Artigos

Ver página do autor

Expressões Regulares (Armamento Pesado)

Por: Renato Guimarães
[Entre em contato com o autor]

Bacharel em Sistemas de Informação e trabalha com tecnologia da informação há mais de 15 anos.

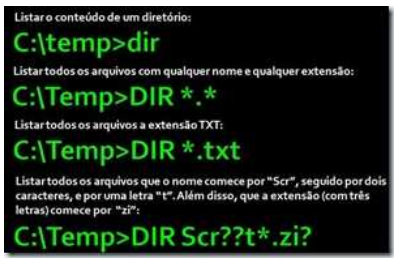
MVP Profile

Expressões Regulares (Armamento Pesado)

Publicado em: 22/09/2008

Hoje gostaria de falar sobre um assunto que gosto muito e que me encantei desde que aprendi na faculdade: Expressões Regulares. Só que, antes disso, deixe-me mostrar que há muito tempo já tivemos algum tipo de contato com este recurso, só que de uma forma mais simples e indiretamente. No post de hoje quero mostrar o poder das expressões regulares, ou RegEx. Isso dá assunto para livro de mais de 1000 páginas e no post de hoje pretendo mostrar os conceitos básicos. Sempre que estudo mais um pouco descubro que não conheço nada e que preciso praticar ainda mais e mais... Isso mesmo, tem que praticar muito e muito. Sempre que possível tento estudar e aprender mais um pouco...

Eu disse que você já conhece este conceito, certo? Então, lembra-se daqueles comandos que você executa no MS-DOS para listar os arquivos de um diretório, o DIR? Por exemplo, C:\Temp>DIR, onde lista todo conteúdo de um diretório. Só que, algumas vezes, você necessitou melhorar sua consulta e usou alguns caracteres específicos (ou coringas) para dizer o comando o padrão de nome que você está procurando, por exemplo: * e ?. Lembra? Então, neste momento você já identificou o padrão do que procura e vai definir uma string que represente este padrão, conforme exemplos abaixo:



Quando você usa estes comandos, basicamente, você identifica um padrão através de um conjunto de caracteres e, em seguida, usa a string como entrada para uma máquina especial que sabe como executar este padrão contra uma fonte de dados, que pode ser um texto, uma lista etc. Ao usar estes comandos no MS-DOS, você está limitado ao conjunto de caracteres deste ambiente. Com o que vou mostrar hoje, você ganhará uma super lista com vários caracteres para representar qualquer padrão que imaginar.

As expressões regulares oferecem um mecanismo poderoso, flexível e eficiente para processamento de texto. Através de uma notação extensa é possível analisar uma grande massa de dados a procura de padrões: para extração, editar e substituir textos de uma string. É essencial para aplicações que lidam com processamento de texto, tais como: processadores HTML, analisadores de Log, analisadores de cabeçalho HTTP, entre outros.

Vou falar sobre expressão regular com Visual C#, mas saiba que pode usá-la em qualquer linguagem que tenha suporte a tal. Por exemplo, você pode usar e abusar de expressão regular com JavaScript, para mais informações acesse o site Core JavaScript 1.5 Reference:Global Objects:RegExp. Para expressões regulares com VBScript, acesse o site Microsoft Beefs Up VBScript with Regular Expressions. O site Regular-Expressions.Info também contém muito material interessante.

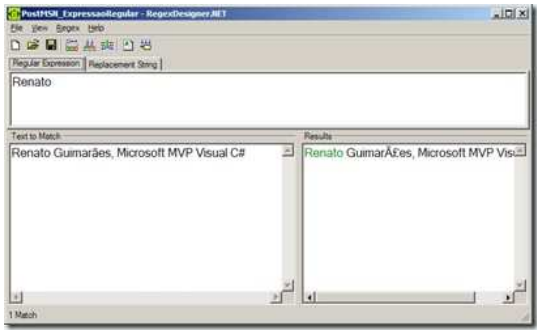
Uma ferramenta para design e teste da Expressão Regular

Antes de você começar a construir um código que fará uso de uma expressão regular, acredito que o mais importante é que você faça uma análise do padrão que deseja capturar e, em seguida, faça o design da sua expressão. Assim que tiver sua expressão pronta, teste-a para garantir que todas as entradas sejam capturadas completamente. Então, para projetar e testar sua expressão, nada melhor do que utilizar o RegExDesigner.NET do SellsBrothers. Com ela você pode testar expressões para combinação, substituição e

