**Регулярные выражение (regex)**

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

using System.Text.RegularExpressions;

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else

{

Console.WriteLine("Совпадений не найдено");

}

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод *Matches* класса *Regex* принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект *Match*. Его свойство *Value* возвращает найденное совпадение.

И в данном случае мы получим следующий консольный вывод

**тупогуб**

**тупогубенький**

**тупа**

**Параметр RegexOptions**

Класс *Regex* имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление *RegexOptions*. Некоторые из значений, принимаемых данным перечислением:

* ***Compiled***: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* ***CultureInvariant***: при установке этого значения будут игнорироваться региональные различия
* ***IgnoreCase***: при установке этого значения будет игнорироваться регистр
* ***IgnorePatternWhitespace***: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* ***Multiline***: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* ***RightToLeft***: приписывает читать строку справа налево
* ***Singleline***: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

**Например:**

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.IgnoreCase);

При необходимости можно установить несколько параметров:

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.Compiled | RegexOptions.IgnoreCase);

**Синтаксис регулярных выражений**

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

* **^:** соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")
* **$:** конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
* **.:** знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
* **\*:** предыдущий символ повторяется 0 и более раз
* **+:** предыдущий символ повторяется 1 и более раз
* **?:** предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D** : соответствует любому символу, не являющемуся десятичной цифрой

Это только небольшая часть элементов. Более подробное описание синтаксиса регулярных выражений можно найти на msdn в статье [Элементы языка регулярных выражений — краткий справочник.](https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference)

Теперь посмотрим на некоторые примеры использования. Возьмем первый пример с скороговоркой "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и найдем в ней все слова, где встречается корень "губ":

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"\w\*губ\w\*");

Так как выражение **\w\*** соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Второй простенький пример - нахождение телефонного номера в формате 111-111-1111:

string s = "456-435-2318";

Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}");

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

string s = "456-435-2318";

Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

Также можно задать диапазон для алфавитных символов: Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v.

Можно также указать отдельные значения: Regex regex = new Regex(@"[2]\*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)

С помощью операции | можно задать альтернативные символы, например:

Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}");

То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Итак, у нас такие символы, как \*, + и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если нам надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем:

Regex regex = new Regex(@"(2|3){3}\.[0-9]{3}\.\d{4}");

// этому выражению будет соответствовать строка "222.222.2222"

**Проверка на соответствие строки формату**

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод IsMatch, который позволяет проверить входную строку с шаблоном на соответствие:

using System.Text.RegularExpressions;

string pattern = @"^(?("")(""[^""]+?""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`\{\}\|~\w])\*)(?<=[0-9a-z])@))" +

@"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]\*[0-9a-z]\*\.)+[a-z0-9]{2,17}))$";

var data = new string[]

{

"tom@gmail.com",

"+12345678999",

"bob@yahoo.com",

"+13435465566",

"sam@yandex.ru",

"+43743989393"

};

Console.WriteLine("Email List");

for(int i = 0; i < data.Length; i++)

{

if (Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase))

{

Console.WriteLine(data[i]);

}

}

Переменная pattern задает регулярное выражение для проверки адреса электронной почты. Данное выражение предлагает нам Microsoft на страницах msdn.

Далее в цикле мы проходим по массиву строк и определяем, какие строки соответствуют этому шаблону, то есть представляют валидный адрес электронной почты. Для проверки соответствия строки шаблону используется метод IsMatch: Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase). Последний параметр указывает, что регистр можно игнорировать. И если строка соответствует шаблону, то метод возвращает true.

**Замена и метод Replace**

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

string text = "Мама мыла раму. ";

string pattern = @"\s+";

string target = " ";

Regex regex = new Regex(pattern);

string result = regex.Replace(text, target);

Console.WriteLine(result);

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

Другой пример - на вход подается номер телефона в произвольном формате, и мы хотим оставить в нем только цифры:

string phoneNumber = "+1(876)-234-12-98";

string pattern = @"\D";

string target = "";

Regex regex = new Regex(pattern);

string result = regex.Replace(phoneNumber, target);

Console.WriteLine(result); // 18762341298

В данном случае шаблон "\D" представляет любой символ, который не является цифрой. Любой такой символ заменяется на пустую строку "", то есть в итоге из строки "+1(876)-234-12-98" мы получим строку "18762341298".

## Как правильно писать регулярные выражения

Прежде, чем садиться и писать, подумайте, что именно вы хотите сделать. Регулярное выражение должно начинаться с мысли "Я хочу найти/заменить/удалить то-то и то-то". Затем вам нужен исходный текст, который содержит как ПРАВИЛЬНЫЕ, так и НЕправильные данные. Затем вы открываете <https://regex101.com/>, вставляете текст и начинаете писать регулярное выражение. Этот инструмент укажет и покажет все ошибки, а также подсветит результаты поиска.

