Expresiones regulares

Este documento es una traducción (~~mas o menos buena~~) del documento escrito por

Richard Carr,

Que se encuentra publicado en esta dirección

http://www.blackwasp.co.uk/RegexGrouping.aspx

Esta (~~pésima~~) traducción la he hecho por interés propio, y para resolver un problema personal. No te quejes si no responde a tus expectativas, y acude al documento original.

En Zaragoza (España) a martes, 9 de enero de 2024 a las 16:50:15 horas

Joaquin Medina Serrano

mailto:gmjms32@gmail.com

(Estándar Dublin Core [http://dublincore.org])

Contenido del documento

- dc.title: Expresiones regulares
- **dc.description**
  - Este documento es una traducción de un tutorial en Ingles
  - Tutorial en el que se describe el uso de expresiones regulares con .NET Framework y C#. Las expresiones regulares definen las reglas para la coincidencia de patrones en las cadenas. Son útiles para buscar y reemplazar texto, y para validar el formato de las cadenas.
- **dc.type:** Text
- **dc.source.bibliographicCitation:**
  - http://www.blackwasp.co.uk/RegexGrouping.aspx
- **dc.relation.ispartof:** Mis Apuntes tácticos
- **dc.identifier:** http://www.blackwasp.co.uk/RegexGrouping.aspx
- **dcterms.isVersionOf:** http://www.blackwasp.co.uk/RegexGrouping.aspx
- **dcterms.isFormatOf:** http://www.blackwasp.co.uk/RegexGrouping.aspx
- **dc.coverage:** Expresiones regulares

Propiedad Intelectual

- **dc.creator:** Richard Carr,
- **dc.publisher** Medina Serrano, Joaquín.
- **dc.rights:** Richard Carr,
- **dc.rights.accessrights:** https://www.blackwasp.co.uk/Copyright.aspx

Información sobre el documento

- **dc.date.created:** 2012-08-28 (Fecha Creación)
- **dc.date.modified:** 2024-01-09 (Fecha Última Modificación)
- **dc:format:** text/PDF (Documento pdf)
- **dc.identifier**
  - http://joaquin.medina.name/web2008/documentos/informatica/lenguajes/puntoNET/System/Text/Regex/C_Sharp_ExpresionesRegulares.html
- **dc:language:** es-ES (Español, España)

Contenido

1 Expresiones regulares (Por Richard Carr)

Por Richard Carr, publicado en http://www.blackwasp.co.uk/RegularExpressions.aspx

Este es el primer artículo de una serie en la que se describe el uso de expresiones regulares con .NET Framework y C#. Las expresiones regulares definen las reglas para la coincidencia de patrones en las cadenas. Son útiles para buscar y reemplazar texto, y para validar el formato de las cadenas.

¿Qué son las expresiones regulares?

Dos problemas comunes que puede tener que resolver al procesar cadenas son validar que un elemento coincide con un patrón determinado y encontrar subcadenas que coincidan con un patrón y extraerlas de una cadena o documento de texto más grande. Por ejemplo, es posible que desee validar que una dirección de correo electrónico, URL, dirección IP o número de teléfono tenga el formato correcto. También es posible que desee extraer dicha información de un texto grande.

Las expresiones regulares proporcionan un lenguaje estandarizado para definir patrones. Utilizan una cadena formada por caracteres alfanuméricos y símbolos. A continuación, utilice un motor de expresiones regulares para comparar la expresión regular con otro fragmento de texto. Dependiendo del motor que utilice, puede determinar si hay una coincidencia, extraer una o coincidencias del texto de origen o incluso reemplazar las coincidencias. Al reemplazarlo, el nuevo texto puede incluir elementos de las coincidencias originales.

Los motores de expresiones regulares son muy potentes. Sin embargo, las expresiones regulares en sí mismas pueden volverse rápidamente muy complejas y, por lo tanto, difíciles de leer. Si nunca has usado expresiones regulares, encontrar una en el código de otro programador puede ser confuso. Debido a la complejidad, también pueden ser lentos de procesar. Por lo tanto, deben usarse solo cuando no estén disponibles opciones más simples y rápidas.

.NET Framework incluye un motor de expresiones regulares basado principalmente en la clase Regex, pero compatible con otros tipos definidos en el espacio de nombres System.Text.RegularExpressions. En este tutorial, exploraremos tanto la sintaxis que puede usar para crear expresiones regulares como las características del motor .NET.

