте предыдущий символ или подвыражение 1 или более раз (как можно больше).
`{n}`	Сопоставьте предыдущий символ или подвыражение ровно *n* раз.
`{min,}`	Сопоставьте предыдущий символ или подвыражение *мин* или более раз (как можно больше).
`{0,max}`	Сопоставьте предыдущий символ или подвыражение *макс* или меньше (максимально приближайтесь к *максимальному*).
`{min,max}`	Сопоставьте предыдущий символ или подвыражение как минимум *мин.* Раз, но не более, чем *максимальное* время (максимально приближенное к *максимуму*).
Леновые кванторы	**Описание**
`??`	Сопоставьте предыдущий символ или подвыражение 0 или 1 раз (предпочтительно 0).
`*?`	Сопоставьте предыдущий символ или подвыражение 0 или более раз (как можно меньше).
`+?`	Сопоставьте предыдущий символ или подвыражение 1 или более раз (как можно меньше).
`{n}?`	Сопоставьте предыдущий символ или подвыражение ровно *n* раз. Нет никакой разницы между жадной и ленивой версией.
`{min,}?`	Сопоставьте предыдущий символ или подвыражение *мин* или более раз (как можно ближе к *минимуму*).

Кванторы	Описание
`{0,max}?`	Сопоставьте предыдущий символ или подвыражение *макс* или меньше (как можно меньше).
`{min,max}?`	Сопоставьте предыдущий символ или подвыражение как минимум *мин.* Раз, но не более, чем *максимальное* время (как можно ближе к *минимуму*).

замечания

Жадность

Жадный квантификатор всегда пытается повторить подзадачу как можно дольше, прежде чем исследовать более короткие совпадения путем обратного отслеживания.

Как правило, жадный шаблон будет соответствовать самой длинной строке.

По умолчанию все кванторы являются жадными.

Лень

Ленивый (также называемый *нежадным* или *неохотно)* квантификатор всегда пытается повторить суб-схему , как *несколько* раз , как это возможно, прежде чем изучать более длинные матчи за счетом расширения.

Как правило, ленивый шаблон будет соответствовать кратчайшей возможной строке.

Сделать кванторы ленивыми, просто добавить `?` к существующему квантору, например `+?` , `{0,5}?` ,

Концепция жадности и лености существует только в двигателях обратного слежения

Понятие жадного / ленивого квантора существует только в механизмах регулярных выражений обратного слежения. В механизмах регулярных выражений без обратного отслеживания или в системах с регулярным выражением, совместимых с POSIX, квантификаторы определяют только верхнюю границу и нижнюю границу повторения, не указав, как найти совпадение - эти механизмы всегда будут соответствовать самой длинной самой левой строке.

Examples

Жадность против лени

Учитывая следующий ввод:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

Мы будем использовать два шаблона: один жадный: `A.*Z` , и один ленивый: `A.*?Z` Эти шаблоны дают следующие совпадения:

- `A.*Z` дает 1 совпадение: `AlazyZgreeedyAlaaazyZ` (примеры: Regex101 , Rubular)
- `A.*?Z` дает 2 совпадения: `AlazyZ` и `AlaaazyZ` (примеры: Regex101 , Rubular)

Сначала сосредоточьтесь на том, что делает `A.*Z` Когда он соответствует первому `A` ,. `.*` , Будучи жадным, затем пытается сопоставить столько же `.` насколько это возможно.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________/
      A.* matched, Z can't match
```

Так как `Z` не соответствует, откат двигателя и `.*` Должны совпадать с меньшим числом `.` :

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________/
      A.* matched, Z can't match
```

Это происходит еще несколько раз, пока это не дойдет до конца:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \___________________/
      A.* matched, Z can now match
```

Теперь `Z` может совпадать, поэтому общий шаблон соответствует:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \___________________/
      A.*Z matched
```

Напротив, неохотное (ленивое) повторение в `A.*?Z` сначала совпадает с несколькими `.` насколько это возможно, а затем принимать больше `.` как необходимо. Это объясняет, почему он находит два совпадения на входе.

Вот визуальное представление того, что соответствовали двум шаблонам:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \______/l      l = lazy
     \_________g_________/       g = greedy
```

Пример, основанный на ответе, полученном полигенными смазочными материалами .

Стандарт POSIX не включает в себя `?` оператора, поэтому многие рексичные двигатели POSIX не имеют ленивого соответствия. Хотя рефакторинг, особенно с «самым большим трюком» , может помочь в некоторых случаях совпадения, единственный способ иметь истинное ленивое соответствие - использовать движок, который его поддерживает.

Границы с несколькими совпадениями

Когда у вас есть вход с четко определенными границами и вы ожидаете более одного совпадения в своей строке, у вас есть два варианта:

- Использование ленивых кванторов;
- Использование отрицательного символьного класса.

Рассмотрим следующее:

У вас простой механизм шаблонов, вы хотите заменить подстроки типа `$[foo]` где `foo` может быть любой строкой. Вы хотите заменить эту подстроку на все, что зависит от части между `[]` .

Вы можете попробовать что-то вроде `\$\[(.*)\]` , А затем использовать первую группу захвата.

Проблема с этим заключается в том, что если у вас есть строка вроде `something $[foo] lalala $[bar] something else` ваш матч будет

```
something $[foo] lalala $[bar] something else
          | \_______CG1_______/|
          \________Match_______/
```

Группа захвата `foo] lalala $[bar` которая может быть или не быть действительной.

У вас есть два решения

1. Использование лености: в этом случае `*` ленивый - это один из способов найти правильные вещи. Таким образом, вы меняете свое выражение на `\$\[(.*?)\]`
2. Используя отрицательный класс символов: `[^\]]` вы меняете свое выражение на `\$\[([^\]]*)\]` .

В обоих решениях результат будет таким же:

```
something $[foo] lalala $[bar] something else
          | \_/|         | \_/|
          \____/         \____/
```

С группой захвата, соответственно, `foo` и `bar` .

Использование отрицательного символьного класса уменьшает проблему обратного отслеживания и может сэкономить ваш процессор много времени, когда дело доходит до больших входов.

Прочитайте Жадные и ленивые кванторы онлайн: https://riptutorial.com/ru/regex/topic/429/жадные-и-ленивые-кванторы

глава 8: Замены с регулярными выражениями

параметры

В соответствии	Описание
$ номер	Заменяет подстроку, соответствующую номеру группы.
$ {Имя}	Заменяет подстроку, соответствующую названию именованной группы .
$$	Исчерпал символ $ $ в строке результата (замены).
$ & *($ 0)*	Заменяет всю согласованную строку.
$ + *($ &)*	Заменяет согласованный текст на последнюю захваченную группу.
$ `	Заменяет весь согласованный текст с каждым несогласованным текстом перед матчем.
$»	Заменяет весь согласованный текст с каждым несогласованным текстом после матча.
$ _	Заменяет весь согласованный текст на всю строку.
Замечания:	*Курсивные* термины означают, что строки неустойчивы (могут варьироваться в зависимости от вашего аромата регулярного выражения).

Examples

Основы замещения

Одним из наиболее распространенных и полезных способов замены текста с помощью регулярного выражения является использование групп захвата .
Или даже группу Named Capture , как ссылку на хранилище или замену данных.

В документах регулярных выражений довольно похожи друг на друга, поэтому может быть важно никогда не замешать **замещения** (т.е. `$1`) с **Backreferences** (т.е. `\1`). Условия замены используются в заменяющем тексте; Backreferences, в чистом выражении Regex.

Несмотря на то, что некоторые языки программирования принимают как для замещений, это не обнадеживает.

Будем говорить, что у нас есть это регулярное выражение: `/hello(\s+)world/i` . Всякий раз, когда ссылается `$number` (в данном случае `$1`), вместо пробелов будут заменены пробелы, сопоставляемые `\s+` .
Тот же результат будет отображаться с помощью регулярного выражения: `/hello(?<spaces>\s+)world/i` . И поскольку у нас есть именованная группа, мы также можем использовать `${spaces}` .

В этом же примере мы также можем использовать `$0` или `$&` (**Примечание:** `$&` вместо этого можно использовать вместо `$+` , что означает получение **LAST** группы захвата в других машинах с регулярными выражениями), в зависимости от используемого вами эффекта regex, чтобы получить весь согласованный текст. (т.е. `$&` должен возвращать `hEllo woRld` для строки: `hEllo woRld of Regex!`)

Взгляните на этот простой пример подстановки, используя адаптированную цитату Джона Леннона, используя синтаксис `$number` и `${name}` :

Пример простой группы захвата:

```
/(Happy)\./g
```

Test String

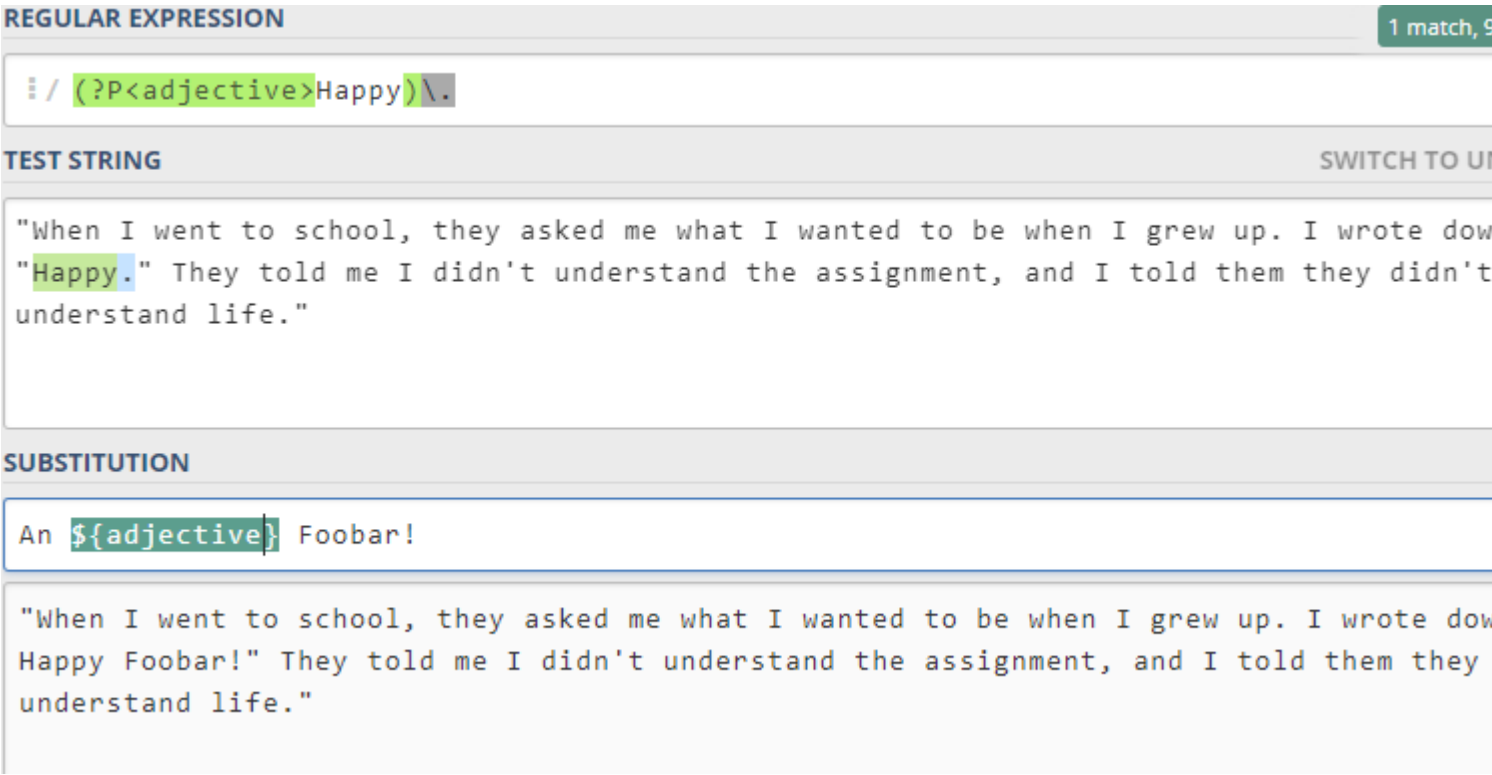
```
"When I went to school, they asked me what I wanted to be when I grew up. I w
me I didn't understand the assignment, and I told them they didn't understand
```

Substitution

```
An $1 Foobar!
```

```
"When I went to school, they asked me what I wanted to be when I grew up. I w
They told me I didn't understand the assignment, and I told them they didn't
```

Именованный пример группы захвата:



Расширенная замена

Некоторые языки программирования имеют свои собственные особенности Regex, например, `$+` term (в C #, Perl, VB и т. Д.), Который заменяет сопоставленный текст на последнюю захваченную группу.

Пример:

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

Пример из сети разработчиков Microsoft Official [1]

Другими редкими условиями замены являются `$`` и `$'` :

`$`` = Заменяет совпадения с текстом **перед** соответствующей строкой
`$'` = Заменяет совпадения с текстом **после** соответствующей строки

В связи с этим эти строки замены должны выполнять свою работу следующим образом:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$`"
Output: "part1part1part3" //Note that part2 was replaced with part1, due &` term
--------------------------------------------------------------------------------
Regex: /part2/
Input: "part1part2part3"
Replacement: "$'"
Output: "part1part3part3" //Note that part2 was replaced with part3, due &' term
```

Ниже приведен пример этих замещений, работающих над частью javascript:

```
var rgx = /middle/;
var text = "Your story must have a beginning, middle, and end"
console.log(text.replace(rgx, "$`"));
//Logs: "Your story must have a beginning, Your story must have a beginning, , and end"
console.log(text.replace(rgx, "$'"))
//Logs: "Your story must have a beginning, , and end, and end"
```