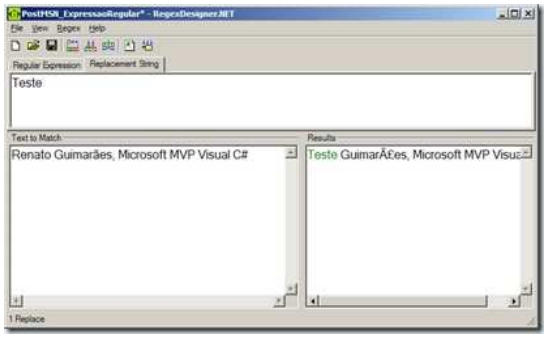
Java
Javascript
LSL (Second Life)
Modelagem
PHP
Python
Sharepoint
Silverlight
SQL
VBA (Office)
Visual Basic
Visual Basic .NET
Visual Fox Pro
WCF/WPF
Web Services
XML
:: Infra
BizTalk Server
CRM
Exchange Server
ForeFront / Antigen / IAG
Interoperabilidade
ISA Server
Linux
MOF
MS Dynamics CRM
Network
OCS / LCS
Outlook
Powershell e Scripts
Redes
Segurança
System Center e Gerenciamento
Virtualização
Windows
Windows Server
:: Banco de Dados
Access
Caché
Firebird
Interbase
MySQL
Oracle
SQL Server
Sybase
:: Gerência
Arquitetura
Ciclo de Vida de Desenvolvimento
Controle de Versão
Estimativas
Metodologias
MOF
Qualidade e Testes

preenchido nos campos da tela. Ou seja, facilmente você já tem o código na sua linguagem favorita pronto para inserir no seu programa. O legal desta ferramenta é que você pode pensar na expressão abstraindo os aspectos da linguagem, mantendo o foco só na expressão. Você também pode testar suas expressões na ferramenta online do RegExLib.Com.

Veja a figura abaixo, um exemplo da IDE do RegExDesigner, onde eu tenho uma expressão "Renato" que desejo validar contra a string "Renato Guimarães, Microsoft MVP Visual C#". Mas a palavra "Renato" é uma expressão? Sim. Ou seja, você está dizendo que o padrão procurado é "Renato", só que este exemplo não é nada frente ao poder dos meta-caracteres das expressões regulares. Clique sobre o botao "Match Text" para ferramenta executar a expressão. Caso queira testar a substituição, clique em "Replace Text".



Perceba do lado direito onde a ferramenta mostra na cor verde parte da string que combinou com o padrão da expressão. Outro exemplo de uso da ferramenta, caso queira testar uma expressão para substituição, só informar o texto na aba "Replacement String". No caso da substituição, a string resultante será substuída em todas as partes onde o padrão combinou, o que no exemplo só aconteceu na primeira palavra do texto de entrada.



Um outro recurso interessante desta ferramenta é a geração de código com base no que foi informado na interface gráfica. Por exemplo, veja o código C# gerado pela ferramenta para o exemplo anterior:

```
using System.Text.RegularExpressions;


// Regex Match code for C#
void MatchRegex()
{
   // Regex match
   RegexOptions   options = RegexOptions.None;
   Regex          regex = new Regex(@"Renato", options);
   string         input = @"Renato Guimarães, Microsoft MVP Visual C#";

   // Check for match
   bool   isMatch = regex.IsMatch(input);
   if( isMatch )
   {
      // TODO: Do something with result
      System.Windows.Forms.MessageBox.Show(input, "IsMatch");
   }

   // Get match
   Match   match = regex.Match(input);
   if( match != null )
   {
      // TODO: Do something with result
      System.Windows.Forms.MessageBox.Show(match.Value, "Match");
   }

   // Get matches
   MatchCollection   matches = regex.Matches(input);
   for( int i = 0; i != matches.Count; ++i )
   {
      // TODO: Do something with result
      System.Windows.Forms.MessageBox.Show(matches[i].Value, "Match");
   }

   // Numbered groups
   for( int i = 0; i != match.Groups.Count; ++i )
   {
      Group   group = match.Groups[i];

      // TODO: Do something with result
      System.Windows.Forms.MessageBox.Show(group.Value, "Group: " + i);
```

LC Blog
(Onde você faz a notícia)

Os 10+ | Autores do dia
Israel Aéce
Anderson Patricio
Luiz Felipe de Freitas
Júlio Cesar Fabris Battisti
Alfred Reinold Baudisch
Ramon Durães
Alessandro de Oliveira Faria
Marcio Silveira Franco e Silva
Wellington Balbo de Camargo
Robert Martim

