

Регулярные выражения

#Общее

#РаботаСоСтроками

#Расширенный_POSIX

Регулярные выражения это шаблоны текста, который нам нужно найти.

Если мы ищем конкретную букву, например "а", то регулярное выражение будет выглядеть следующим образом:

```
а
```

Схожим образом можно задать и слова, например "Арка"

Варианты:

```
Арка
МояАрка
арка
моярка
аркан
Аркан
```

Регулярное выражение:

```
Арка
```

Результат поиска:

```
Арка
МояАрка
Аркан
```

Из примера можно увидеть, что результат поиска дал нам не совсем то, что нужно. Нас интересовало только "Арка".

 **Регулярные выражения ПО УМОЛЧАНИЮ регистрозависимые, т.е. 'А' и 'а' это разные символы.**

ПО УМОЛЧАНИЮ - намекает на то, что этим можно управлять.

На одну позицию строки можно задавать несколько символов, которые нас устроят.

Делается это при помощи **символьных классов** и задаются они при помощи квадратных скобок следующим синтаксисом:

```
[Символ_1Символ_2Символ_N]
```

или

```
[Начало_диапазона_1-Конец_диапазона_1Начало_диапазона_2-Конец_диапазона_2СимволNНачало_диапазона_N-Конец_диапазона_N]
```

🔥 Так как квадратные скобки используются для формирования регулярных выражений, для поиска самих символов '[' ']' в тексте их нужно экранировать, т.е. добавить перед ними символ '\\'.

Список большинства символов, которые нужно экранировать, будет представлен в конце .

⚠️ Обратите внимание, что разделителей внутри квадратных скобок нет. Никаких пробелов, запятых, точек и пр. только '-' как обозначение диапазона. Точка, запятая, пробел это все символы и если вы их вводите в символьный класс - они включаются в выборку, что может привести к нежелательным результатам.

Также внутри символьного класса можно использовать символ '^' для того, чтобы сделать не "белый список", а "чёрный список", т.е. вместо поиска только перечисленных символов, мы наоборот, ищем всё, кроме этих символов.

Синтаксис:

```
[^Символ1Символ2Начало_диапазона_N-Конец_диапазона_NСимволN]
```

Пример:

Варианты:

а
б
в
г
д
е

Регулярное выражение:

```
[^абг]
```

Результат поиска:

в
д
е

Понятно, что вам нужно будет фильтровать некоторые конкретные группы символов, такие как 'только буквы' или 'только цифры', особенно часто. Чтобы постоянно не писать выражения по типу [А-Яа-яЁё] для фильтрации текста, были продуманы специальные **метасимволы**:

Символ	Символьный класс	Соответствия
.		Вообще любой символ
\d	[0-9]	Цифры
\D	[^0-9]	Все, кроме цифр
\s	[\f\n\r\t\v]	Пробельные символы
\S	[^\f\n\r\t\v]	Все, кроме пробельных символов

Символ	Символьный класс	Соответствия
\w	[A-Za-z0-9_]	Буквенный или цифровой символы или знак подчёркивания
\W	[^A-Za-z0-9_]	Все, кроме букв, цифр и знака подчёркивания

🔗 Можно запомнить только 3 буквы: 'd' (digit), 's' (space), 'w' (word), а если хотим отрицание, то просто записать их с заглавной.

Также вместо диапазонов можно использовать заранее прописанные классы:

Класс символов	Пояснение
[:alnum:]	Буквы или цифры
[:alpha:]	Только буквы
[:digit:]	Только цифры
[:graph:]	Только отображаемые символы (не учитываются пробелы, \t, \n и подобное)
[:print:]	Отображаемые символы и пробелы
[:space:]	Пробельные символы
[:punct:]	Знаки пунктуации (! " \$ % (+ / \ < = и т.д)
[:word:]	\w

⚠️ Во время поиска букв, при помощи метасимволов и готовых классов, могут возникнуть проблемы с символами, которых нет в латинском алфавите, однако, в большинстве случаев, это решается корректной кодировкой.

Нижe описаны **пробельные символы**, которые также можно указать в список допустимых символов:

Символ	Пояснение
	Пробел
\r	Возврат курсора на начало строки
\n	Постановка курсора на новую строку
\t	Табуляция
\v	Вертикальная табуляция
\f	Конец страницы
\b	Возврат на 1 символ

🔗 Поиск '\b' не значит, что можно найти все места, где текст стирался. Он используется, если автор заменял символы явно, при наборе.

При создании строка имеет вид: "abc\bdefg"

При отображении она будет выглядеть: "abdefg"

Вот такие '\b' и будут найдены регулярным выражением.

Бывает, что необходимо найти либо вариант 1, либо вариант 2, либо вариант N. Для этого используется **перечисление**, при котором вместо "либо" пишется символ '|'. Например, `gray|grey` или `gr(a|e)y` найдут строки `gray` или `grey`. Однако, в данном случае, это эквивалентно `gr[ae]y` и в ситуациях когда нам нужно

выбрать только среди односимвольных альтернатив лучше использовать его, так как сравнение с символьным классом быстрее, чем обработка *группы* (содержание круглых скобок).