Существует также термин `$_` который вместо этого извлекает весь сопоставленный текст:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$_"
Output: "part1part1part2part3part3" //Note that part2 was replaced with part1part2part3,
                                                                    // due $_ term
```

Преобразование этого в VB даст нам следующее:

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

Пример из сети разработчиков Microsoft Official [2]

И последнее, но не менее важное выражение - `$$` , которое переводится в выражение регулярных выражений, будет таким же, как `\$` (escape-версия буквального **$**).

Если вы хотите сопоставить строку вроде этого: `USD: $3.99` например, и хотите сохранить

`3.99` , но замените ее на `$3.99` только одним регулярным выражением, вы можете использовать:

```
Regex: /USD:\s+\$([\d.]+)/
Input: "USD: $3.99"
Replacement: "$$$1"
To Store: "$1"
Output: "$3.99"
Stored: "3.99"
```

Если вы хотите проверить это с помощью Javascript, вы можете использовать код:

```
var rgx = /USD:\s+\$([\d.]+)/;
var text = "USD: $3.99";
var stored = parseFloat(rgx.exec(text)[1]);
console.log(stored); //Logs 3.99
console.log(text.replace(rgx, "$$$1")); //Logs $3.99
```

Рекомендации

[1]: Подстановка последней захваченной группы
[2]: Подстановка всей входной строки

Прочитайте Замены с регулярными выражениями онлайн:
https://riptutorial.com/ru/regex/topic/9852/замены-с-регулярными-выражениями

глава 9: Именованные группы захвата

Синтаксис

- Создайте названную группу захвата (`x` - шаблон, который вы хотите захватить):

  (? 'name'X) (? X) (? PX)

- Ссылка на названную группу захвата:

  $ {name} \ {name} g \ {name}

замечания

Python и Java не позволяют нескольким группам использовать одно и то же имя.

Examples

Как называется названная группа захвата

Учитывая вкусы, названная группа захвата может выглядеть так:

```
(?'name'X)
(?<name>X)
(?P<name>X)
```

Когда `x` - это шаблон, который вы хотите захватить. Рассмотрим следующую строку:

> Когда-то была *довольно маленькая девочка* ...
>
> Когда-то был *единорог со шляпой* ...
>
> Когда-то была *лодка с пиратским флагом* ...

В котором я хочу захватить предмет (*курсивом)* каждой строки. Я буду использовать следующее выражение `.* was a (?<subject>[\w ]+)[.]{3} .`

Результат совпадения будет иметь место:

```
MATCH 1
subject    [29-47]    `pretty little girl`
MATCH 2
subject    [80-99]    `unicorn with an hat`
MATCH 3
subject    [132-155]    `boat with a pirate flag`
```

Ссылка на указанную группу захвата

Как вы можете (или не знаете), вы можете ссылаться на группу захвата с помощью:

```
$1
```

`1` - номер группы.

Точно так же вы можете ссылаться на названную группу захвата с помощью:

```
${name}
\{name}
g\{name}
```

Давайте рассмотрим предыдущий пример и заменим совпадения на

```
The hero of the story is a ${subject}.
```

В результате получим:

```
The hero of the story is a pretty little girl.
The hero of the story is a unicorn with an hat.
The hero of the story is a boat with a pirate flag.
```

Прочитайте Именованные группы захвата онлайн: https://riptutorial.com/ru/regex/topic/744/именованные-группы-захвата

глава 10: Классы символов

замечания

Простые классы

Regex	Матчи
`[abc]`	Любой из следующих символов: `a` , `b` или `c`
`[az]`	Любой символ от `a` до `z` , **включительно** (это называется *диапазоном*)
`[0-9]`	Любая цифра от `0` до `9` **включительно**

Общие классы

Некоторые группы / диапазоны символов так часто используются, у них есть специальные сокращения:

Regex	Матчи
`\w`	Буквенно-цифровые символы плюс символ подчеркивания (также называемый «символами слова»)
`\W`	Несловные символы (такие же, как `[^\w]`)
`\d`	Цифры (*более широкие,* чем `[0-9]` включают в себя персидские цифры, индийские и т. Д.)
`\D`	Без цифр (*короче* `[^0-9]` так как отбросить персидские цифры, индийские и т. Д.)
`\s`	Простые символы (пробелы, вкладки и т. Д.) **Примечание** : может варьироваться в зависимости от вашего движка / контекста
`\S`	Небелые символы

Отрицательные классы

Каретка (^) после открытия квадратной скобки работает как отрицание символов, которые следуют за ней. Это будет соответствовать всем символам, которые не относятся к классу символов.

Отрицательные классы символов также соответствуют символам прерывания строки, поэтому, если они не должны быть сопоставлены, в класс (\ r и / или \ n) следует добавить специальные символы разрыва строки.

Regex	Матчи
`[^AB]`	Любой символ, **отличный** от `A` и `B`
`[^\d]`	Любой символ, **кроме** цифр

Examples

Основы

Предположим, у нас есть список команд, названных так: `Team A` , `Team B` , ..., `Team Z` Затем:

- `Team [AB]` : Это будет соответствовать либо `Team A` либо `Team B`
- `Team [^AB]` : это будет соответствовать любой команде, **кроме** `Team A` или `Team B`

Нам часто приходится сопоставлять символы, которые «принадлежат» вместе в каком-то контексте или другом (например, буквы от `A` до `Z`), и для этого предназначены классы символов.

Совпадение разных, похожих слов

Рассмотрим класс символов `[aeiou]` . Этот класс символов можно использовать в регулярном выражении, чтобы соответствовать набору слов с аналогичной точностью.

`b[aeiou]t` соответствует:

- летучая мышь
- ставка
- немного
- бот
- но

Он не соответствует:

- бой
- БТТ
- Б.Т.

Классы символов по своему усмотрению соответствуют одному и только одному персонажу за раз.

Совпадение без алфавитно-цифровых символов (класс отрицательных символов)

```
[^0-9a-zA-Z]
```

Это будет соответствовать всем символам, которые не являются ни числами, ни буквами (буквенно-цифровые символы). Если символ подчеркивания _ также должен быть отменен, выражение можно сократить до:

```
[^\w]
```

Или же:

```
\W
```

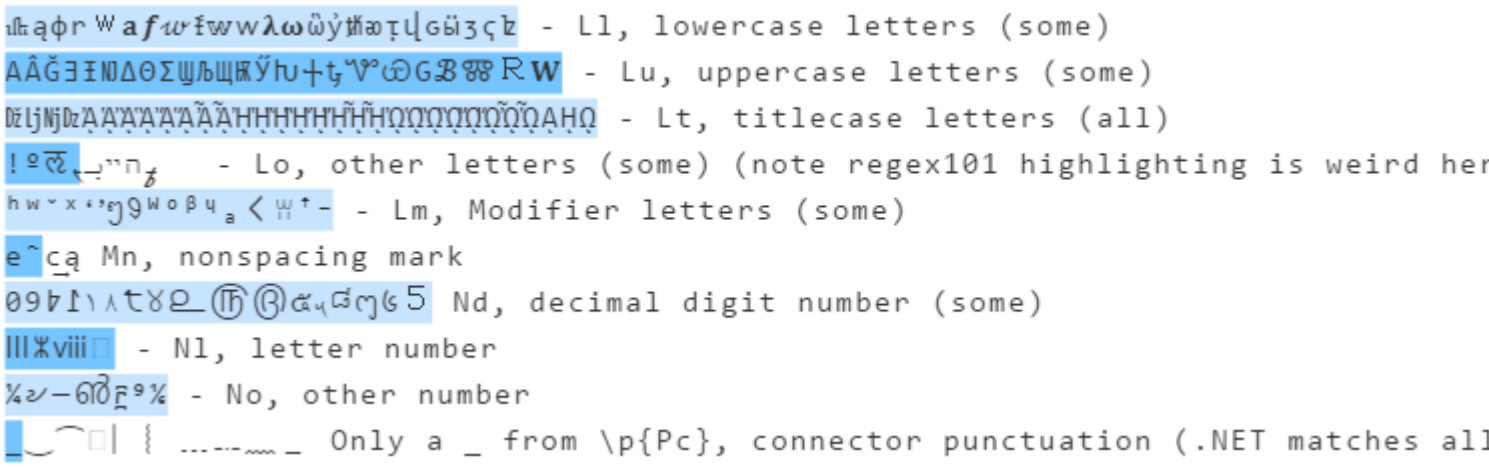
В следующих предложениях:

1. Привет как дела?
2. Я не могу дождаться 2017 года!

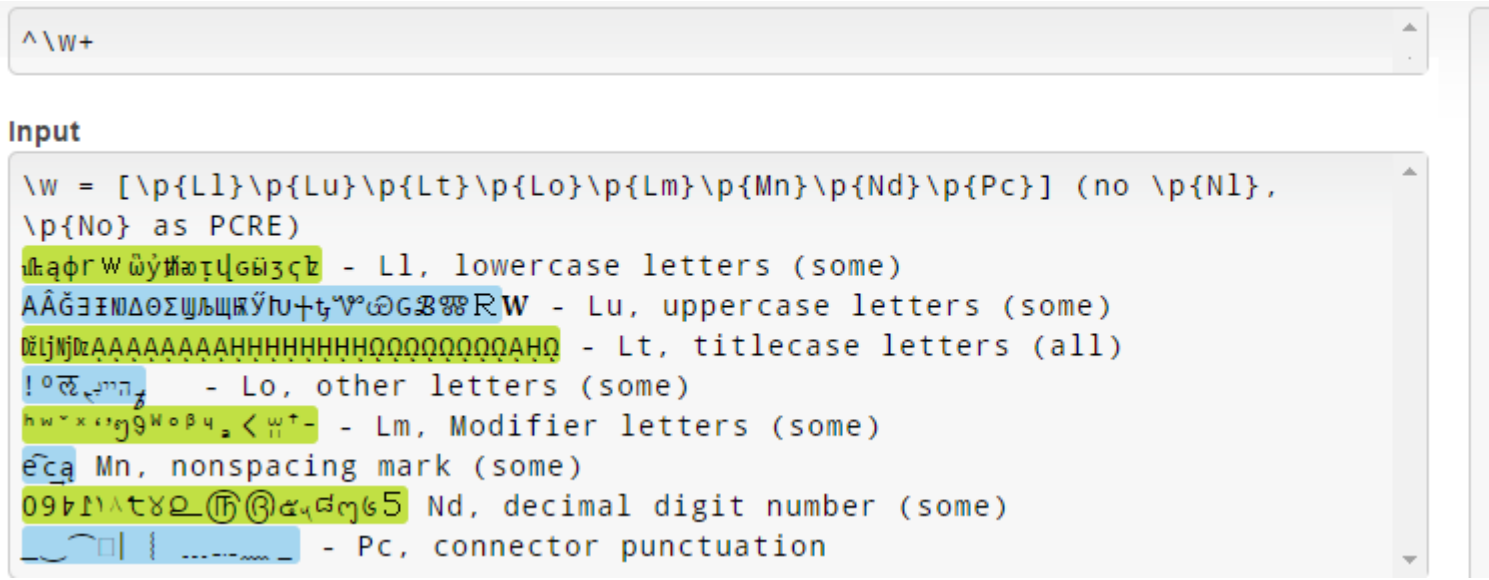
Следующие символы соответствуют:

1. , , , ' , ? и символ конца строки.
2. ' , , ! и символ конца строки.

ПРИМЕЧАНИЕ UNICODE
Обратите внимание, что некоторые варианты с поддержкой свойств символов Unicode могут интерпретировать `\w` и `\W` как `[\p{L}\p{N}_]` и `[^\p{L}\p{N}_]` что означает, что другие буквы Unicode и числовые символы также будут включены (см. документы PCRE). Вот тест PCRE `\w` :

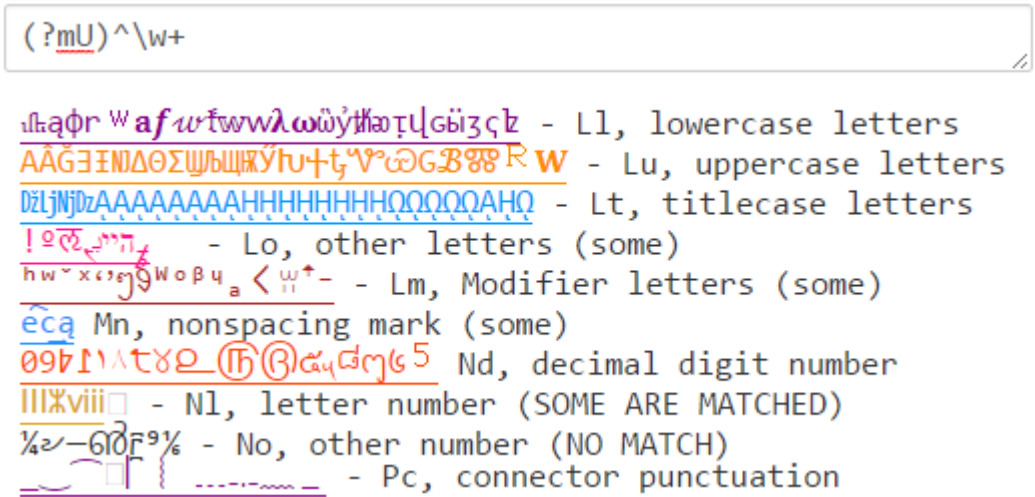


В .NET, `\w = [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]` , и обратите внимание, что он не соответствует `\p{Nl}` и `\p{No}` отличие от PCRE (см. документацию `\w` .NET):



Обратите внимание, что по какой-либо причине строчные буквы Unicode 3.1 (например, 𐐨𐐩𐐪𐐫 𐓘𐓙𐓚𐓛) не сопоставляются.

Java `(?U)\w` будет соответствовать сочетанию того, что `\w` соответствует в PCRE и .NET:



Совпадение без цифр (класс отрицательных символов)