Os 10+ | Artigos do dia
HTML Básico
HTML Avançado
Criando aplicativos para o Orkut
Instalando Apache + MySQL + PHP 5 no Windows
Engenharia de Componentes - Parte 3
Tutorial de Tabelas Dinâmicas no Excel – Parte 1
PL/SQL - Procedures e Funções
Aplicando maquiagem facial no Adobe Photoshop Cs 2®
Comandos básicos em SQL - insert, update, delete e select
Tutorial: Desenhando com o Corel Draw

```
    // Named groups
    string   groupA = match.Groups["groupA"].Value;
    string   groupB = match.Groups["groupB"].Value;

    // TODO: Do something with result
    System.Windows.Forms.MessageBox.Show(groupA, "Group: groupA");
    System.Windows.Forms.MessageBox.Show(groupB, "Group: groupB");
}
```

Como está o suporte do .NET a expressão regular?

O .NET Framework incorpora as características mais populares dos principais mecanismos de expressão regular: Perl e awk. Foi projetado para ser compatível com o mecanismo de expressão regular do Perl 5.0. Além disso, também implementa características ainda não vista em outras implementações, tais como combinação right-to-left e compilação on-the-fly. O mecanismo de RegExp do .NET faz parte das classes básicas e pode ser usado por qualquer linguagem ou ferramenta que suporte o CLR, inclusive ASP.NET e Visual Studio .NET. As classes estão no namespace **System.Text.RegularExpressions**, tendo como principais classes **Regex, Match, MatchCollection, GroupCollection, CaptureCollection, Group e Capture**. Você pode criar uma instância desta classe ou usar algum dos seus métodos estáticos, por exemplo, o método **Match()**.

```
using System;

using System.Text.RegularExpressions;

namespace PostsMSN.Samples.RegularExpression
{
   class Program{
      static void Main(string[] args){
         //String usada como entrada
         string input = "Renato Guimarães";

         //Executa o Match para procurar a combinação com base no padrão
         Match match = Regex.Match(input, "Renato");

         //Se o método Match foi bem sucedido, retorna a posição e o texto encontrado
         if (Regex.Match(input, "Renato").Success){
            Console.WriteLine("Posicão: " + match.Index + " Texto: " + match.Value);
         }

         //OBS: Pode-se usar o método Match, visto que o padrão pode ser encontrado
         //mais de uma vez na string de entrada.

         //Resultado => Posicão: 0 Texto: Renato
      }

   }
}
```

O Visual Studio .NET possui uma documentação excelente sobre Expressões Regulares, tão bom quanto um livro. Por exemplo, caso quira saber os detalhes do funcionamento do mecanismo no .NET, procure no help do VS.NET por "Details of Regular Expression Behavior". Recomendo também a leitura do tópico "Compilation and Reuse", também do tutorial do VS.NET. Por exemplo, alerta que o uso da opção RegexOptions.Compiled deve ser feita com cautela, pois os recursos usados para geração de código para melhorar performance não são liberados quando a instância de Regex criada for removida da memória, por exemplo.

Quais são os caracters que posso usar numa expressão?

Já falei demais e ainda não mostrei os operadores, caracteres e construres que podem ser usados na definição de uma expressão regular. Sendo assim, antes de vermos qualquer outro exemplo, vamos conhecer as tabelas dos caracteres de escape, classes de caracteres, quantificadores, agrupadores, referência e alternação. Antes disso, saiba que qualquer outro caractere que não seja um destes (. $ ^ { [(|) * + ? \ ), são considerados como o próprio caractere. Por exemplo, no começo do post coloquei um exemplo com a expressão regular "Renato", ou seja, cada caractere representa o seu próprio significado.

Caracteres de Escape: Caracteres que têm significado especial quando são precedidos pelo caractere "\".

Caractere	Descrição
\a	*Mapeia o caractere \u0007 (alarme)*
\b	*Mapeia um backspace se estiver entre []*
\t	*Mapeia um tab \u0009*
\r	*Mapeia um retorno de carro\u000D*
\v	*Mapeia um tab vertical \u000B*
\f	*Mapeia uma alimentação de papel (form feed) \u000C*
\n	*Mapeia uma nova linha \u000A*
\e	*Mapeia um esc \u001B*
\040	*Mapeia um caractere ASCII como Octal*
\x20	*Mapeia um caractere ASCII usando representação hexadecimal*
\cC	*Mapeia um caractere de controle ASCII, por exemplo, \cC é Control-C*
\u0020	*Mapeia um caractere Unicode usando representação hexadecimal*
\	*Quando não é seguido por um caractere de escape, mapeia o próprio caractere, por exemplo, *.*

Classes de Caracteres: Para simplificar o post, esta tabela foi resumida para ilustrar somente as classes usadas nos exemplos. Para mais detalhes, consultar documentação do Visual Studio .NET. Na tabela abaixo perceba que uma classe em letra minúscula tem seu inverso usando a mesma classe em letra maiúscula. Além disso, perceba que o caractere "^" é usado para negar o conteúdo de um grupo.