❓ Группировку рассмотрим позднее.

⚠ Символ '|' работает СО ВСЕМ содержимым, которое находится слева и справа от него в одной группе. Это значит, что выражения `gr(a)|(e)y` или `gra|ey` не будут работать корректно в данном контексте.

Иногда нужно также учитывать позицию найденного фрагмента в строке, для этого используются специальные символы:

Символ	Позиция	Пример	Соответствие
^	Начало текста (или строки при модификаторе ?m)		aaa a aaa
\$	Конец текста (или строки при модификаторе ?m)	a\$	aaa a aaa
\b	Граница слова	\ba\b	aaa a aaa
\B	Не граница слова	\Ba\B	aaa a aaa
\G	Предыдущий успешный поиск	\Ga	aaa a aaa (поиск прекратился на 4 позиции, тк там не нашлось a)

❓ Модификаторы рассмотрим позднее.

Большая часть, что описана выше - используется для фильтрации символов на одной конкретной позиции. Т.е, исходя из имеющейся на данный момент информации, для того, чтобы найти строку и 5 цифр надо составить следующее регулярное выражение:

```
\d\d\d\d\d
```

Было бы печально, если бы это был единственный способ. К счастью, существуют *квантификаторы*.

Квантификатор	Число повторений
?	Ноль или одно
*	Ноль или более
+	Одно или более
{n}	Ровно n раз
{m,n}	От m до n(включительно) раз
{m,}	Не менее m раз
{,n}	Не более n раз

Синтаксис:

```
Допустимые_значенияКвантификатор
или
ГруппаКвантификатор
```

Пример:

Варианты:

```
aaaaaaaaa
aaa
aa
a
б
ббббббб
абабабаб
ааабббб
(пустая строка)
```

Регулярные выражения:

- 1.) .*
- 2.) а*
- 3.) (аа)*
- 4.) а+
- 5.) [аб]*
- 6.) (аб)+
- 7.) а?
- 8.) а{3}

Результат:

- 1.)

```
aaaaaaaaa
aaa
aa
a
б
ббббббб
абабабаб
ааабббб
(пустая строка)
```
- 2.)

```
aaaaaaaaa
aaa
aa
a
(пустая строка)
```
- 3.)

```
aaaaaaaaa
aa
(пустая строка)
```
- 4.)

```
aaaaaaaaa
aaa
aa
a
```
- 5.)

```
aaaaaaaaa
```

```

aaa
aa
a
б
ббббббб
абабабаб
aaaббббб
(пустая строка)
6.)
абабабаб
7.)
а
(пустая строка)
8.)
aaa

```

Существует несколько реализаций **квантификаторов**:

Тип	Как задать	Что находит	Пример
Ленивый	*? +? {n,}?	Минимально длинная строка из возможных	абабаббаабабабабаббааб
Жадный	* + {n,} ?	Максимально длинная строка из возможных	абабаббаабабабабаббааб
Ревнивый	*+ ?+ ++ {n,}+	Первопопавшееся совпадение	абабаббаабабабабаббааб

Теперь рассмотрим **группировку**. **Группировкой** в регулярных выражениях простое взятие в круглые скобки какого-то фрагмента, а сам этот фрагмент именуется **группой**. Помимо очевидного применения **групп** для обозначения порядка выполнения каких-то операций или обозначений границ обрабатываемых символов для **перечисления**, **квантификаторов** или чего-то ещё, у них есть и другая функция - повторное использование найденной подстроки.

Пример:

Варианты:

```

<h1>SomeText</h1>
<h2>SomeText</h2>
<h1>SomeText</h3>
<h2>SomeText<h2>

```

Это теги для html разметки, первые две строки - корректны, а остальные две - нет. Нам нужно найти только те строки, где теги выставлены правильно, т.е. уровень заголовка совпадает и во втором теге присутствует '/'. Сделать это можно при помощи следующего регулярного выражения:

```
<(.)>[^<>]+</\1>
```

Результат:

```

<h1>SomeText</h1>
<h2>SomeText</h2>

```

Как так?

Все **группы** нумеруются по порядку появления.

```
(#1(#2) (#3(#4) (#5))) (#6(#7) (#8))
```

При обработке выражения подстроки, найденные по шаблону внутри **группы**, сохраняются в отдельной области памяти и в дальнейшем мы можем к ним обращаться при помощи выражения `\Номер_группы` (обычно сохраняется до 9 **групп**).

Теперь, если разбирать пример, можно понять, что при прохождении, например первой строки, мы сохранили значение `h1` в **группу** #1 и проверили соответствие содержания закрывающего тега содержанию первой **группы**.

Такая **группировка** называется **группировкой с обратной связью**.