```
[^0-9]
```

Это будет соответствовать всем символам, которые не являются цифрами ASCII.

Если цифры Unicode также должны быть отменены, в зависимости от ваших настроек вкуса / языка можно использовать следующее выражение:

```
[^\d]
```

Это можно сократить до:

```
\D
```

Возможно, вам потребуется включить поддержку свойств символов Unicode явно, используя модификатор `u` или программно на некоторых языках, но это может быть неочевидным. Чтобы явно передать намерение, можно использовать следующую конструкцию (когда доступна поддержка):

```
\P{N}
```

Это *по определению* означает: любой символ, который не является числовым символом в любом скрипте. В отрицательном диапазоне символов вы можете использовать:

```
[^\p{N}]
```

В следующих предложениях:

1. Привет как дела?
2. Я не могу дождаться 2017 года!

Будут сопоставлены следующие символы:

1. `,` , `'` , `?` , символ конца строки и все буквы (строчные и прописные).
2. `'` , `!` , символ конца строки и все буквы (строчные и прописные).

Характерный класс и общие проблемы, с которыми сталкивается новичок

1. Класс символов

Класс символов обозначается `[]` . Содержимое внутри символьного класса рассматривается как `single character separately` . например, предположим, что мы используем

```
[12345]
```

В приведенном выше примере это означает соответствие `1 or 2 or 3 or 4 or 5` . Говоря простыми словами, это можно понять как `or condition for single characters` (**стресс на одном персонаже**)

1.1 Осторожно!

- В классе символов нет понятия соответствия строки. Итак, если вы используете regex `[cat]` , это не означает, что он должен соответствовать слову `cat` буквально, но это означает, что он должен соответствовать либо `c` либо `a` или `t` . Это очень распространенное недоразумение, существующее среди людей, которые новичок в регулярном выражении.
- Иногда люди используют `|` (чередование) внутри класса персонажа, думая, что он будет действовать как `OR condition` которое является неправильным. например, используя `[a|b]` фактически означает совпадение `a` или `|` (буквально) или `b` .

2. Диапазон в символьном классе

Диапазон в символьном классе обозначается знаком `-` . Предположим, мы хотим найти любого символа в английских алфавитах от `A` до `Z` Это можно сделать, используя следующий класс символов

```
[A-Z]
```

Это можно сделать для любого допустимого диапазона ASCII или Unicode. Наиболее часто используемые диапазоны включают `[AZ]` , `[az]` или `[0-9]` . Более того, эти диапазоны можно комбинировать в классе символов, как

```
[A-Za-z0-9]
```

Это означает, что соответствует любому символу в диапазоне от `A to Z` или от `a to z` или от `0 to 9` . Заказ может быть любым. Таким образом, приведенное выше эквивалентно `[a-zA-Z0-9]` если диапазон, который вы определяете, является правильным.

2.1. Осторожность

- Иногда при написании диапазонов от `A` до `Z` люди записывают его как `[Az]` . В большинстве случаев это неправильно, потому что мы используем `z` вместо `Z` Таким образом, это означает, что любой символ из ASCII диапазона `65` (от A) до `122` (из z), который включает много непреднамеренных символов после диапазона ASCII `90` (из Z). **ОДНАКО** , `[Az]` может использоваться для сопоставления всех букв `[a-zA-Z]` в регулярном выражении в стиле POSIX, если для определенного языка задано сопоставление. `[[ "ABCEDEF[]_abcdef" =~ ([Az]+) ]] && echo "${BASH_REMATCH[1]}"` на Cygwin с `LC_COLLATE="en_US.UTF-8"` дает `ABCEDF` . Если вы установите `LC_COLLATE` на `C` (на

Cygwin, сделанный с `export`), он даст ожидаемый `ABCEDEF[]_abcdef` .

- Значение `-` внутри класса персонажа является особенным. Он обозначает диапазон, как описано выше. *Что, если мы хотим совместить `-` символ буквально?* Мы не можем его поместить, иначе он будет обозначать диапазоны, если он помещается между двумя символами. В этом случае мы должны поставить `-` в запуске класса символов , как `[-AZ]` или в конце класса символов , как `[AZ-]` или `escape it` , если вы хотите использовать его в середине , как `[AZ\-az]` .

3. Отрицательный класс символов

Отрицательный класс символов обозначается `[^..]` . Значок каретки `^` обозначает совпадение любого символа, кроме символа, присутствующего в классе символов. например

```
[^cat]
```

означает соответствие любому символу, кроме `c` или `a` или `t` .

3.1 Осторожность

- Значение знака каретки `^` отображает отрицание только в том случае, если оно находится в начале класса символов. Если его где-нибудь еще в классе символов, он трактуется как буквальный символ каретки без какого-либо особого значения.
- Некоторые люди пишут регулярное выражение, например `[^]` . В большинстве движков регулярных выражений это дает ошибку. Причина, заключающаяся в том, что вы используете `^` в исходной позиции, он ожидает, что по крайней мере один символ должен быть отменен. В *JavaScript,* однако, это допустимая конструкция, соответствующая *чему угодно, но ничего* , т.е. соответствует любому возможному символу (но диакритики, по крайней мере, в ES5).

Классы символов POSIX

Классы символов POSIX представляют собой предопределенные последовательности для определенного набора символов.

Класс символов	Описание
`[:alpha:]`	Алфавитные символы
`[:alnum:]`	Алфавитные символы и цифры
`[:digit:]`	Digits
`[:xdigit:]`	Шестнадцатеричные цифры

Класс символов	Описание
`[:blank:]`	Пространство и вкладка
`[:cntrl:]`	Управляющие символы
`[:graph:]`	Видимые символы (все, кроме пробелов и управляющих символов)
`[:print:]`	Видимые символы и пробелы
`[:lower:]`	Строчные буквы
`[:upper:]`	Заглавные буквы
`[:punct:]`	Знаки пунктуации и символы
`[:space:]`	Все пробельные символы, включая разрывы строк

Дополнительные классы символов, которые будут доступны в зависимости от реализации и / или языка.

Класс символов	Описание
`[:<:]`	Начало слова
`[:>:]`	Конец слова
`[:ascii:]`	Персонажи ASCII
`[:word:]`	Буквы, цифры и символ подчеркивания. Эквивалентно `\w`

Чтобы использовать внутреннюю последовательность скобок (класс символов), вы также должны включить квадратные скобки. Пример:

```
[[:alpha:]]
```

Это будет соответствовать одному буквенному символу.

```
[[:digit:]-]{2}
```

Это будет соответствовать 2 символам, которые являются либо цифрами, либо `-` .
Следующее будет соответствовать:

- `--`
- `11`
- `-2`
- `3-`

Дополнительная информация доступна на: Regular-expressions.info

Прочитайте Классы символов онлайн: https://riptutorial.com/ru/regex/topic/1757/классы-символов

глава 11: Когда вы НЕ должны использовать регулярные выражения

замечания

Поскольку регулярные выражения ограничены либо регулярной грамматикой, либо контекстно-свободной грамматикой, существует множество распространенных злоупотреблений регулярными выражениями. Поэтому в этой теме есть несколько примеров того, когда вы *НЕ* должны использовать регулярные выражения, но вместо этого используйте свой любимый язык.

> *Некоторые люди, столкнувшись с проблемой, думают:*
> *«Я знаю, я буду использовать регулярные выражения».*
> *Теперь у них есть две проблемы.*
> - Джейми Завински

Examples

Соответствующие пары (например, скобки, скобки ...)

Некоторые механизмы регулярных выражений (например, .NET) могут обрабатывать контекстно-свободные выражения и будут работать. Но это не относится к большинству стандартных двигателей. И даже если они это сделают, у вас будет сложное трудно читаемое выражение, тогда как использование библиотеки синтаксического анализа может облегчить задачу.

- Как найти все возможные регулярные выражения в python?

Простые строковые операции

Поскольку *регулярные выражения* могут делать много, у вас есть соблазн использовать их для самых простых операций. Но использование механизма regex имеет затраты на использование памяти и процессора: вам нужно скомпилировать выражение, сохранить автомат в памяти, инициализировать его и затем передать его с помощью строки для ее запуска.

И есть много случаев, когда просто не нужно его использовать! Независимо от вашего выбора языка, он всегда имеет базовые инструменты для обработки строк. Итак, как правило, когда есть инструмент для выполнения действия в стандартной библиотеке, используйте этот инструмент, а не регулярное выражение:

- Разделить строку?

Например, следующий сниппт работает в Python, Ruby и Javascript:

```
'foo.bar'.split('.')
```

Что легче читать и понимать, а также намного эффективнее, чем (как-то) эквивалентное регулярное выражение:

```
(\w+)\.(\w+)
```

- Проложить пробелы?

То же самое относится к конечным пространствам!

```
'foobar     '.strip() # python or ruby
'foobar     '.trim() // javascript
```

Что было бы эквивалентно следующему выражению:

```
([^\n]*)\s*$ # keeping \1 in the substitution
```

Анализ HTML (или XML, или JSON, или C-кода, или ...)

Если вы хотите извлечь что-то с веб-страницы (или любого языка представления / программирования), регулярное выражение является неправильным инструментом для задачи. Вместо этого вы должны использовать библиотеки своего языка для достижения этой цели.

Если вы хотите читать HTML, или XML или JSON, просто используйте библиотеку, которая разбирает ее правильно и служит ей как полезные объекты на вашем любимом языке! В итоге вы получите читаемый и более удобный код, и вы не станете

- RegEx соответствуют открытым тегам, за исключением тегов XHTML
- Анализ Python HTML с использованием регулярных выражений
- существует ли регулярное выражение для генерации всех целых чисел для определенного языка программирования

Прочитайте Когда вы НЕ должны использовать регулярные выражения онлайн: https://riptutorial.com/ru/regex/topic/4527/когда-вы-не-должны-использовать-регулярные-выражения

глава 12: Ловушки Regex

Examples

Почему точка (.) Не соответствует символу новой строки ("\ n")?

`.*` в регулярном выражении в основном означает «поймать **все** до конца ввода».

Итак, для простых строк, таких как `hello world` , `.*` работает отлично. Но если у вас есть строка, представляющая, например, строки в файле, эти строки будут разделены *разделителем строк* , например `\n` (новая *строка*) в Unix-подобных системах и `\r\n` (возврат каретки и новая строка) на Окна.

По умолчанию в большинстве движков регулярных выражений `.` **не** соответствует символам новой строки, поэтому совпадение останавливается в конце каждой *логической строки* . Если хочешь `.` чтобы соответствовать **действительно** всему, включая новые строки, вам нужно включить режим «dot- `re.DOTALL` -all» в выбранном вами двигателе регулярных выражений (например, добавить флаг `re.DOTALL` в Python или `/s` в PCRE.

Почему регулярное выражение пропускает некоторые закрывающие скобки / круглые скобки и сопоставляет их потом?

Рассмотрим этот пример:

> Он вошел в кафе «Достоевский» и сказал: «Добрый вечер».

Здесь мы имеем два набора котировок. Предположим, мы хотим сопоставить оба, так что наше регулярное выражение совпадает с `"Dostoevski"` **и** `"Good evening."`

Сначала у вас может возникнуть соблазн сохранить его просто:

```
".*"  # matches a quote, then any characters until the next quote
```

Но это не работает: оно соответствует первой цитате в `"Dostoevski"` и **до** заключительной цитаты в `"Good evening."` , включая `and said:` часть. Демо-версия Regex101

Почему так случилось?

Это происходит из-за механизма регулярных выражений, когда он встречается `.*` , « Съедает» весь вход до самого конца. Затем он должен соответствовать финалу `"` . Таким образом, он« отступает »от конца матча, отбрасывая согласованный текст до тех пор, пока первый `"` будет найден - и это, конечно, последний `"` в матче , в конце `"Good evening."` .

Как предотвратить это и точно соответствовать первым котировкам?

Используйте `[^"]*` . Он не ест все входные данные - только до первого `"` , как это необходимо. Демо-версия Regex101

Прочитайте Ловушки Regex онлайн: https://riptutorial.com/ru/regex/topic/10747/ловушки-regex

глава 13: Модификаторы регулярных выражений (флаги)

Вступление

Шаблоны регулярных выражений часто используются с *модификаторами* (также называемыми *флагами*), которые переопределяют поведение регулярных выражений. Модификаторы *регулярных* выражений могут быть *регулярными* (например, `/abc/i`) и *встроенными* (или *встроенными*) (например, `(?i)abc`). Наиболее распространенными модификаторами являются глобальные, нечувствительные к регистру, многострочные и точечные модификаторы. Тем не менее, ароматы регулярных выражений отличаются количеством поддерживаемых модификаторов regex и их типов.

замечания

Модификаторы PCRE

Модификатор	В соответствии	Описание
PCRE_CASELESS	(?я)	Нечувствительность к регистру
PCRE_MULTILINE	(? M)	Совпадение нескольких строк
PCRE_DOTALL	(? S)	. соответствует новым строкам
PCRE_ANCHORED	(? A)	Мета-символ ^ соответствует только в начале
PCRE_EXTENDED	(?Икс)	Белые пробелы игнорируются
PCRE_DOLLAR_ENDONLY	н /	Мета-символ $ соответствует только в конце
PCRE_EXTRA	(?ИКС)	Строгий разбор партирования
PCRE_UTF8		Обрабатывает символы `UTF-8`
PCRE_UTF16		Обрабатывает символы `UTF-16`
PCRE_UTF32		Обрабатывает символы `UTF-32`

Модификатор	В соответствии	Описание
PCRE_UNGREEDY	(? U)	Устанавливает движок на ленивое совпадение
PCRE_NO_AUTO_CAPTURE	(? :)	Отключает группы автозахвата

Модификаторы Java

Модификатор (`Pattern.###`)	Значение	Описание
UNIX_LINES	1	Включает режим линий Unix .
БЕЗ УЧЕТА РЕГИСТРА	2	Включает совпадение без учета регистра.
КОММЕНТАРИИ	4	Разрешает пробелы и комментарии в шаблоне.
MULTILINE	8	Включает многострочный режим.
LITERAL	16	Включает литеральный синтаксический анализ шаблона.
DOTALL	32	Включает режим dotall.
UNICODE_CASE	64	Включает свертывание флагов Unicode.
CANON_EQ	128	Включает каноническую эквивалентность.
UNICODE_CHARACTER_CLASS	256	Включает Unicode версию предопределенных классов символов и классов символов POSIX.

Examples

Модификатор DOTALL

Шаблон регулярного выражения, в котором модификатор DOTALL (в большинстве выражений regex, выраженный с помощью `s`), изменяет поведение . позволяя ему соответствовать символу новой строки (LF):

```
/cat (.*?) dog/s
```

Это регулярное выражение в стиле Perl будет соответствовать строке, такой как `"cat fled from\na dog"` захватив `"fled from\na"` в группу 1.

Встроенная версия: `(?s)` (например, `(?s)cat (.*?) dog`)

Примечание . В Ruby эквивалент модификатора DOTALL - `m` , модификатор `Regexp::MULTILINE` (например, `/a.*b/m`).

Примечание . JavaScript не предоставляет модификатор DOTALL, поэтому a `.` никогда не может быть разрешено соответствовать символу новой строки. Для достижения такого же эффекта необходимо обходное решение, например, заменяя все `.` s с классом символов catch-all, подобным `[\S\s]` , или классом символов « *ничего* » `[^]` (однако эта конструкция будет рассматриваться как ошибка всеми другими двигателями и, следовательно, не переносима).

Модификатор MULTILINE

Другим примером является модификатор MULTILINE (обычно выражается флагом `m` (не в Oniguruma (например, Ruby), который использует `m` для обозначения модификатора DOTALL)), который делает привязки `^` и `$` совпадающими с началом и концом *строки* , а не с началом / концом всей строки.