Caractere	Descrição
[grupo]	Mapeia qualquer caractere especificado no grupo, por exemplo, [aeiou].
[^ grupo]	Mapeia qualquer caractere que não esteja especificado no grupo, ou seja, negação do grupo, por exemplo, [^aeiou]
[primeiro-ultimo]	Mapeia qualquer caractere no intervalo, por exemplo, [A-Z a-z 0-9]
\w	Mapeia qualquer letra ou número, equivalente a [a-zA-Z0-9]
\W	Mapeia qualquer caractere que não seja letra, equivalente a [^a-zA-Z0-9]
\s	Mapeia qualquer caractere que seja espaço em branco, equivalente a [\f \n \r \t \v]
\S	Mapeia qualquer caractere que não seja espaço em branco, equivalente a [^ \f \n \r \t \v]
\d	Mapeia qualquer caractere que seja um dígito, equivalente a [0-9]
\D	Mapeia qualquer caractere que não seja um dígito, equivalente a [^0-9]
.	Mapeia qualquer caractere, com exceção do \n, mas caso a opção SingleLine seja usada, mapeia todos os caracteres, sem exceção.

Afirmações de posição: Caracteres que indicam se a validação foi bem sucedida, ou não, dependendo da posição corrente da string. Perceba que o caractere "^", neste caso tem um significado diferente da tabela acima (onde ele deve acontecer entre []).

Afirmação	Descrição
^	Indica que a combinação deve ocorrer no início da string ou no início da linha.
$	Indica que a combinação deve ocorrer no fim da string, antes de um \n no fim da string, ou no fim da linha
\A	Indica que a combinação deve ocorrer no início da string, ignora a opção Multline
\Z	Indica que a combinação deve ocorrer no fim da string ou antes de um \n no fim da string, ignora a opção Multine.
\z	Indica que a combinação deve ocorrer no fim da string, ignora a opção Multline
\G	Indica que a combinação deve ocorrer no ponto onde a combinação anterior terminou. Quando usado com Match.NextMatch(), assegura que todas as combinações são adjacentes.
\b	Indica que a combinação deve ocorrer numa fronteira entre um \w (alfanumérico) e \W (não alfanumérico). Quando ocorre numa fronteira \w, quer dizer que o primeiro e último caracteres devem ser um \W (não alfanumérico).
\B	Indica que a combinação não ocorre numa fronteira \b

Quantificadores: São usados para indicar a quantidade de vezes que um padrão deve acontecer. Podem ser aplicados a um caractere, um grupo ou a uma classe de caracteres. Na tabela abaixo, n e m são números inteiros.

Quantificador	Descrição
*	Indica zero ou mais combinações. Por exemplo, \w* ou (abc)*. Equivalente a {0,}
+	Indica uma ou mais combinações. Por exemplo, \w+ ou (abc)+. Equivalente a {1,}
?	Indica zero ou uma combinação. Por exemplo, \w? ou (abc)?. Equivalente a {0,1}
{n}	Indica o número de combinações, por exemplo, [casa]{2}, onde um dos caracteres "c", "a" e "s" devem aparecer duas vezes.
{n,}	Indica que deve acontecer pelo menos n combinações, por exemplo, (abc){2,}
{n,m}	Indica que deve acontecer pelo menos n combinações, não mais do que m.(abc){2,4}
*?	Indica que a primeira combinação deve consumir o menor número de repetições possíveis
+?	Indica menos combinações possíveis, tendo pelo menos uma
??	Indica uma ou mais repetições.
{n}?	Equivalente a {n}
{n,}?	Indica menos combinações possíveis, pelo menos n
{n,m}?	Indica menos combinações possíveis, entre n e m

Agrupadores: São usados para definir sub-expressões de uma expressão regular e capturar substrings da string de entrada.