⚡ **Группировка с обратной связью задается следующим образом:**

```
(содержание)
```

Но если мы не хотим сохранять результат **группы**? Тогда можно использовать **группировку без обратной связи**.

Под результат такой **группировки** не выделяется отдельная область памяти и она не тратит наш лимит в 9 **групп**.

⚡ **Группировка без обратной связи задается следующим образом:**

```
(?: содержание)
```

Также существует **атомарная группировка**. Она эквивалентна **группировке без обратной связи**, но запрещает возвращаться назад, если шаблон найден.

⚡ **Атомарная группировка задается следующим образом:**

```
(?>содержание)
```

Чтобы лучше понимать разницу между **группировкой без обратной связи** и **атомарной группировкой** рассмотрим пример:

Пример	Соответствие
<code>a(?:bc b x)cc</code>	<code>abccaxcc</code> и <code>abccaxcc</code>
<code>a(?:>bc b x)cc</code>	<code>abccaxcc</code> (вариант 'x' найден, возвращаться смысла нет)
<code>a(?:>x*)ха</code>	<code>аххха</code> (не найден ничего, так как все 'x' заняты и нет возврата внутрь группы)

В регулярных выражениях также существуют **модификаторы**, которые изменяют обработку регулярного выражения.

Модификатор	Описание часть 1	Описание часть 2
<code>(?i)</code> <code>(?-i)</code>	Включает Выключает	нечувствительность выражения к регистру символов.

Модификатор	Описание часть 1	Описание часть 2
(?s) (?-s)	Включает Выключает	режим соответствия точки символам \r и \n.
(?m) (?-m)	Символы ^ и \$ вызывают соответствие только	после и до символов новой строки. с началом и концом текста.
(?x) (?-x)	Включает Выключает	Режим без учёта пробелов между частями регулярного выражения и позволяет использовать # для комментариев.

Пример использования:

Варианты:


- TVset
- tvset
- tvSET
- TVSET

Регулярные выражения:

- 1.) (?i:tv)set
- 2.) tvset
- 3.) (?i)tvset

Результат:

- 1.)
 - TVset
 - tvset
- 2.)
 - tvset
- 3.)
 - TVset
 - tvset
 - tvSET
 - TVSET

 **Внутри группы можно сделать комментарий.**

Синтаксис: (?#Комментарий)

Пример: A(?:#мой текст)B обрабатывается как AB

В большинстве реализаций регулярных выражений есть способ производить поиск фрагмента текста, «просматривая» (но не включая в найденное) окружающий текст, который расположен до или после искомого фрагмента текста.

Синтаксис	Вид просмотра	Пример	Соответствие
(? =шаблон)	Позитивный просмотр вперёд	Людовик(?=XVI)	ЛюдовикXV, ЛюдовикXVI, ЛюдовикXVIII, ЛюдовикLXVII, ЛюдовикXXL
(?!шаблон)	Негативный просмотр вперёд (с	Людовик(?!XVI)	ЛюдовикXV, ЛюдовикXVI, ЛюдовикXVIII, ЛюдовикLXVII, ЛюдовикXXL

Синтаксис	Вид просмотра	Пример	Соответствие
	отрицанием)		
(?<=шаблон)	Позитивный просмотр назад	(?<=Сергей)Иванов	Сергей Иванов, Игорь Иванов
(?!шаблон)	Негативный просмотр назад (с отрицанием)	(?!Сергей)Иванов	Сергей Иванов, Игорь Иванов

И, наконец, **поиск по условию**.

Возможность выбирать, по какому пути пойдёт проверка в том или ином месте регулярного выражения на основании уже найденных значений.

Синтаксис	Пояснение	Пример	Соответствие
(?(? =если)то иначе)	Если операция просмотра успешна, то далее выполняется часть <code>то</code> , иначе выполняется часть <code>иначе</code> . В выражении может использоваться любая из четырёх операций просмотра. Следует учитывать, что операция просмотра нулевой ширины, поэтому части <code>то</code> в случае позитивного или <code>иначе</code> в случае негативного просмотра должны включать в себя описание шаблона из операции просмотра.	(?(?<=a)м п)	мам, пап
(?(n)то иначе)	Если n-я группа вернула значение, то поиск по условию выполняется по шаблону <code>то</code> , иначе по шаблону <code>иначе</code> .	(a)?(?(1)м п)	мам, пап

☰ Обещанный список символов, которые нужно экранировать.

- []
- { }
- ()
- \
- /
- ^
- \$
- .
- |
- ?
- *
- +

🔗 Чтобы сразу экранировать несколько символов можно использовать конструкцию `\Qсимволы\E`

📖 Источники.

[Регулярные выражения \(regex\) — основы / Хабр \(habr.com\)](#) Дата обращения: 05.11.2023

[Регулярные выражения — Википедия \(wikipedia.org\)](#) Дата обращения: 05.11.2023