```
/^My Line \d+$/gm
```

найдут все *строки,* которые начинаются с `My Line` , а затем содержат пробел и 1 + цифры до конца строки.

Встроенная версия: `(?m)` (например, `(?m)^My Line \d+$`)

ПРИМЕЧАНИЕ . В Oniguruma (например, в Ruby), а также практически в любых текстовых редакторах, поддерживающих регулярные выражения, якоря `^` и `$` обозначают позиции начала и окончания *строки по умолчанию* . Вам нужно использовать `\A` для определения всего начала документа / строки и `\z` для обозначения конца документа / строки. Разница между `\Z` и `\z` заключается в том, что первая может соответствовать перед последним символом новой строки (LF) в конце строки (например, `/\Astring\Z/` найдет совпадение в `"string\n"`) (за исключением Python, где поведение `\Z` равно `\z` и `\z` якорь не поддерживается).

Модификатор IGNORE CASE

Общим модификатором для игнорирования является `i` :

```
/fog/i
```

будет соответствовать `Fog` , `foG` и т. д.

Встроенная версия модификатора выглядит как `(?i)` .

Заметки:

В Java по умолчанию *нечувствительность к регистру предполагает, что только символы в кодировке US-ASCII сопоставляются. `CASE_INSENSITIVE` регистра Unicode может быть активирована путем указания флага `UNICODE_CASE` в сочетании с этим (`CASE_INSENSITIVE`)* . (например, `Pattern p = Pattern.compile("YOUR_REGEX", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);`). Некоторые из них можно найти в *случае нечувствительности к регистру в Java RegEx* . Кроме того, `UNICODE_CHARACTER_CLASS` может использоваться для обеспечения соответствия Unicode.

VERBOSE / COMMENT / IgnorePatternWhitespace-модификатор

Модификатор, который позволяет использовать пробелы внутри некоторых частей шаблона для его форматирования для лучшей читаемости и для комментариев, начинающихся с `#` :

```
/(?x)^           # start of string
  (?=\D*\d)      # the string should contain at least 1 digit
  (?!\d+$)       # the string cannot consist of digits only
  \#             # the string starts with a hash symbol
  [a-zA-Z0-9]+ # the string should have 1 or more alphanumeric symbols
  $              # end of string
/
```

Пример строки: `#word1here` . Обратите внимание, что символ `#` экранирован, чтобы обозначить литерал `#` который является частью шаблона.

Невыбежденное пустое пространство в шаблоне регулярных выражений игнорируется, избегайте его, чтобы сделать его частью шаблона.

Обычно пробелы внутри классов символов (`[...]`) рассматриваются как буквальные пробелы, за исключением Java.

Кроме того, стоит упомянуть, что в PCRE, .NET, Python, Ruby Oniguruma, ICU, Boost regex flavors можно использовать `(?#:...)` комментарии внутри шаблона регулярного выражения.

Явный модификатор Capture

Это специальный модификатор .NET regex, выраженный с помощью `n` . При использовании неназванные группы (например, `(\d+)`) не записываются. Только допустимые записи являются явно названными группами (например, `(?<name> subexpression)`).

```
(?n)(\d+)-(\w+)-(?<id>\w+)
```

будет соответствовать целому `123-1_abc-00098` , но `(\d+)` и `(\w+)` не будут создавать группы в результирующем объекте совпадения. Единственная группа будет `${id}` . См. Демонстрацию .

Модификатор UNICODE

Модификатор UNICODE, обычно выражаемый как `u` (PHP, Python) или `U` (Java), заставляет движок regex обрабатывать шаблон и строку ввода как строки и шаблоны Unicode, создавать классы сокращенного типа, такие как `\w` , `\d` , `\s` и т. д. Unicode-aware.

```
/\A\p{L}+\z/u
```

является регулярным выражением PHP для соответствия строкам, состоящим из 1 или более букв Unicode. См. Демо-версию regex .

Обратите внимание, что в **PHP** модификатор `/u` позволяет механизму PCRE обрабатывать строки как строки UTF8 (путем включения глагола `PCRE_UTF8`) и создавать классы сокращенного символа в шаблоне, поддерживающем Unicode (путем включения глагола `PCRE_UCP` , см. Подробнее на *pcre.org*) ,

> **Строка и предметные строки рассматриваются как UTF-8.** Этот модификатор доступен с PHP 4.1.0 или выше в Unix и с PHP 4.2.3 на win32. UTF-8 справедливость шаблона и объекта проверяется с PHP 4.3.5. Недействительный объект приведет к тому, что функция preg_ * не будет соответствовать ничему; недопустимый шаблон вызовет ошибку уровня E_WARNING. Пять и шесть октетных последовательностей UTF-8 считаются недействительными с PHP 5.3.4 (соответственно PCRE 7.3 2007-08-28); ранее они считались действительными UTF-8.

В Python 2.x `re.UNICODE` влияет только на сам шаблон: *Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` зависимости от базы данных свойств символов Unicode.*

Встроенная версия: `(?u)` в Python, `(?U)` в Java. Например:

```
print(re.findall(ur"(?u)\w+", u"Dąb")) # [u'D\u0105b']
print(re.findall(r"\w+", u"Dąb"))      # [u'D', u'b']

System.out.println("Dąb".matches("(?U)\\w+")); // true
System.out.println("Dąb".matches("\\w+"));     // false
```

Модификатор PCRE_DOLLAR_ENDONLY

PCRE-совместимый модификатор *PCRE_DOLLAR_ENDONLY* , который делает *привязку* `$` anchor в *самом конце строки* (исключая позицию перед окончательной новой строкой в строке).

```
/^\d+$/D
```

равно

```
/^\d+\z/
```

и соответствует целой строке, которая состоит из 1 или более цифр и не будет соответствовать `"123\n"` , но будет соответствовать `"123"` .

Модификатор PCRE_ANCHORED

Другой модификатор, совместимый с PCRE, выраженный модификатором `/A` Если этот модификатор установлен, шаблон вынужден быть «привязан», то есть он должен соответствовать только в начале искомой строки («строка субъекта»). Этот эффект также может быть достигнут с помощью соответствующих конструкций в самом шаблоне, что является единственным способом сделать это в Perl.

```
/man/A
```

такой же как

```
/^man/
```

Модификатор PCRE_UNGREEDY

Соответствующий PCRE флаг PCRE_UNGREEDY, выраженный с помощью `/U` Он переключает жадность внутри шаблона: `/a.*?b/U` = `/a.*b/` и наоборот.

Модификатор PCRE_INFO_JCHANGED

Еще один модификатор PCRE, который позволяет использовать дубликаты названных групп.

ПРИМЕЧАНИЕ . Поддерживается только *встроенная* версия - `(?J)` и должна быть размещена в начале шаблона.

Если вы используете

```
/(?J)\w+-(?:new-(?<val>\w+)|\d+-empty-(?<val>[^-]+)-collection)/
```

значения группы «val» никогда не будут пустыми (всегда будут установлены). Аналогичный эффект может быть достигнут при сбросе ветвей.

Модификатор PCRE_EXTRA

Модификатор PCRE, вызывающий ошибку, если за обратным слэшем в шаблоне следует письмо, которое не имеет особого значения. По умолчанию обратная косая черта, сопровождаемая буквой без специального значения, рассматривается как литерал.

Например

```
/big\y/
```

будет соответствовать `bigy` , но

```
/big\y/X
```

будет выдавать исключение.

Встроенная версия: `(?X)`

Прочитайте Модификаторы регулярных выражений (флаги) онлайн: https://riptutorial.com/ru/regex/topic/5138/модификаторы-регулярных-выражений--флаги-

глава 14: Обратная ссылка

Examples

ОСНОВЫ

Обратные ссылки используются для сопоставления с тем же текстом, который ранее был сопоставлен группой захвата. Это помогает в повторном использовании предыдущих частей вашего шаблона и в обеспечении соответствия двух частей строки.

Например, если вы пытаетесь проверить, что строка имеет цифру от нуля до девяти, разделитель, такой как дефисы, косые черты или даже пробелы, строчная буква, другой разделитель, затем другая цифра от нуля до девяти, вы можете используйте регулярное выражение:

```
[0-9][-/ ][a-z][-/ ][0-9]
```

Это будет соответствовать `1-a-4` , но оно *также* будет соответствовать `1-a/4` или `1 a-4` . Если мы хотим, чтобы разделители соответствовали друг другу, мы можем использовать группу захвата и обратную ссылку. В обратной ссылке будет отображаться совпадение, найденное в указанной группе захвата, и убедитесь, что местоположение обратной ссылки соответствует точно.

Используя тот же пример, регулярное выражение будет выглядеть следующим образом:

```
[0-9]([-/ ])[a-z]\1[0-9]
```

`\1` обозначает первую группу захвата в шаблоне. При этом небольшом изменении регулярное выражение теперь соответствует `1-a-4` или `1 a 4` но не `1 a-4` или `1-a/4` .

Номер, который будет использоваться для обратной ссылки, зависит от местоположения вашей группы захвата. Число может быть от одного до девяти и может быть найдено путем подсчета ваших групп захвата.

```
([0-9])([-/ ])[a-z][-/ ]([0-9])
|--1--||--2--|           |--3--|
```

Вложенные группы захвата слегка меняют этот показатель. Сначала вы подсчитываете группу внешнего захвата, затем следующий уровень и продолжаете, пока не покинете гнездо:

```
(([0-9])([-/ ]))([a-z])
 |--2--||--3--|
|-------1------||--4--|
```

Неоднозначные обратные ссылки

Проблема: вам нужно сопоставить текст определенного формата, например:

```
1-a-0
6/p/0
4 g 0
```

Это цифра, разделитель (один из `-` , `/` или пробел), буква, тот же разделитель и нуль.

Наивное решение: адаптируя регулярное выражение из примера Basics , вы можете найти это регулярное выражение:

```
[0-9]([-/ ])[a-z]\10
```

Но это, вероятно, не сработает. Большинство разновидностей регулярных выражений поддерживают более девяти групп захвата, и очень немногие из них достаточно умны, чтобы понять, что, поскольку есть только одна группа захвата, `\10` должна быть обратной ссылкой на группу 1, за которой следует буква `0` . Большинство вкусов будут относиться к ней как к обратной стороне группы 10. Некоторые из них будут вызывать исключение, потому что нет группы 10; остальные просто не совпадают.

Существует несколько способов избежать этой проблемы. Один из них - использование названных групп (и названных обратных ссылок):

```
[0-9](?<sep>[-/ ])[a-z]\k<sep>0
```

Если ваш язык регулярного выражения поддерживает его, формат `\g{n}` (где `n` - число) может заключить номер обратной ссылки в фигурные скобки, чтобы отделить его от любых цифр после него:

```
[0-9]([-/ ])[a-z]\g{1}0
```

Другим способом является использование расширенного форматирования регулярных выражений, разделение элементов на несущественные пробелы (в Java вам нужно избежать пробела в скобках):

```
(?x) [0-9] ([-/ ]) [a-z] \1 0
```

Если ваш аромат регулярного выражения не поддерживает эти функции, вы можете добавить ненужный, но безвредный синтаксис, например, группу, не связанную с захватом:

```
[0-9]([-/ ])[a-z](?:\1)0
```

... или фиктивный квантификатор (возможно, это единственное обстоятельство, в котором

`{1}` полезно):

```
[0-9]([-/ ])[a-z]\1{1}0
```

Прочитайте Обратная ссылка онлайн: https://riptutorial.com/ru/regex/topic/4072/обратная-ссылка

глава 15: Откат

Examples

Что вызывает обратное отслеживание?

Чтобы найти совпадение, механизм регулярных выражений будет потреблять символы один за другим. Когда начинается частичное совпадение, двигатель запомнит начальную позицию, чтобы вернуться в случае, если следующие символы не совпадают.

- Если совпадение завершено, то нет возврата
- Если совпадение не завершено, движок будет возвращать строку (например, когда вы перематываете старую ленту), чтобы попытаться найти целое совпадение.

Например: `\d{3}[az]{2}` против строки `abc123def` будет просматриваться как таковой:

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does match \d (third one)
abc123def
      ^ Does match [a-z] (first one)
abc123def
       ^ Does match [a-z] (second one)
           MATCH FOUND
```