Agrupador	Descrição
(subexpressão)	Captura a subexpressão e são numeradas automaticamente com base na ordem do caractere "(", iniciando em 1. A primeira captura, o elemento zero, é o texto
(?<nome> subexpressão)	Captura uma subexpressão dentro de um nome ou número de grupo. A string usada para o nome não deve ter pontuação e não pode começar por número. Pode-se usar apostrófos no lugar dos caracteres "<" e ">".
(?<nome1-nome2>subexpressão)	(Balanceamento da definição do grupo). Exclui a definição do grupo nome2, definido anteriormente, e armazena no grupo nome1 o intervalo entre o grupo nome2 e o grupo corrente. Como a exclusão da última definição do nome2 revela a definição anterior de nome2, este construtor permite que a pilha de capturas para o grupo nome2 seja usada como contador para manter o rastro dos construtores aninhados, tal como parênteses. Neste construtor, o nome1 é opcional. Por exemplo, você deseja validar se uma expressão matemática está correta no que
(?:subexpressão)	Não captura a substring combinada pela subexpressão.
(?imnsx-imnsx:subexpressão)	Aplica ou desabilita opções específicas na sub expressão. Por exemplo, (?i-s:) liga o case insensitive e desabilita o mode de única linha.
(?=subexpressão)	Continua a combinação somente se a subexpressão combina na posição da direita. Por exemplo, \w+(?=\d) combina uma palavra seguida por um dígito, sem combinar
(?!subexpressão)	Continua a combinação somente se a subexpressão não combina na posição da direita. Por exemplo, \b(?!un)\w+\b combina palavras que não comemça com "un".
(?<=subexpressão)	Continua a combinação somente se a subexpressão combina na posição da esquerda. Por exemplo, (?<=19)99 combina instâncias de 99 precedidas por 19.
(?<!subexpressão)	Continua a combinação somente se a subexpressão não combina na posição da
(?>subexpressão)	A subexpressão é combinada por completo somente uma vez, e depois não participa do retorno de trilha gradativo, ou seja, a subexpressão combina somente strings que combinariam com a subexpressão sozinha. Por padrão, se a combinação não for bem sucedida, o retorno de trilha procura por outras combinações possíveis.

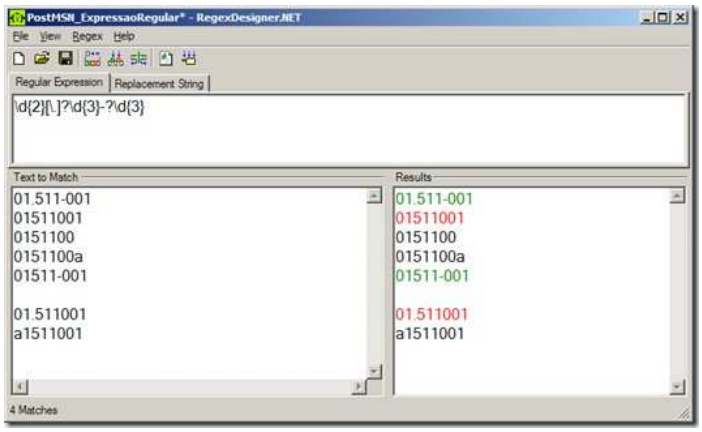
Estes agrupadores podem ser nomeados ou, caso não receba um nome específico, receberão um nome padrão. Além disso, são numerados sequencialmente, com base na ordem de abertura dos parênteses da esquerda para direita, porém a numeração dos agrupamentos nomeados só se inicia após os grupamentos não nomeados. Por exemplo, na expressão ((?<One>abc)\d+)?(?<Two>xyz)(.*) os agrupamentos são capturados por nome e número. O agrupamento zero (a primeira captura) é sempre para expressão toda (exemplo do Visual Studio).

- **Número:** 0 **Nome:** default **Padrão:** ((?<One>abc)\d+)?(?<Two>xyz)(.*)

- **Número:** 2 **Nome:** default **Padrão:** (.*)
- **Número:** 3 **Nome:** One **Padrão:** (?<One>abc)
- **Número:** 4 **Nome:** Two **Padrão:** (?<Two>xyz)

Vamos praticar alguns exemplos com as tabelas acima

Após elaborar uma expressão, é importante que você defina uma massa de dados para testar os casos onde a expressão combina ou não o padrão. Além disso, é importante lembrar que não existe somente uma forma de definir uma expressão, por exemplo, não existe só uma forma possível de montar uma expressão que combine um número de telefone, pois tudo dependerá do padrão interessado. Outra coisa importante, expressão regular é igual a construção de um trecho de código, você pode ter n formas de escrever, porém cada uma tem sua lógica. Em resumo, quero dizer que não há só uma expressão correta, pois elas só diferem no número de padrões que podem capturar. Na figura abaixo, veja que montei uma expressão para capturar uma string no formato de CEP e, logo na caixa abaixo, inseri alguns exemplos para validação da expressão. O cep pode conter, ou não, o "." e o "-" para definição da máscara. Do lado direito da figura em vermelho e verde estão as strings que combinaram com o padrão da expressão.