Для примера возьмём валидацию ip-адреса. Первая мысль должна быть: "Я хочу валидировать ip-адрес. А что такое ip-адрес? Из чего он состоит?". Затем нужен список валидных и невалидных адресов:

# Валидные адреса

0.0.0.0

0.1.2.3

99.99.99.99

199.199.199.199

255.255.255.255

# Невалидные адреса

01.01.01.01

.1.2.3

1.2.3.

255.0.0.256

Валидный адрес должен содержать четыре числа (байта) от 0 до 255. Если он содержит число больше 255, это уже ошибка. Если бы мы делали валидацию на каком-либо языке программирования, то можно было бы разбить выражение на четыре части и проверить каждое число отдельно. Но регулярные выражения не поддерживают проверки больше или меньше, поэтому придётся делать по-другому.

Для начала упростим задачу: будем валидировать не весь ip-адрес, а только один байт. А байт это всегда есть либо одно-, либо дву-, либо трёхзначное число. Для одно- и двузначного числа шаблон очень простой - любая цифра. А вот для трёхзначного числа первая цифра либо единица, либо двойка. Если первая цифра единица, то вторая и третья могут быть от нуля до девяти. Если же первая цифра двойка, то вторая может быть только от нуля до пяти. Если первая цифра двойка и вторая пятёрка, то третья может быть только от ноля до пяти. Давайте формализуем:

# Валидация байта

от 0 до 9 \d

от 10 до 99 [1-9]\d

от 100 до 199 1\d\d

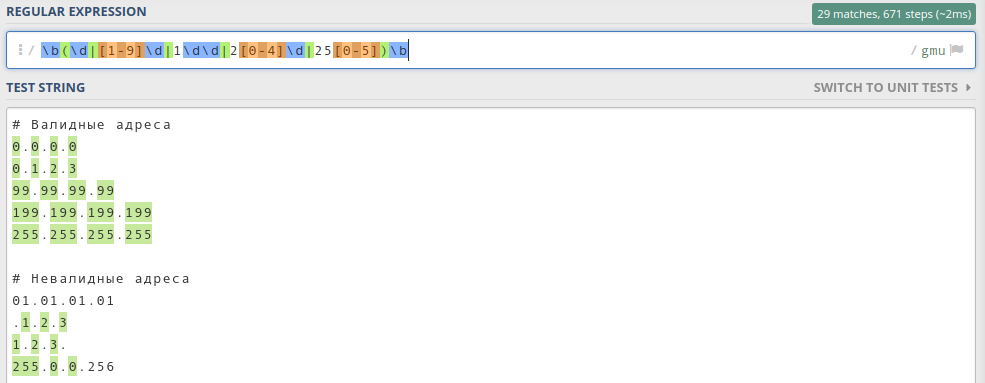
от 200 до 249 2[0-4]\d

от 250 до 255 25[0-5]

Теперь, зная все диапазоны байта, можно объединить их в одно выражение через вертикальную палочку | (ИЛИ):

\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b

Пробуем регулярку в деле:

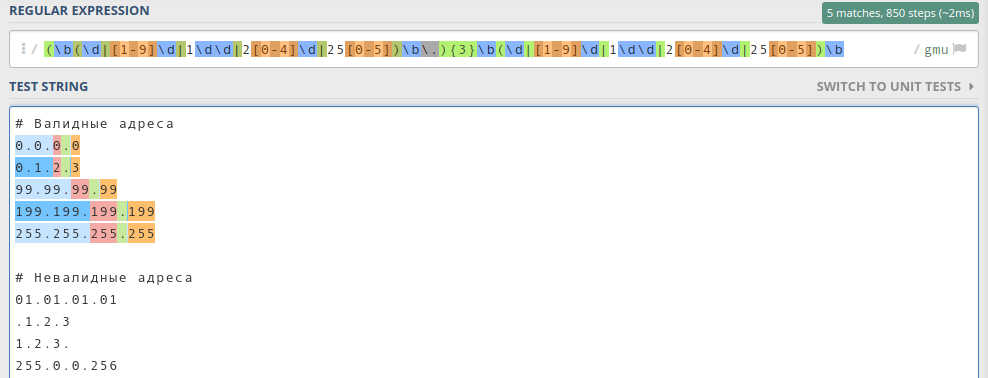


Как видим, все байты стали зелёненькими. Это значит, что мы на верном пути.

Осталось дело за малым: сделать так, чтобы искать четыре байта, а не один. Нужно учесть, что байты разделены тремя точками. То есть мы ищем три байта с точкой на конце и один без точки:

(\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b\.){3}\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b

Результат выглядит так:



Подсветились только валидные ip-адреса, значит регулярное выражение работает корректно.

**Источники:**

<http://microsin.net/programming/pc/csharp-before-string.html> про @

<https://metanit.com/sharp/tutorial/7.4.php>

<https://anton-pribora.ru/articles/it/regular-expressions/?ysclid=lh67k9vu3k951346125>