Теперь изменим регулярное выражение на `\d{2}[az]{2}` на ту же строку (`abc123def`):

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does not match [a-z]
abc123def
    ^ BACKTRACK to catch \d{2} => (23)
abc123def
      ^ Does match [a-z] (first one)
```

```
abc123def
       ^ Does match [a-z] (second one)
          MATCH FOUND
```

Почему возвращение может быть ловушкой?

Откат может быть вызван дополнительными кванторами или альтернативными конструкциями, поскольку механизм регулярных выражений будет пытаться исследовать каждый путь. Если вы запустите регулярное выражение `a+b` против `aaaaaaaaaaaaa` нет, и двигатель найдет его довольно быстро.

Но если вы измените регулярное выражение на `(aa*)+b` количество комбинаций будет расти довольно быстро, и большинство (не оптимизированных) движков будут пытаться исследовать все пути, и у них будет целая вечность, чтобы попытаться найти совпадение или бросить исключение тайм-аута. Это называется **катастрофическим отступлением** .

Конечно, `(aa*)+b` кажется ошибкой новичка, но здесь здесь, чтобы проиллюстрировать этот момент, и иногда вы столкнетесь с той же проблемой, но с более сложными шаблонами.

Более экстремальный случай катастрофического обратного следования происходит с регулярным выражением `(x+x+)+y` (вы, вероятно, видели его здесь и здесь), которому требуется экспоненциальное время, чтобы выяснить, что строка, содержащая `x` s и ничего (например, `xxxxxxxxxxxxxxxxxxxx`) не соответствуют этому.

Как этого избежать?

Будьте как можно более конкретными, максимально уменьшите возможные пути. Обратите внимание, что некоторые регулярные выражения не уязвимы для обратного отслеживания, например, в `awk` или `grep` поскольку они основаны на Thompson NFA .

Прочитайте Откат онлайн: https://riptutorial.com/ru/regex/topic/977/откат

глава 16: Полезная Regex Витрина

Examples

Совпадение даты

Вы должны помнить, что регулярное выражение предназначено для сопоставления даты (или нет). Утверждение о том, что дата *действительна,* является гораздо более сложной борьбой, поскольку для этого потребуется много обработки исключений (см. Условия високосного года).

Начнем с сопоставления месяца (1 - 12) с необязательным ведущим 0:

```
0?[1-9]|1[0-2]
```

Чтобы сопоставить день, также с необязательным ведущим 0:

```
0?[1-9]|[12][0-9]|3[01]
```

И чтобы соответствовать году (давайте просто предположим, что диапазон 1900 - 2999):

```
(?:19|20)[0-9]{2}
```

Сепаратор может быть пространством, тире, косой чертой, пустым и т. Д. Не стесняйтесь добавлять все, что, по вашему мнению, можно использовать в качестве разделителя:

```
[-\\/ ]?
```

Теперь вы соедините все это и получите:

```
(0?[1-9]|1[0-2])[-\\/ ]?(0?[1-9]|[12][0-9]|3[01])[-/ ]?(?:19|20)[0-9]{2} // MMDDYYYY
(0?[1-9]|[12][0-9]|3[01])[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Если вы хотите быть немного более педантичным, вы можете использовать обратную ссылку, чтобы быть уверенным, что два разделителя будут одинаковыми:

```
(0?[1-9]|1[0-2])([-\\/ ]?)(0?[1-9]|[12][0-9]|3[01])\2(?:19|20)[0-9]{2} // MMDDYYYY
                                                    ^ refer to [-/ ]
(0?[1-9]|[12][0-9]|3[01])([-\\/ ]?)(0?[1-9]|1[0-2])\2(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}([-\\/ ]?)(0?[1-9]|1[0-2])\2(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Сопоставьте адрес электронной почты

> Согласование адреса электронной почты внутри строки является трудной задачей, поскольку спецификация, определяющая ее, RFC2822 , сложна, что трудно реализовать в качестве регулярного выражения. Более подробную информацию о том, почему не рекомендуется сопоставлять электронную почту с регулярным выражением, см. Пример антипаттера, если вы не используете регулярное выражение: для сопоставления писем . Лучшим советом, который следует учитывать на этой странице, является использование экспертной оценки и широкой библиотеки на вашем любимом языке для ее реализации.

Подтвердить формат адреса электронной почты

Когда вам нужно быстро проверить запись, чтобы убедиться, что она *похожа* на электронную почту, лучшим вариантом является ее простота:

```
^\S{1,}@\S{2,}\.\S{2,}$
```

Это регулярное выражение будет проверять, что почтовый адрес представляет собой не пространственную разделяемую последовательность символов длиной больше единицы, за которой следует `@` , за которой следуют две последовательности символов без пробелов длиной две или более, разделенные символом a `.` , Это не идеально, и может проверять недействительные адреса (в соответствии с форматом), но самое главное, это не является недействительным действительные адреса.

Проверить адрес

Единственный надежный способ проверить правильность электронного письма - проверить его существование. Раньше была команда `VRFY` SMTP, которая была разработана для этой цели, но, к сожалению, после злоупотребления спамерами она теперь больше не доступна .

Таким образом, единственный способ, которым вы остались, проверить, что почта действительна и существует, - это фактически отправить электронное письмо на этот адрес.

Огромные альтернативы Regex

Хотя, это не невозможно проверить адрес электронной почты с помощью регулярных выражений. Единственные проблемы в том, что чем ближе к спецификации эти регулярные выражения будут, тем больше они будут и, как следствие, их невозможно прочитать и сохранить. Ниже вы найдете пример такого более точного регулярного выражения, которое используется в некоторых библиотеках.

> Следующее регулярное выражение дается для документации и целей обучения,

скопировать их в свой код - плохая идея. Вместо этого используйте эту библиотеку напрямую, поэтому вы можете полагаться на вышеперечисленных разработчиков кода и разработчиков сверстников, чтобы обновлять и обновлять код анализа электронной почты.

Модуль согласования адресов Perl

Лучшие примеры такого регулярного выражения находятся в некоторых языках стандартных библиотек. Например, есть один из модуля `RFC::RFC822::Address` в библиотеке Perl, который пытается быть максимально точным в соответствии с RFC. Для вашего любопытства вы можете найти версию этого регулярного выражения по этому URL-адресу , которое было создано из грамматики, и если у вас есть соблазн скопировать его, вот цитата из автора регулярного выражения:

> « *Я не поддерживаю регулярное выражение [связанное]. В нем могут быть ошибки, которые уже были исправлены в модуле Perl».*

.Net-адресный модуль соответствия

Другой, более короткий вариант - тот, который используется стандартной библиотекой . Net в модуле `EmailAddressAttribute` :

```
^((([a-z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])+(\.([a-
z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])+)*)|((\x22)((((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(([\x01-\x08\x0b\x0c\x0e-
\x1f\x7f]|\x21|[\x23-\x5b]|[\x5d-\x7e]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(\\([\x01-
\x09\x0b\x0c\x0d-\x7f]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])))*(((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(\x22)))@((([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])|(([a-z]|\d|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-
|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])*([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])))\.)+(([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(([a-z]|[\u00A0-
\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])*([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])))\.?$
```

Но даже если он *короче,* он все еще слишком велик, чтобы быть читаемым и легко ремонтируемым.

Модуль согласования адресов Ruby

В рубине композиция регулярного выражения используется в модуле rfc822 для соответствия адресу. Это аккуратная идея, так как в случае обнаружения ошибок будет легче определить часть регулярного выражения, чтобы изменить и исправить ее.

Модуль согласования адресов Python

В качестве встречного примера модуль синтаксического анализа электронной почты python не использует регулярное выражение, а вместо него реализует его с помощью

синтаксического анализатора.

Сопоставьте номер телефона

Вот как сопоставить префиксный код (а `+` или (00), а затем число от 1 до 1939 с дополнительным пространством):
Это не ищет *допустимый* префикс, а может быть префиксом. Полный список префиксов

```
(?:00|\+)?[0-9]{4}
```

Тогда, поскольку длина всего номера телефона составляет, самое большее, 15, мы можем искать до 14 цифр:
Для префикса тратится не менее 1 разряда

```
[0-9]{1,14}
```

Номера могут содержать пробелы, точки или тире и могут быть сгруппированы по 2 или 3.

```
(?:[ .-][0-9]{3}){1,5}
```

С дополнительным префиксом:

```
(?:(?:00|\+)?[0-9]{4})?(?:[ .-][0-9]{3}){1,5}
```

Если вы хотите сопоставить определенный формат страны, вы можете использовать этот поисковый запрос и добавить страну, вопрос уже задан.

Соответствие IP-адресу

IPv4

Чтобы соответствовать формату адреса IPv4, вам нужно проверить номера `[0-9]{1,3}` три раза `{3}` разделенные точками `\.` и заканчивается другим номером.

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Это регулярное выражение слишком простое - если вы хотите, чтобы он был точным, вам нужно проверить, что числа находятся в диапазоне от `0` до `255` , причем регулярное выражение выше принимает `444` в любой позиции. Вы хотите проверить 250-255 с `25[0-5]` или любым другим значением 200 `2[0-4][0-9]` или любым значением 100 или менее с `[01]?[0-9][0-9]` . Вы хотите проверить, что за ним следует период `\.` три раза `{3}` а затем один раз без периода.

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IPv6

IPv6 - адрес принимает форму 8 16-битовых слов , разделенных шестигранных с двоеточием (`:`) характером. В этом случае мы проверяем 7 слов, за которыми следуют двоеточия, а затем один, который не является. Если слово имеет ведущие нули, они *могут* быть усечены, то есть каждое слово может содержать от 1 до 4 шестнадцатеричных цифр.

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

Этого, однако, недостаточно. Поскольку адреса IPv6 могут стать довольно «многословными», стандарт указывает, что слова с нулевым значением могут быть заменены на `::` . Это может быть сделано только один раз в адресе (где-то между 1 и 7 последовательными словами), поскольку в противном случае это было бы неопределенным. Это приводит к ряду (довольно неприятных) вариаций:

```
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Теперь, соединяя все вместе (используя чередование), получается:

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$|
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Не забудьте записать его в многострочном режиме и с кучей комментариев, чтобы тот, кому неизбежно поставили задачу выяснить, что это значит, не приходит после вас с тупым объектом.

Подтвердить строчку времени 12 часов и 24 часа

Для формата времени в 12 часов можно использовать:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9]\s*[ap]m$
```

куда

- `(?:0?[0-9]|1[0-2])` - это час

- `[-:]` - это разделитель, который можно настроить в соответствии с вашими потребностями
- `[0-5][0-9]` - это минута
- `\s*[ap]m` следует за любым количеством пробельных символов, а `am` или `pm`

Если вам нужны секунды:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9][-:][0-5][0-9]\s*[ap]m$
```

Для 24-часовой формат времени:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9]$
```

Куда:

- `(?:[01][0-9]|2[0-3])` - это час
- `[-:h]` разделитель, который можно настроить в соответствии с вашими потребностями
- `[0-5][0-9]` - это минута

С секундами:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9][-:m][0-5][0-9]$
```

Где `[-:m]` - второй разделитель, заменяющий `h` часами на `m` за минуты, а `[0-5][0-9]` является вторым.

Соответствующий британский почтовый индекс

Regex соответствует почтовым индексам в Великобритании.

Формат выглядит следующим образом, где A означает букву и цифру 9:

Формат	покрытие	пример
клетка	клетка	
AA9A 9AA	Площадь почтового индекса WC; EC1-EC4, NW1W, SE1P, SW1	EC1A 1BB
A9A 9AA	E1W, N1C, N1P	W1A 0AX
A9 9AA, A99 9AA	B, E, G, L, M, N, S, W	M1 1AE, B33 8TH
AA9 9AA, AA99 9AA	Все остальные почтовые индексы	CR2 6XH, DN55 1PT

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-
9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY])))) [0-9][A-Z-[CIKMOV]]{2})
```

Где первая часть:

```
 (GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-
9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY]))))
```

Во-вторых:

```
[0-9][A-Z-[CIKMOV]]{2})
```

Прочитайте Полезная Regex Витрина онлайн: https://riptutorial.com/ru/regex/topic/3605/полезная-regex-витрина

глава 17: Потенциальные квантификаторы

замечания

NB Эмулирующие притяжательные кванторы

Examples

Основное использование потенциальных квантификаторов

Потенциальные кванторы - это еще один класс кванторов во многих вариантах регулярных выражений, которые позволяют эффективно отключать откат для данного токена. Это может помочь улучшить производительность, а также предотвратить совпадения в определенных случаях.

Класс притяжательных кванторов можно отличить от ленивых или жадных кванторов путем добавления a `+` после квантификатора, как показано ниже:

Квантор	жадный	ленивый	Притяжательный падеж
Ноль или больше	`*`	`*?`	`*+`
Один или больше	`+`	`+?`	`++`
Нулевой или один	`?`	`??`	`?+`

Рассмотрим, например, два шаблона `".*"` И `".*+"` , Работающие на строке `"abc"d` . В обоих случаях `"` в начале строки сопоставляется, но после этого два шаблона будут иметь разные поведения и результаты.

Тогда жадный квантификатор вырвет оставшуюся часть строки `abc"d` . Поскольку это не соответствует шаблону, оно будет возвращаться назад и отбрасывать `d` , оставляя **квантификатор,** содержащий `abc"` . Поскольку это все еще не соответствует шаблону, квантификатор будет отбрасывать значение `"` , оставляя его содержащим только `abc` . Это соответствует шаблону (поскольку `"` сопоставляется буквам, а не квантификатором »), а регулярное выражение сообщает об успешности.

Притяжательный квантификатор также разрушит остальную часть строки, но, в отличие от жадного квантификатора, он не отступит. Поскольку его содержимое, `abc"d` , не разрешает остальную часть шаблона совпадения, регулярное выражение останавливается и сообщается о сбое.

Поскольку притяжательные квантификаторы не выполняют обратный поиск, они могут

привести к значительному увеличению производительности по длинным или сложным шаблонам. Однако они могут быть опасны (как показано выше), если кто-то не знает, как точно квантификаторы работают внутри страны.

Прочитайте Потенциальные квантификаторы онлайн:
https://riptutorial.com/ru/regex/topic/5916/потенциальные-квантификаторы

глава 18: Регулярное повторение пароля

Examples

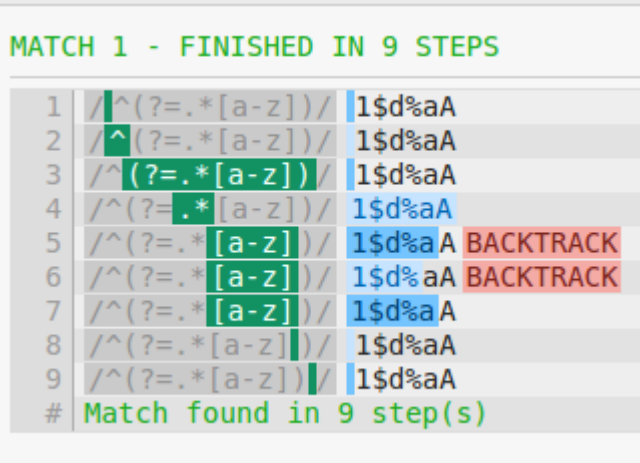
Пароль, содержащий как минимум 1 прописную букву, 1 строчную букву, 1 цифру, 1 специальный символ и длину не менее 10

Поскольку символы / цифры могут быть в любом месте внутри строки, нам нужны образы. Lookaheads имеют `zero width` означает, что они не потребляют ни одной строки. В простых словах положение проверки сбрасывается до исходного положения после выполнения каждого условия просмотра.

Предположение : - *Рассмотрение несловных символов как специальных*

```
^(?=.{10,}$)(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*\W).*$
```

Прежде чем приступать к объяснению, давайте посмотрим, как работает регулярное выражение `^(?=.*[az])` (*длина здесь не рассматривается*) в строке `1$d%aA`



Кредит на изображение : - **https://regex101.com/**

Что нужно заметить

- Проверка начинается с начала строки из-за метки привязки `^` .
- Позиция проверки сбрасывается до начала после выполнения условия просмотра.

Распределение регулярных выражений

```
^ #Starting of string
 (?=.{10,}$) #Check there is at least 10 characters in the string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[a-z]) #Check if there is at least one lowercase in string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[A-Z]) #Check if there is at least one uppercase in string.
```

```
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[0-9]) #Check if there is at least one digit in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*\W) #Check if there is at least one special character in string.
          #As this is lookahead the position of checking will reset to starting again
.*$ #Capture the entire string if all the condition of lookahead is met. This is not required
if only validation is needed
```

Мы также можем использовать *нежелательную* версию вышеупомянутого регулярного выражения

```
^(?=.{10,}$)(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[0-9])(?=.*?\W).*$
```

Пароль, содержащий как минимум 2 прописных, 1 строчный, 2 цифры и длиной не менее 10

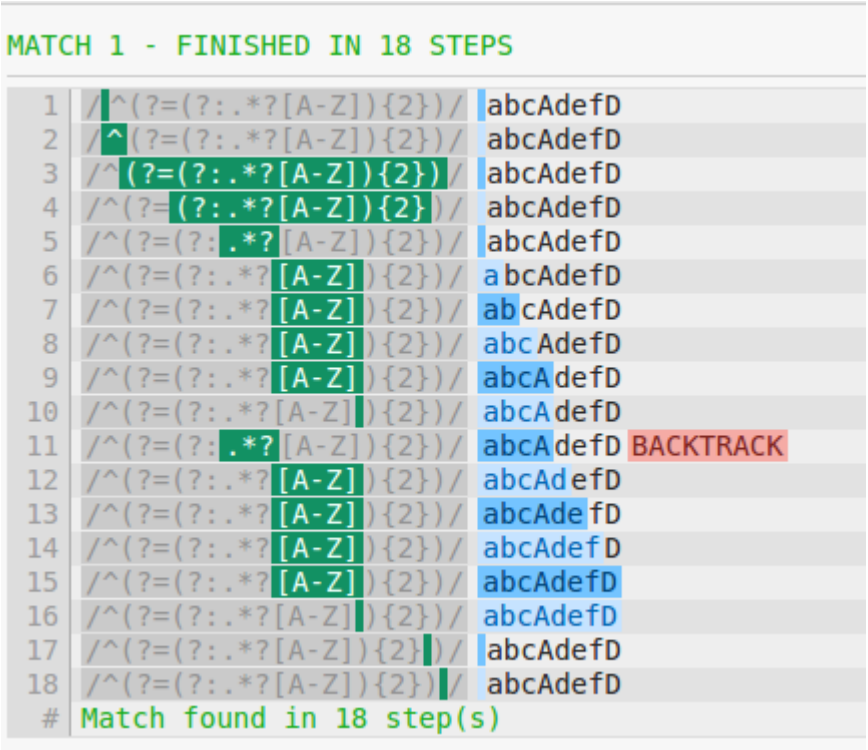
Это может быть сделано с некоторой модификацией в вышеупомянутом регулярном выражении

```
^(?=.{10,}$)(?=(?:.*?[A-Z]){2})(?=.*?[a-z])(?=(?:.*?[0-9]){2}).*$
```

или же

```
^(?=.{10,}$)(?=(?:.*[A-Z]){2})(?=.*[a-z])(?=(?:.*[0-9]){2}).*
```

Посмотрим, как простое регулярное выражение `^(?=(?:.*?[AZ]){2})` работает на строке `abcAdefD`



Кредит на изображение : - **https://regex101.com/**

Прочитайте Регулярное повторение пароля онлайн: https://riptutorial.com/ru/regex/topic/5340/регулярное-повторение-пароля

глава 19: Рекурсия

замечания

Рекурсия в основном доступна в Perl-совместимых ароматах, таких как:

- Perl
- PCRE
- Oniguruma
- Увеличение

Examples

Восстановите весь шаблон

Конструкция `(?R)` эквивалентна `(?0)` (или `\g<0>`) - она позволяет вам повторить весь шаблон:

```
<(?>[^<>]+|(?R))+>
```

Это будет соответствовать правильно сбалансированным угловым скобкам с любым текстом между скобками, например `<a<b>c<d>e>` .

Записаться в подшаблон

Вы можете записаться в подшаблон, используя следующие конструкции (в зависимости от вкуса), предполагая, что `n` - номер группы захвата, и `name` имя группы захвата.

- `(?n)`
- `\g<n>`
- `\g'0'`
- `(?&name)`
- `\g<name>`
- `\g'name'`
- `(?P>name)`

Следующий шаблон:

```
\[(?<angle><(?&angle)*+>)*\]
```

Будет соответствовать текст, например: `[<<><>><>]` - хорошо сбалансированные угловые скобки в квадратных скобках. Рекурсия часто используется для сопоставления сбалансированных конструкций.

Определения подшаблонов

Конструкция `(?(DEFINE) ... )` позволяет вам определять подшаблоны, которые вы можете ссылаться позже через рекурсию. Когда встречается в шаблоне, он *не* будет сопоставляться.

Эта группа должна содержать именованные подшаблонные определения, которые будут доступны только через рекурсию. Вы можете определить грамматики следующим образом:

```
(?x) # ignore pattern whitespace
(?(DEFINE)
  (?<string> ".*?" )
  (?<number> \d+ )
  (?<value>
    \s* (?:
        (?&string)
      | (?&number)
      | (?&list)
    ) \s*
  )
  (?<list> \[ (?&value) (?: , (?&value) )* \] )
)
^(?&value)$
```

Этот шаблон будет проверять текст следующим образом:

```
[42, "abc", ["foo", "bar"], 10]
```

Обратите внимание, как список может содержать одно или несколько значений, а сами значения могут быть списком.

Относительные ссылки групп

На подшаблонах можно ссылаться на их *относительный* номер группы:

- `(?-1)` будет возвращаться в *предыдущую* группу
- `(?+1)` перейдет в *следующую* группу

Также можно использовать синтаксис `\g<N>` .

Обратные ссылки в рекурсии (PCRE)

В PCRE сопоставляемые группы, используемые для обратных ссылок до рекурсии, хранятся в рекурсии. Но после рекурсии они все сбрасываются до того, что они были перед входом в нее. Другими словами, согласованные группы в рекурсии забыты.

Например:

```
(?J)(?(DEFINE)(\g{a}(?<a>b)\g{a}))(?<a>a)\g{a}(?1)\g{a}
```

Матчи

```
aaabba
```

Рекурсии являются атомарными (PCRE)

В PCRE он не отслеживает после того, как найдено первое совпадение для рекурсии. Так

```
(?(DEFINE)(aaa|aa|a))(?1)ab
```

не соответствует

```
aab
```

потому что после того, как он сопоставил `aa` в рекурсии, он никогда не пытается снова соответствовать только `a` .

Прочитайте Рекурсия онлайн: https://riptutorial.com/ru/regex/topic/739/рекурсия

глава 20: Сброс матча: \ K

замечания

Regex101 определяет функциональность \ K как:

> `\K` сбрасывает начальную точку сообщенного матча. Любые ранее употребляемые символы больше не включаются в финальный матч

Управляющая последовательность `\K` поддерживается несколькими двигателями, языками или инструментами, такими как:

- увеличить (с тех пор)
- grep -P ← *использует PCRE*
- Онигурума (с 5.13.3)
- PCRE (с 7,2)
- Perl (с 5.10.0)
- PHP (начиная с 5.2.4)
- Ruby (начиная с 2.0.0)

... и (пока) не поддерживается:

- .СЕТЬ
- AWK
- удар
- GNU
- ICU
- Джава
- Javascript
- Notepad ++
- Objective-C
- POSIX
- питон
- Qt / QRegExp
- СЕПГ
- Tcl
- напор
- XML
- XPath

Examples

Поиск и замена с помощью оператора \ K

Учитывая текст:

> foo: bar

Я хотел бы заменить что-нибудь следующее «foo:» на «baz», но я хочу сохранить «foo:». Это можно сделать с помощью группы захвата, например:

```
s/(foo: ).*/$1baz/
```

Что приводит к тексту:

> foo: baz

Пример 1

или мы могли бы использовать `\K` , который «забывает» все, что он ранее сопоставил, с таким шаблоном:

```
s/foo: \K.*/baz/
```

Регулярное выражение соответствует «foo:», а затем встречается с `\K` , предыдущие символы совпадений воспринимаются как должное и оставляются регулярным выражением, что означает, что только строка, соответствующая `.*` Будет заменена на « baz», в результате получится текст:

> foo: baz

Пример 2.

Прочитайте Сброс матча: \ K онлайн: https://riptutorial.com/ru/regex/topic/1338/сброс-матча----k

глава 21: Спасаясь

Examples

Литералы необработанных строк

Это лучше всего для удобства чтения (и вашего здравомыслия), чтобы избежать утечки. Вот здесь и появляются литералы строк. (Обратите внимание, что некоторые языки допускают разделители, которые обычно предпочтительнее строк. Но это еще один раздел.)

Они обычно работают так же, как описывает этот ответ :

> [A] обратная косая черта, `\` , воспринимается как означающая «просто обратная косая черта» (за исключением случаев, когда это происходит прямо перед цитатой, которая в противном случае завершала бы литерал) - нет «escape-последовательностей» для представления новых строк, вкладок, обратных пространств, форм-каналов , и так далее.

Не все языки имеют их, а те, которые используют различный синтаксис. C # фактически называет их *стенографическими строками* , но это то же самое.

ПИТОН

```
pattern = r"regex"
```

```
pattern = r'regex'
```

C ++ (11+)

Синтаксис здесь чрезвычайно универсален. Единственное правило - использовать разделитель, который не появляется нигде в регулярном выражении. Если вы это сделаете, для чего-либо в строке не требуется дополнительное экранирование. Обратите внимание, что скобки `()` не являются частью регулярного выражения:

```
pattern = R"delimiter(regex)delimiter";
```

VB.NET

Просто используйте обычную строку. Обратные косые черты ВСЕГДА литералы .

C