Antes de profundizar en las expresiones regulares, veamos un ejemplo sencillo. En el archivo de clase generado automáticamente dentro de un nuevo proyecto de aplicación de consola, agregue una directiva using para el espacio de nombres:

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

Agregue el siguiente código dentro del método Main del programa. El código comienza creando una cadena de origen que contiene los detalles de algunos servidores de una red. La llamada a Regex.Matches usa una expresión regular para buscar cada fragmento de texto de la cadena de origen que se parezca a una dirección IP.

```
string source = @"
    Intranet Servers:    192.168.0.10/192.168.0.11/192.168.0.12
    SQL Server:          192.168.0.20
    Integration Servers: 192.168.0.30 (MSMQ)
                         192.168.0.40 (Services)";

var ips = Regex.Matches(source, @"\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}");

foreach (var ip in ips)
{
    Console.WriteLine(ip);
}

/* Output

192.168.0.10
192.168.0.11
192.168.0.12
192.168.0.20
192.168.0.30
192.168.0.40

*/
```

En el ejemplo, la expresión regular es algo ingenua. En realidad, busca texto que esté formado por cuatro números, cada uno de entre uno y tres dígitos de longitud, separados por puntos o puntos. Esto significa que encontraría direcciones IP no válidas, como 999.999.999.999. Es posible crear una expresión regular más compleja que solo coincida con direcciones IP válidas utilizando las herramientas y la sintaxis que veremos en futuros artículos de este tutorial.

1.1 Coincidencia de literales

Las expresiones regulares son ideales para validar que una cadena coincide con un patrón determinado o para encontrar las partes de una cadena que coinciden con el patrón. En casi todos los casos, el patrón incluye caracteres de control, lo que significa que las coincidencias localizadas no serán idénticas al propio patrón.

Es inusual usar expresiones regulares para encontrar cadenas literales. Sin embargo, en algunos casos es posible que tenga que hacerlo, y la búsqueda de literales es una forma útil de demostrar los métodos de coincidencia proporcionados por el motor de expresiones regulares de .NET. En este artículo veremos tres métodos de coincidencia de claves de la clase Regex, buscando literales en todos los casos.

1.1.1 IsMatch (Método)

El método IsMatch es la opción de coincidencia más básica y es ideal para la validación de la entrada del usuario. Compara una cadena de entrada con un patrón de expresión regular y devuelve true si la cadena de entrada contiene una coincidencia y false en caso contrario. Se puede llamar a IsMatch como un método estático o un método de instancia. En este artículo solo veremos la opción estática.

El método básico IsMatch tiene dos parámetros. El primero recibe una cadena que contiene el texto que se va a buscar. La segunda es una cadena que contiene el patrón. Como estamos usando cadenas literales para la expresión regular, el siguiente código simplemente detecta si la cadena de entrada contiene el texto del segundo argumento.

```
        Dim input = "El veloz zorro marrón salta sobre el perro."
        Dim isMatch1 As Boolean = Regex.IsMatch(input, "marrón")  ' true
        Dim isMatch2 As Boolean = Regex.IsMatch(input, "negro")  ' false
```

1.1.2 Método de coincidencia

El método Match se usa de forma similar a IsMatch. Sin embargo, en lugar de devolver un valor booleano, produce un objeto Match que contiene detalles sobre el texto coincidente. Esto incluye el texto real encontrado, que generalmente difiere del patrón, el índice inicial del texto coincidente dentro de la cadena de entrada y la longitud del texto coincidente.

Por ejemplo, el código siguiente coincide con el texto literal, "brown", dentro de la oración de la cadena de entrada. La llamada a Console.WriteLine muestra el índice de la coincidencia, el texto encontrado y su longitud.

```
    Dim input = "El veloz zorro marrón salta sobre el perro."
    Dim match As Match = Regex.Match(input, "marrón")
    Console.WriteLine("{0}: '{1}' {2} Caracteres ",
                      match.Index, match.Value, match.Length)
     '
    ' OUTPUT
    ' 15: 'marrón' 6 Caracteres
    '
```

Si no se encuentra ningún texto coincidente, el valor devuelto coincide con la propiedad estática, Match.Empty. La propiedad Success de este objeto se establece en false, como se muestra a continuación:

```
   Dim input = "El veloz zorro marrón salta sobre el perro."
   Dim operacionRedEx As Match = Regex.Match(input, "NEGRO")
   Dim matched As Boolean = operacionRedEx.Success    ' false
   Dim notAMatch = operacionRedEx Is Match.Empty      ' true
```

Por último, el objeto Match incluye un método denominado NextMatch. Esto le permite encontrar el siguiente fragmento de texto que coincida con la expresión regular. Al igual que antes, devuelve un objeto Match. Es Match.Empty una vez que se han agotado los fósforos.