Para ilustrar mais exemplos de expressões, criei uma classe com alguns métodos para validação de formato, veja código abaixo. No caso do Visual C#, como o caractere "\" tem significa especial, indicação de espace, ou você acrescenta uma outra "\" ou adiciona um "@" (Verbatim String) antes da string.

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

namespace PostsMSN.Samples.RegularExpression{
   /// <summary>

   /// Exemplos de Expressões Regulares para validação de dados
   /// </summary>
   public class StringUtil{

      /// <summary>
      /// Verifica se um número de telefone está no formato válido,
      /// inclusive código do país e da cidade.
      /// </summary>

      /// <param name="input">Número do telefone</param>
      /// <returns>Verdadeiro se o número estiver no formato válido</returns>
      public static bool IsTelefoneValido(string input) {
         string pattern = @"^\+?\d{2,3}?\s*\(\d{2}\)\s*\d{4}-\d{4}$";
         return Regex.IsMatch(input, pattern);
      }

      /// <summary>

      /// Verifica se o CEP está no formato correto.
      /// </summary>
      /// <param name="input">Número do cep</param>
      /// <returns>Verdadeiro se estiver no formato correto</returns>

      public static bool IsCepFormatoValido(string input){
         string pattern = @"^\d{2}[\.]?\d{3}-?\d{3}$";
         return Regex.IsMatch(input, pattern);
      }

      /// <summary>
      /// Verifica se o CPF está no formato correto, mas não testa se é válido.
      /// </summary>

      /// <param name="input">Número do cpf com a máscara</param>
      /// <returns>Verdadeiro se estiver no formato correto</returns>
      public static bool IsCpfFormatoValido(string input){
         string pattern = @"^\d{3}\.\d{3}\.\d{3}-\d{2}$";
         return Regex.IsMatch(input, pattern);
      }

      /// <summary>

      /// Verifica se a hora está no formato correto.
      /// </summary>
      /// <param name="input">Hora no formato HH:MM</param>
      /// <returns>Verdadeiro se estiver no formato correto</returns>
```

```
            return Regex.IsMatch(input, pattern);
        }

        /// <summary>
        /// Verifica se e-mail está no formato válido
        /// </summary>

        /// <param name="input">e-mail</param>
        /// <returns>Verdadeiro se estiver no formato correto</returns>
        public static bool IsEmailFormatoValido(string input){
            string pattern = @"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
                        {1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
            return Regex.IsMatch(input, pattern);
        }

        /// <summary>

        /// Verifica se o GUID está no formato válido
        /// </summary>
        /// <param name="input">GUID com ou sem chaves {GUID}</param>
        /// <returns>Verdadeiro se estiver no formato correto</returns>

        public static bool IsGuidFormatoValido(string input){
         string pattern = @"^[{|\(]?[0-9a-fA-F]{8}[-]?([0-9a-fA-F]{4}[-]?){3}[0-9a-fA-F]{12}[\)|}]?$";
         return Regex.IsMatch(input, pattern);
        }

        /// <summary>
        /// Remove os caracteres especiais de uma string
        /// </summary>

        /// <param name="input"></param>
        /// <returns></returns>
        public static string LimparString(string input){
            string pattern = @"[^\w\.@-]";
            return Regex.Replace(input, pattern, "");
        }
    }
}
```

O programa usado para testar as expressões da classe acima:

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

namespace PostsMSN.Samples.RegularExpression{
    class Program{

        static void Main(string[] args){
            //Testa o formato do telefone
            string input = "+55 (81) 3411-9337";
            Console.WriteLine("Teste Fone: " + StringUtil.IsTelefoneValido(input));

            //Testa o formato do CEP
            input = "01.522-002";
            Console.WriteLine("Teste Cep: " + StringUtil.IsCepFormatoValido(input));

            //Testa o formato do CPF
            input = "987.980.098-09";
            Console.WriteLine("Teste CPF: " + StringUtil.IsCpfFormatoValido(input));

            //Testa o formato da hora
            input = "23:59";
            Console.WriteLine("Teste Hora: " + StringUtil.IsHoraFormatoValido(input));

            //Testa o formato do e-mail
            input = "renato.guimaraes@gmail.com";
            Console.WriteLine("Teste Email: " + StringUtil.IsEmailFormatoValido(input));

            input = "{914D226A-2F5B-4944-934D-96BBE6571977}";
            Console.WriteLine("Teste GUID: " + StringUtil.IsGuidFormatoValido(input));

            input = "teste#$98#%";
            Console.WriteLine("Resultado Limpar: " + StringUtil.LimparString(input));
            Console.ReadLine();
        }

    }
}
```