```
pattern = @"regex";
```

Обратите внимание, что этот синтаксис также позволяет `""` (две двойные кавычки) в качестве экранированной формы `"` .

Струны

В большинстве языков программирования, чтобы иметь обратную косую черту в строке, генерируемой из строкового литерала, каждый обратный слэш должен быть удвоен в строковом литерале. В противном случае это будет интерпретироваться как побег для следующего символа.

К сожалению, любая обратная косая черта, требуемая регулярным выражением, должна быть буквальной обратной косой чертой. Вот почему возникает необходимость «избежать экранов» (`\\`), когда регулярные выражения генерируются из строковых литералов.

Кроме того, кавычки (`"` или `'`) в строковом литерале могут быть экранированы, в зависимости от того, что окружает строковый литерал. На некоторых языках можно использовать либо стиль кавычек для строки (выберите наиболее читаемый для избегая всего строкового литерала).

На некоторых языках (например: Java <= 7) регулярные выражения не могут быть выражены непосредственно как литералы, такие как `/\w/` ; они должны быть сгенерированы из строк, и обычно используются строковые литералы - в этом случае `"\\w"` . В этих случаях буквальные символы, такие как кавычки, обратные косые черты и т. Д., Должны быть экранированы. Самый простой способ сделать это можно с помощью инструмента (например, *RegexPlanet*). Этот конкретный инструмент предназначен для Java, но он будет работать для любого языка с аналогичным строковым синтаксисом.

Какие символы нужно избегать?

Экранирование символов - это то, что позволяет определенным символам (зарезервированным движком регулярных выражений для манипуляций поисками) буквально искать и находить во входной строке. Escaping зависит от контекста, поэтому этот пример не охватывает экранирование строки или разделителя .

Обратные косые

Говорить, что обратная косая черта - символ «побега», немного вводит в заблуждение.

Искажения обратной косой черты и обратная косая черта приходят; он фактически переключает или выключает метасимвол или буквенный статус персонажа перед ним.

Чтобы использовать буквальную обратную косую черту в любом месте регулярного выражения, она должна быть экранирована другой обратной косой чертой.

Escaping (внешние классы символов)

Существует несколько символов, которые нужно избегать, чтобы их можно было буквально (по крайней мере, вне классов char):

- Кронштейны: `[]`
- Скобки: `()`
- Кудрявые фигурные скобки: `{}`
- Операторы: `*` , `+` `?` , `|`
- Якоря: `^` , `$`
- Другие: `.` , `\`
- Чтобы использовать литерал `^` в начале или литерал `$` в конце регулярного выражения, символ должен быть экранирован.
- Некоторые ароматы используют только `^` и `$` как метасимволы, когда они находятся в начале или в конце регулярного выражения соответственно. В этих ароматах не требуется дополнительного вылета. Как правило, лучше всего их избежать.

Экранирование в классах символов

- Лучше всего избегать квадратных скобок (`[` и `]`), когда они появляются как литералы в классе char. При определенных условиях это не требуется, в зависимости от вкуса , но это наносит вред читаемости.
- Карет, `^` , является метасимволом, когда он ставится как первый символ в классе char: `[^aeiou]` . В любом другом месте класса char это просто буквальный символ.
- Символ, `-` , является метасимволом, если только в начале или конце символьного класса. Если первым символом в классе char является каретка `^` , то он будет литералом, если он является вторым символом в классе char.

Избежать замены

Существуют также правила экранирования в рамках замены, но ни одно из вышеприведенных правил не применяется. Единственными метасимволами являются `$` и `\` , по крайней мере, когда `$` можно использовать для ссылки на группы захвата (например, `$1` для группы 1). Чтобы использовать литерал `$` , сбегите его: `\$5.00` . Аналогично `\` : `C:\\Program Files\\` .

Исключения из BRE

Хотя ERE (расширенные регулярные выражения) отражает типичный синтаксис в стиле Perl, BRE (основные регулярные выражения) имеет существенные отличия, когда дело доходит до экранирования:

- Существует другой сокращенный синтаксис. Все `\d` , `\s` , `\w` и т. Д. Исчезли. Вместо этого он имеет свой собственный синтаксис (который POSIX смущает называет «классы символов»), например `[:digit:]` . Эти конструкции должны быть в пределах класса символов.
- Есть несколько метасимволов (`.` , `*` , `^` , `$`), Которые можно использовать в обычном режиме. ВСЕ другие метасимволы должны быть экранированы иначе:

Брекеты `{}`

- `a{1,2}` соответствует `a{1,2}` . Чтобы сопоставить либо `a` либо `aa` , используйте `a\{1,2\}`

Скобки `()`

- `(ab)\1` недействительна, так как нет группы захвата 1. Чтобы исправить ее и использовать `abab` используйте `\(ab\)\1`

бэкслэш

- Внутри классов char (которые называются выражениями скобок в POSIX) обратная косая черта не является метасимволом (и не требует экранирования). `[\d]` соответствует `\` или `d` .
- В любом месте убегайте, как обычно.

Другой

- `+` и `?` являются литералами. Если механизм BRE поддерживает их как метасимволы, они должны быть экранированы как `\?` и `\+` .

/ Разделители /

Многие языки позволяют регулярному выражению заключать или делиться между несколькими конкретными символами, как правило, косой чертой `/` .

Разделители оказывают влияние на экранирование: если разделитель есть `/` и регулярное выражение должно искать `/` литералы, тогда передняя косая черта должна быть экранирована, прежде чем она станет буквальной (`\/`).

Чрезмерное избежание удобочитаемости, поэтому важно рассмотреть доступные варианты:

Javascript уникален, потому что он позволяет косую черту в качестве разделителя, но не более того (хотя это позволяет выполнять строгие регулярные выражения).

Perl 1

Perl, например, позволяет почти что-либо быть ограничителем. Даже арабские символы:

```
$str =~ m ش ش
```

Конкретные правила упоминаются в документации Perl .

PCRE допускает два типа разделителей: сопоставленные разделители и разделители в стиле скобок. Соответствующие разделители используют пару одного персонажа, в то время как разделители в стиле скобок используют пару символов, которые представляют собой открывающую и закрывающуюся пару.

- Соответствующие разделители `!"#$%&'*+,./:;=?@^_`|~-`
- Разделители в стиле скобок: `()` , `{}` , `[]` , `<>`

Прочитайте Спасаясь онлайн: https://riptutorial.com/ru/regex/topic/4524/спасаясь

глава 22: Сравнение простых шаблонов

Examples

Сопоставьте символ с одной цифрой, используя [0-9] или \ d (Java)

`[0-9]` и `\d` являются эквивалентными шаблонами (если только ваш механизм Regex не поддерживает кодировку Unicode, а `\d` также совпадает с такими, как ②). Они будут совпадать с символом одной цифры, чтобы вы могли использовать любую нотацию, которую вы считаете более читаемой.

Создайте строку шаблона, который вы хотите сопоставить. Если вы используете нотацию \ d, вам нужно будет добавить вторую обратную косую черту, чтобы избежать первой обратной косой черты.

```
String pattern = "\\d";
```

Создайте объект Pattern. Передайте строку шаблона в метод compile ().

```
Pattern p = Pattern.compile(pattern);
```

Создайте объект Matcher. Передайте строку, которую вы ищете, чтобы найти шаблон в методе matcher (). Проверьте, найден ли шаблон.

```
Matcher m1 = p.matcher("0");
m1.matches(); //will return true

Matcher m2 = p.matcher("5");
m2.matches(); //will return true

Matcher m3 = p.matcher("12345");
m3.matches(); //will return false since your pattern is only for a single integer
```

Соответствие различным номерам

`[ab]` где a и b - цифры в диапазоне от `0` до `9`

```
[3-7] will match a single digit in the range 3 to 7.
```

Соответствие нескольких цифр

```
\d\d       will match 2 consecutive digits
\d+        will match 1 or more consecutive digits
\d*        will match 0 or more consecutive digits
\d{3}      will match 3 consecutive digits
\d{3,6}    will match 3 to 6 consecutive digits
```

```
\d{3,}     will match 3 or more consecutive digits
```

`\d` в приведенных выше примерах можно заменить на ряд чисел:

```
[3-7][3-7]    will match 2 consecutive digits that are in the range 3 to 7
[3-7]+        will match 1 or more consecutive digits that are in the range 3 to 7
[3-7]*        will match 0 or more consecutive digits that are in the range 3 to 7
[3-7]{3}      will match 3 consecutive digits that are in the range 3 to 7
[3-7]{3,6}    will match 3 to 6 consecutive digits that are in the range 3 to 7
[3-7]{3,}     will match 3 or more consecutive digits that are in the range 3 to 7
```

Вы также можете выбрать определенные цифры:

```
[13579]       will only match "odd" digits
[02468]       will only match "even" digits
1|3|5|7|9     another way of matching "odd" digits - the | symbol means OR
```

Соответствие номеров в диапазонах, содержащих более одной цифры:

```
\d|10        matches 0 to 10    single digit OR 10.  The | symbol means OR
[1-9]|10     matches 1 to 10    digit in range 1 to 9 OR 10
[1-9]|1[0-5] matches 1 to 15    digit in range 1 to 9 OR 1 followed by digit 1 to 5
\d{1,2}|100  matches 0 to 100   one to two digits OR 100
```

Соответствующие числа, делящиеся на другие числа:

```
\d*0         matches any number that divides by 10  - any number ending in 0
\d*00        matches any number that divides by 100 - any number ending in 00
\d*[05]      matches any number that divides by 5   - any number ending in 0 or 5
\d*[02468]   matches any number that divides by 2   - any number ending in 0,2,4,6 or 8
```

совпадающие числа, которые делятся на 4 - любое число, которое равно 0, 4 или 8 или заканчивается в 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60 , 64, 68, 72, 76, 80, 84, 88, 92 или 96

```
[048]|\d*(00|04|08|12|16|20|24|28|32|36|40|44|48|52|56|60|64|68|72|76|80|84|88|92|96)
```

Это можно сократить. Например, вместо использования `20|24|28` мы можем использовать `2[048]` . Кроме того, поскольку 40, 60 и 80 имеют один и тот же шаблон, мы можем их включить: `[02468][048]` а остальные имеют шаблон `[13579][26]` . Таким образом, вся последовательность может быть уменьшена до:

```
[048]|\d*([02468][048]|[13579][26])    - numbers divisible by 4
```

Соответствующие номера, которые не имеют такого шаблона, как те, которые делятся на 2,4,5,10 и т. Д., Не всегда могут выполняться лаконично, и вам обычно приходится прибегать к ряду чисел. Например, сопоставление всех чисел, делящихся на 7 в диапазоне от 1 до 50, можно сделать простым перечислением всех этих чисел:

```
7|14|21|28|35|42|49

or you could do it this way

7|14|2[18]|35|4[29]
```

Соответствие пробелов ведущего / конечного пробела

Трейлинг пространства

`\s*$` : Это будет соответствовать любому (`*`) пробелу (`\s`) в конце (`$`) текста

Ведущие пространства

`^\s*` : Это будет соответствовать любому (`*`) пробелу (`\s`) в начале (`^`) текста

замечания

`\s` является общим метасимволом для нескольких движков RegExp и предназначен для захвата пробельных символов (например, пробелов, новых строк и вкладок). **Примечание** : он, вероятно *, не будет* записывать все символы пробела в Юникоде . Проверьте информацию о своих двигателях, чтобы быть уверенным в этом.

Сопоставьте любые поплавки

```
[\+\-]?\d+(\.\d*)?
```

Это будет соответствовать любому подписанному float, если вы не хотите знаков или анализируете уравнение remove `[\+\-]?` поэтому у вас есть `\d+(\.\d+)?`

Объяснение:

- `\d+` соответствует любому целому числу
- `()?` означает, что содержимое круглых скобок является необязательным, но всегда должно появляться вместе
- '\'. match '.', мы должны избегать этого, так как '.' обычно соответствует любому символу

Таким образом, это выражение будет соответствовать

```
5
+5
-5
5.5
```

```
+5.5
-5.5
```

Выбор определенной строки из списка на основе слова в определенном месте

У меня есть следующий список:

```
1. Alon Cohen
2. Elad Yaron
3. Yaron Amrani
4. Yogev Yaron
```

Я хочу выбрать первое имя ребят с фамилией Ярона.

Поскольку я не забочусь о том, что это такое, я просто поставлю его как любую цифру и соответствующую точку и пробел после него с начала строки, например: `^[\d]+\.\s` ,

Теперь нам нужно будет сопоставить пространство и первое имя, так как мы не можем определить, будут ли они капитальными или малыми буквами, которые мы будем `[a-zA-Z]+\s` : `[a-zA-Z]+\s` или `[aZ]+\s` и также может быть `[\w]+\s` .

Теперь мы укажем требуемую фамилию, чтобы получить только строки, содержащие Ярон в качестве фамилии (в конце строки): `\sYaron$` .

Объединим все это `^[\d]+\.\s[\w]+\sYaron$` .

Пример: https://regex101.com/r/nW4fH8/1

Прочитайте Сравнение простых шаблонов онлайн: https://riptutorial.com/ru/regex/topic/343/сравнение-простых-шаблонов

глава 23: Типы выражений с регулярным выражением

Examples

NFA

Двигатель NFA (недетерминированный конечный автомат) *управляется шаблоном* .

Принцип

Шаблон регулярного выражения анализируется на дерево.

Текущий указатель *позиции* установлен на начало входной строки, и в этой позиции выполняется попытка совпадения. Если совпадение fais, позиция увеличивается до следующего символа в строке, а другое совпадение выполняется из этой позиции. Этот процесс повторяется до тех пор, пока не будет найдено совпадение или не будет достигнут конец входной строки.

Для каждой попытки матча

Алгоритм работает путем выполнения обхода дерева шаблонов для заданной начальной позиции. По мере продвижения по дереву он обновляет *текущую позицию ввода* , потребляя соответствующие символы.

Если алгоритм встречается с узлом дерева, который не соответствует входной строке в текущей позиции, ему придется *отступить* . Это выполняется, возвращаясь к родительскому узлу в дереве, сбросив текущую позицию ввода до значения, которое оно имело при вводе родительского узла, и попробовав следующую альтернативную ветку.

Если алгоритму удается выйти из дерева, он сообщает об успешном совпадении. В противном случае, когда все возможности были опробованы, совпадение не выполняется.

Оптимизации

Двигатели Regex обычно применяют некоторые оптимизации для лучшей производительности. Например, если они определяют, что совпадение должно начинаться с заданного символа, они будут пытаться выполнить совпадение только в тех положениях

входной строки, где отображается этот символ.

пример

Сопоставьте `a(b|c)a` с входной строкой `abeacab` :

Дерево шаблонов может выглядеть примерно так:

```
CONCATENATION
    EXACT: a
    ALTERNATION
        EXACT: b
        EXACT: c
    EXACT: a
```

Сопоставление выполняется следующим образом:

```
a(b|c)a      abeacab
^            ^
```

`a` находится во входной строке, потребляет ее и переходит к следующему элементу в дереве шаблонов: чередование. Попробуйте первую возможность: точную `b` .

```
a(b|c)a      abeacab
  ^            ^
```

`b` , поэтому чередование завершается успешно, потребляет его и переходит к следующему элементу в конкатенации: точный `a` :

```
a(b|c)a      abeacab
      ^        ^
```

`a` *не* находится в ожидаемом положении. Возвратитесь к чередованию, сбросьте позицию ввода до значения, которое она имела при входе в чередование в первый раз, и попробуйте *вторую* альтернативу:

```
a(b|c)a      abeacab
    ^         ^
```

`c` *не* находится в этом положении. Откат к конкатенации. На данный момент нет других возможностей попробовать, поэтому в начале строки нет совпадений.

Попытайтесь выполнить второе совпадение в следующей позиции ввода:

```
a(b|c)a      abeacab
^             ^
```

`a` *не* соответствует. Попытайтесь провести еще одно совпадение на следующей позиции:

```
a(b|c)a      abeacab
^              ^
```

Не повезло. Перейдите на следующую позицию.

```
a(b|c)a      abeacab
^                ^
```

матчи, поэтому потреблять его и ввести чередование:

```
a(b|c)a      abeacab
  ^                ^
```

`b` не соответствует. Попробуйте вторую альтернативу:

```
a(b|c)a      abeacab
    ^            ^
```

`c` , поэтому потребляйте его и переходите к следующему элементу в конкатенации:

```
a(b|c)a      abeacab
      ^            ^
```

`a` спички, и конец дерева были достигнуты. Сообщить об успешном матче:

```
a(b|c)a      abeacab
               \_/
```

DFA

Входной сигнал управляется двигателем DFA (детерминированный конечный автомат).

Принцип

Алгоритм сканирует входную строку *один раз* и запоминает все возможные пути в регулярном выражении, которые могут совпадать. Например, когда в шаблоне встречается чередование, создаются два новых пути и предпринимаются попытки независимо. Когда данный путь не соответствует, он отбрасывается из набора возможностей.

Последствия

Время согласования ограничено размером входной строки. Отказов нет, и двигатель

может найти несколько совпадений одновременно, даже совпадающих совпадений.

Основным недостатком этого метода является уменьшенный набор функций, который может поддерживаться двигателем по сравнению с типом двигателя NFA.

пример

Сопоставьте `a(b|c)a` против `abadaca` :

```
abadaca      a(b|c)a
^            ^        Attempt 1      ==> CONTINUE

abadaca      a(b|c)a
 ^           ^        Attempt 2      ==> FAIL
               ^      Attempt 1.1    ==> CONTINUE
                  ^   Attempt 1.2    ==> FAIL

abadaca      a(b|c)a
  ^          ^        Attempt 3      ==> CONTINUE
                 ^    Attempt 1.1    ==> MATCH

abadaca      a(b|c)a
   ^         ^        Attempt 4      ==> FAIL
               ^      Attempt 3.1    ==> FAIL
                  ^   Attempt 3.2    ==> FAIL

abadaca      a(b|c)a
    ^        ^        Attempt 5      ==> CONTINUE

abadaca      a(b|c)a
     ^       ^        Attempt 6      ==> FAIL
               ^      Attempt 5.1    ==> FAIL
                  ^   Attempt 5.2    ==> CONTINUE

abadaca      a(b|c)a
      ^      ^        Attempt 7      ==> CONTINUE
                 ^    Attempt 5.2    ==> MATCH

abadaca      a(b|c)a
       ^       ^      Attempt 7.1    ==> FAIL
                  ^   Attempt 7.2    ==> FAIL
```

Прочитайте Типы выражений с регулярным выражением онлайн:
https://riptutorial.com/ru/regex/topic/2861/типы-выражений-с-регулярным-выражением

глава 24: Якорные персонажи: доллар ($)

замечания

Большое количество двигателей регулярных выражений использует режим « многострочный» для поиска нескольких строк в файле независимо.

Поэтому при использовании `$` эти двигатели будут соответствовать окончаниям всех строк. Тем не менее, двигатели, которые не используют этот тип многострочного режима, будут соответствовать только последней позиции строки, предоставленной для поиска.

Examples

Сопоставьте букву в конце строки или строки

```
g$
```

Вышеупомянутое соответствует одной букве (буква `g`) в конце *строки* в большинстве движков регулярных выражений (не в Oniguruma , где `$` anchor соответствует концу строки по умолчанию, а модификатор `m` (*MULTILINE*) используется для создания `.` соответствуют любым символам, включая символы разрыва строки, в качестве модификатора DOTALL в большинстве других ароматизаторов NFA). `$` Anchor будет соответствовать первому вхождению буквы `g` до конца следующих строк:

В следующих предложениях выделяются только буквы, **выделенные жирным шрифтом** :

> Якорями являются символы, которые, по сути, не соответствуют ни одному символу в strin **g**
>
> Их цель - сопоставить определенную позицию в этой строке.
>
> Боб был helpin **г**
>
> Но его редактирование ввело примеры, которые не соответствовали друг другу!

В большинстве регулярных выражений флаги `$` anchor могут также соответствовать перед символом новой строки или символом прерывания строки (последовательности) в режиме *MULTILINE* , где `$` совпадает в конце каждой строки, а не только в конце строки. Например, снова используя `g$` в нашем регулярном выражении, в многострочном режиме, курсивом в следующей строке будет соответствовать:

```
tvxlt obofh necpu riist g\n aelxk zlhdx lyogu vcbke pzyay wtsea wbrju jztg\n drosf ywhed bykie lqmzg wgyhc lg\n qewrx ozrvm jwenx
```

Прочитайте Якорные персонажи: доллар ($) онлайн: https://riptutorial.com/ru/regex/topic/1603/якорные-персонажи--доллар----

глава 25: Якорные персонажи: Карет (^)

замечания

терминология

Характер Caret (^) также обозначается следующими терминами:

- шапка
- контроль
- стрелка вверх
- шеврон
- округлый акцент

использование

Он имеет два применения в регулярных выражениях:

- Чтобы обозначить начало строки
- Если он используется сразу после квадратной скобки (`[^`), он действует, чтобы свести на нет множество допустимых символов (т. Е. `[123]` означает, что символ 1, 2 или 3 разрешен, а оператор `[^123]` означает любой символ, отличный от 1 , 2 или 3.

Сбой символа

Чтобы выразить каретку без особого значения, ее следует избегать, предшествуя ей обратным слэшем; т. е. `\^` .

Examples

Начало линии

Когда многострочный `(?m)` модификатор выключен , `^` соответствует только началу входной строки:

Для регулярного выражения

```
^He
```

Следующие строки ввода соответствуют:

- `Hedgehog\nFirst line\nLast line`
- `Help me, please`
- `He`

И следующие строки ввода **не** совпадают:

- `First line\nHedgehog\nLast line`
- `IHedgehog`
- `Hedgehog` (из-за белых пространств)

Когда многострочный `(?m)` модификатор включен, `^` совпадает с началом каждой линии в:

```
^He
```

Вышеупомянутое будет соответствовать любой входной строке, содержащей строку, начинающуюся с `He` .

Учитывая `\n` как новый символ линии, следующие строки соответствуют:

- `Hello`
- `First line\nHedgehog\nLast line` (только вторая строка)
- `My\nText\nIs\nHere` (только последняя строка)

И следующие строки ввода **не** совпадают:

- `Camden Hells Brewery`
- `Helmet` (из-за белых пространств)

Соответствие пустых строк с помощью `^`

Другим типичным вариантом использования для каретки является сопоставление пустых строк (или пустая строка, если многострочный модификатор выключен).

Чтобы соответствовать пустой строке (multi-line **on**), каретка используется рядом с `$` который является другим символом привязки, представляющим позицию в конце строки (Anchor Characters: Dollar ($)). Поэтому следующее регулярное выражение будет соответствовать пустой строке:

```
^$
```

Избегание символа каретки

Если вам нужно использовать символ `^` в символьном классе (классы символов), поставьте его где-то иначе, чем начало класса:

```
[12^3]
```

Или избегайте `^` используя обратную косую черту `\` :

```
[\^123]
```

Если вы хотите совместить символ каретки вне класса персонажа, вам нужно сбежать от него:

```
\^
```

Это предотвращает интерпретацию `^` как якорный символ, представляющий начало строки / строки.

Сравнение начала линейного якоря и начала привязки строки

Хотя многие считают, что `^` означает начало строки, это на самом деле означает начало строки. Для фактического начала использования привязки строк, `\A`

Строка `hello\nworld` (или более ясно)

```
hello
world
```

Соответствовали бы регулярным выражениям `^h` , `^w` и `\Ah` но не `\Aw`

Многострочный модификатор

По умолчанию каретки `^` метасимвол соответствует **положению** перед первым символом в строке.

Учитывая, что строка « **charsequence** » применяется к следующим шаблонам: `/^char/` & `/^sequence/` , движок будет пытаться соответствовать следующим образом:

- `/^char/`
    - **^** - CharSequence
    - **c** - `c` harsequence
    - **h** - `ch` arsequence
    - **a** - `cha` rsequence
    - **r** - последовательность `char`

  Найдено совпадение

- `/^sequence/`
    - **^** - CharSequence

- **s** - CharSequence

Матч не найден

Такое же поведение будет применено, даже если строка содержит *ограничители строк* , такие как `\r?\n` Будет соответствовать только позиция в начале строки.

Например:

```
/^/g
```

┊ char \ r \ n
\ Г \ п
последовательность

Однако, если вам нужно сопоставлять после каждого терминатора линии, вам нужно будет установить **многострочный** режим (`//m` , `(?m)`) внутри вашего шаблона. Поступая таким образом, каретка `^` будет соответствовать «начало каждой строки», что соответствует положению в начале строки и позиции **сразу же после** того, как [1] линия терминаторов.

[1] В некоторых вариантах (Java, PCRE, ...), `^` не будет совпадать после терминатора строк, если терминатор линии является последним в строке.

Например:

```
/^/gm
```

┊ char \ r \ n
┊ \ r \ n
┊ sequence

Некоторые из двигателей регулярных выражений, которые поддерживают модификатор Multiline:

- Джава

```
Pattern pattern = Pattern.compile("(?m)^abc");
Pattern pattern = Pattern.compile("^abc", Pattern.MULTILINE);
```

- .СЕТЬ

```
var abcRegex = new Regex("(?m)^abc");
var abdRegex = new Regex("^abc", RegexOptions.Multiline)
```

- PCRE

```
/(?m)^abc/
```

```
/^abc/m
```

- Python 2 и 3 (встроенный модуль `re`)

```
abc_regex = re.compile("(?m)^abc");
abc_regex = re.compile("^abc", re.MULTILINE);
```

Прочитайте Якорные персонажи: Карет (^) онлайн: https://riptutorial.com/ru/regex/topic/452/якорные-персонажи--карет----

кредиты

S. No	Главы	Contributors
1	Начало работы с регулярными выражениями	0rkan, Addison, balpha, Community, Confiqure, Ibrahim, J F, JelmerS, JohnLBevan, Kendra, Laurel, Maria Deleva, Mariano, Mateus, mnoronha, Rudy M, Stephen Leppik, Tot Zam, TylerH, Wolf, Yaron, zmo
2	Lookahead и Lookbehind	BoppreH, hwnd, Lucas Trzesniewski, Maria Deleva, Wiktor Stribiżew
3	UTF-8: письма, знаки, знаки препинания и т. Д.	mudasobwa
4	Атомная группировка	OnlineCop
5	Граница Word	cdm, jonathanking, kdhp, Maria Deleva, Peter G, rgoliveira, Tushar
6	Группы захвата	Addison, Alan Moore, Lucas Trzesniewski, Tomalak, Vogel612
7	Жадные и ленивые кванторы	0rkan, Confiqure, David Knipe, GradientByte, Laurel, Mario, Mark Stewart, Nathan Arthur, nhahtdh, phatfingers, sweaver2112, Thomas Ayoub, Tim Pietzcker
8	Замены с регулярными выражениями	Mateus
9	Именованные группы захвата	Thomas Ayoub
10	Классы символов	Acey, CPHPython, Dmitry Bychenko, HamZa, kdhp, Lucas Trzesniewski, Maria Deleva, RamenChef, rgoliveira, rock321987, Wiktor Stribiżew
11	Когда вы НЕ должны использовать регулярные выражения	dorukayhan, Kendra, zmo

12	Ловушки Regex	BrightOne
13	Модификаторы регулярных выражений (флаги)	Eder, Mateus, Tim Pietzcker, Wiktor Stribiżew
14	Обратная ссылка	Alan Moore, Kendra, OnlineCop
15	Откат	dorukayhan, Mike, Miljen Mikic, SQB, Thomas Ayoub, Vituel
16	Полезная Regex Витрина	depperm, Devid Farinelli, Echelon, Herb, Kendra, Matas Vaitkevicius, nhahtdh, Sebastian Lenartowicz, Steve Chambers, Thomas Ayoub, Tomasz Jakub Rup, zmo
17	Потенциальные квантификаторы	Mark Hurd, Sebastian Lenartowicz
18	Регулярное повторение пароля	rock321987
19	Рекурсия	Keith Hall, Laurel, Lucas Trzesniewski, user23013
20	Сброс матча: \ K	nhahtdh, Wiktor Stribiżew, Will Barnwell
21	Спасаясь	CPHPython, David Knipe, Laurel
22	Сравнение простых шаблонов	balpha, GradientByte, Graham, Joe, Mariano, rgoliveira, Tot Zam, Yaron
23	Типы выражений с регулярным выражением	Lucas Trzesniewski, Markus Jarderot
24	Якорные персонажи: доллар ($)	ArtOfCode, CPHPython, hjpotter92, Kendra, rubayet.R, Tom Lord, UNagaswamy, Wiktor Stribiżew
25	Якорные персонажи: Карет (^)	CPHPython, Eder, J F, JohnLBevan, Jojodmo, knut, Mateus, Mike H-R, Mr. Deathless, nhahtdh, revo, rgoliveira, Tom Lord, zb226