Falando um pouco sobre uso deste poderoso mecanismo no dia-a-dia, outro dia precisava construir um programa que obtivesse um página HTML de um determinado endereço e, em seguida, extraísse todas as notícias. Para isso, identifiquei como estava o padrão do HTML e localizei a região de demarcava as notícias e montei uma string que recuperava a região do documento. Depois, montei uma outra expressão que recuperava o conteúdo entre um os elementros "" e "" onde o atributo class tivesse um valor específico (poderia ter matado tudo numa única expressão). Um outro caso interessante foi quando fiz um protótipo que faz consultas usando os serviços do Serasa, onde o resultado vem num determinado padrão, minha expressão capturava cada parte do retorno e, em seguida, eu processava o trecho conforme o layout. Para fechar, um outro exemplo, fiz um programa que analisava os arquivos de log do Sharepoint e retornava os erros de um determinado tipo, por exemplo, erros de workflow.

Abaixo segue mais dois exemplos interessantes (do Visual Studio .NET) que mostrar a utilização de expressões regulares com agrupamento.

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

namespace PostsMSN.Samples.RegularExpression{
   class Program{
      static void Main(string[] args){
         //Entrada para validacao da expressao que extrai o target="_blank" href
         String input = "<div>" +
                        "<a target="_blank" href=\"https://www.msdnbrasil.com.br/cadastro/default.aspx\">" +
                        "<img alt=\"Meu MSDN\" src=\"ms348103.my_msdn2_BR.jpg\" /></a>" +
                        "</div>";

         //No detalhe, a expressão significa que a string capturada deve:
         // a) iniciar por target="_blank" href (target="_blank" href);
         // b) ser seguida por um sinal de igual, e pode ter nenhum ou
         //    vários espaços entre eles (\\s*=);
         // c) depois do sinal de igual pode ter nenhum ou vários espaços (\\s*);
         // d) seguida pelo agrupamento (?:\"(?<1>[^\"]*)\"|(?<1>\\S+)), mas não
         //    será capturado
         // e) no agrupamento <1> pode ter nenhum ou vários caracteres que não
         //    seja um " ([^\"]*), seguido por um "
         // f) ou o grupo <1> pode ser um caracter que não seja um espaço(\\S+)
         Regex r = new Regex("target="_blank" href\\s*=\\s*(?:\"(?<1>[^\"]*)\"|(?<1>\\S+))");
         for (Match m = r.Match(input); m.Success; m = m.NextMatch()){
            Console.WriteLine("target="_blank" href encontrado: " + m.Groups[1] + " na posição "
               + m.Groups[1].Index);
         }

         //A expressão abaixo usa o recurso de agrupamento com balanceamento para
         // validar se uma expressão matématica contém um ")" para cada "(".
         // Perceba que a expressão não valida se só tem números e
         // os operadores possíveis.
         string pattern = @"^[^\(\)]*(((?'Abre'\()[^\(\)]*)+((?'Fecha-Abre'\))"+
                          @"[^\(\)]*)+)*(?(Abre)(?!))$";

         input = "(10 + 10) * (10-5)";
         //input = "(10 + 10) * (10-5))"; Falha porque sobra um ")"

         Match match = Regex.Match(input, pattern);
         if (match.Success == true)
            Console.WriteLine("Entrada: \"{0}\" \nCombinação: \"{1}\"",
               input, match);
         else
            Console.WriteLine("Combinação Falhou.");

         Console.ReadLine();

         //Resultado:
         //   target="_blank" href encontrado: https://www.msdnbrasil.com.br/cadastro/default.aspx
         //                na posição 14
         //   Entrada: "(10 + 10) * (10-5)"
         //   Combinação: "(10 + 10) * (10-5)"

      }
   }
}
```

Fico por aqui e minha recomendação é que você tente aplicar este recurso sempre que possível, pois facilita e reduz a quantidade de código para rotinas complexas. Lógico, o que falei aqui não é nada frente ao mundo de coisas que pode se escrever sobre expressões regulares, pois existe muita teoria (Autômatos Determinísticos, Autômatos Não-Determinísticos, Máquina de Turing, Linguagens Regulares, entre outros) por trás destes mecanismos. Pode-se dizer a construção de uma expressão é uma forma de programar e o limite está a cargo da sua imaginação. Cada vez que tento entender algum exemplo avançado que pego pela Internet fico admirado com o que é possível fazer... Uma boa forma de aprender é analisar exemplos e tentar entender o que faz cada coisa da expressão do exemplo. De primeira vez não é fácil montar uma expressão mas com alguma prática fica fácil entender como funciona e depois é só praticar, praticar e praticar... Assim você virará um mestre no assunto(eu não sou porque não pratico tanto).

Referências

- **.NET Framework Regular Expressions (documentação Visual Studio .NET)**
- **Regular-Expressions.Info**
- **Exemplos de Expressões Regulares no Regular-Expressions.Info**
- **RegExLib.com**
- **Exemplos de Expressões Regulares no RegexLib.com**

Abraço,

Renato Guimarães, MS MVP C#

Inclua um comentário sobre o artigo

Topo

Elogios e críticas são muito bem vindos, porém o comentário deve ter referência ao artigo em pauta.
O portal e o autor agradecem.

Nome:

E-mail:

Comentários:

Digite a palavra abaixo:

Para dúvidas técnicas, **NÃO UTILIZE ESTE ESPAÇO**, utilize nosso fórum de discussão.
http://linhadecodigo.com.br/cs2/forum

Enviar comentário

Comentários sobre o artigo

Ver Todos comentários

Ainda não existem comentários sobre este artigo. Seja o(a) primeiro(a)!

Produtos relacionados

Topo

Ainda não existem produtos relacionados.

Outros artigos do autor

Topo

Vídeo: Passagem de Parâmetros por Valor e por Referência com C#

Vídeo: Herança com Visual C#

ADO.NET 2.0: A importância do Pool de Conexões

Struct e Class: Quando usar?

Você tem componentes COM e quer aproveitá-los em .NET?

Manipulando processos através da classe System.Diagnostics.Process

Consultando e invocando métodos dincamicamente usando Reflection

Células Acadêmicas .NET! Uma forma rápida e econômica de se manter atualizado

Como criar um projeto no Visual Studio.NET dentro de um diretório diferente do inetpub/wwwroot

ASP.NET – Uma mudança radical no desenvolvimento web... Sua vida vai mudar!

ASP.NET – Veja como manipular JScript e obter algumas facilidades com a classe Page

ASP.NET - Como transformar um Web Form em um User Control (continuação)

ASP.NET - Como transformar um Web Form em um User Control

Exceções - Uma visão geral

Visual Studio.NET – Agora temos uma verdadeira forma de depurar scripts

ASP.NET - Trabalhando com o modelo de programação Code in Page usando WebMatrix

Artigos relacionados

Topo

Monitorando Arquivos e Diretórios com FileSystemWatcher

Trabalhando com Linq To XML parte 2: Linq vS Lambda Expression

Criando arquivo XML com Linq To XML

Trabalhando com cache em propriedades

Concatenação de Strings

Trabalhando com CLR: Stored Procedure – Segundo Passo

Componente TreeView C#

Criando um Windows Service

Criptografando dados com C#

Monitorando o consumo de memória e tempo de execução

Salvando em XML

Melhorando WinApps seguindo WebApps

Cadastro de um Consultório em Windows Forms, com C# e SQL Server – Parte 12

Criar arquivo de integração com Integration Services

Trabalhando com Array Params no C#

Pegando o horário inicial e final do uso do Windows usando C#

Como funcionam as Arrays (Matrizes) Multidimensionais no C#?

Cadastro de um Consultório em Windows Forms, com C# e SQL Server – Parte 11

Documentação de Código .Net

ASP.NET MVC Custom Helpers

Eval em C# com IronRuby

Trabalhando com CLR: Stored Procedure – Primeiro passo

Entendendo interfaces com C#

Executar aplicações em background com Agendador de Tarefas Windows

ExpandoObject: dinamismo no .NET 4.0

Integração C# + Ruby

Utilizando um Tipo T como parâmetro e recuperando seus valores

Detectando mudanças em objetos

URL Routing com o Visual Studio .NET 2010

Acessando dados com Textboxes e botões de navegação