En el ejemplo siguiente se buscan todas las instancias del texto, "brown", dentro de la cadena de entrada.

```
string input = "The quick brown fox jumps over the brown dog.";

Match match = Regex.Match(input, "brown");
while (match.Success)
{
    Console.WriteLine("{0}: '{1}' {2} chars",
                      match.Index, match.Value, match.Length);
    match = match.NextMatch();
}

/* OUTPUT
10: 'brown' 5 chars
35: 'brown' 5 chars
*/
```

```
Dim input = "El veloz zorro marrón salta sobre el perro marrón."
Dim buscaMatch As Match = Regex.Match(input, "marrón")
While buscaMatch.Success
      Debug.WriteLine(" Posición {0}: '{1}' {2} Caracteres",
                      buscaMatch.Index, buscaMatch.Value, buscaMatch.Length)
     buscaMatch = buscaMatch.NextMatch()
End While
' OUTPUT
'
' Posición 15 'marrón' 6 Caracteres
' Posición 43 'marrón' 6 Caracteres
'
```

1.1.3 Método de coincidencias

Si necesita buscar todos los elementos de la cadena de entrada que se ajusten al patrón definido por una expresión regular, puede llamar al método Matches. Esto procesa toda la cadena y devuelve una colección de objetos Match en una instancia de MatchCollection.

El código siguiente tiene una funcionalidad similar a la del ejemplo anterior. A medida que la colección se rellena en una operación, el ejemplo también genera el número de coincidencias, mediante la propiedad Count de la colección.

```
string input = "The quick brown fox jumps over the brown dog.";

MatchCollection matches = Regex.Matches(input, "brown");
Console.WriteLine("{0} match(es)", matches.Count);
foreach (Match match in matches)
{
    Console.WriteLine("{0}: '{1}' {2} chars",
                      match.Index, match.Value, match.Length);
}

/* OUTPUT

2 match(es)
10: 'brown' 5 chars
35: 'brown' 5 chars
*/
```

```
    Public Shared Function Prueba() As String

        Dim texto As String
        Using sw As New System.IO.StringWriter

            Dim input = "El veloz zorro marrón salta sobre el perro marrón."

            Dim encontradosColeccion As MatchCollection = _
                                Regex.Matches(input, "marrón")
            sw.WriteLine("La colección tiene [{0}] elemento(s)",
                         encontradosColeccion.Count)

            For Each elementoEncontrado As Match In encontradosColeccion
                sw.WriteLine(" Posición {0}: '{1}' {2} Caracteres",
                             elementoEncontrado.Index,
                             elementoEncontrado.Value,
                             elementoEncontrado.Length)
            Next

           sw.Flush()
           texto = sw.ToString
        End Using
        Return texto

        ' OUTPUT
        '
        'La colección tiene [2] elemento(s)
        'Posición 15 'marrón' 6 Caracteres
        'Posición 43 'marrón' 6 Caracteres
        '
    End Function
```

1.1.4 Caracteres especiales de escape

Como veremos en los artículos posteriores de este tutorial, las expresiones regulares pueden incluir muchos caracteres especiales que ayudan a definir el patrón que debe coincidir. Si desea encontrar uno de estos caracteres dentro de la cadena de entrada, debe escaparlo prefijando el carácter con una barra diagonal inversa (\).

Para demostrarlo, intente ejecutar el siguiente código. Esto parece estar buscando el texto "perro". Sin embargo, como el símbolo del signo de interrogación es un carácter especial en una expresión regular, solo las tres letras de "dog" coinciden realmente.

```
string input = "The quick brown fox jumps over the lazy dog?";

Match match = Regex.Match(input, "dog?");
Console.WriteLine("{0}: '{1}' {2} chars", match.Index, match.Value, match.Length);

/* OUTPUT

40: 'dog' 3 chars

*/
```

```
   Public Shared Function Prueba() As String

       Dim texto As String
       Using sw As New System.IO.StringWriter

         Dim input = "El rápido zorro marrón salta sobre el perezoso perro?"

         Dim match As Match = Regex.Match(input, "perro?")
         sw.WriteLine("{0}: '{1}' {2} chars",
                      match.Index,
                      match.Value,
                      match.Length)


        sw.Flush()
        texto = sw.ToString
     End Using
     Return texto

       ' OUTPUT
       ' 47: 'perro' 5 chars
       '
   End Function
```

Para incluir el signo de interrogación, puede escaparlo, como se muestra a continuación:

```
string input = "The quick brown fox jumps over the lazy dog?";

Match match = Regex.Match(input, @"dog\?");
Console.WriteLine("{0}: '{1}' {2} chars", match.Index, match.Value, match.Length);

/* OUTPUT

40: 'dog?' 4 chars

*/
```

```
Public Shared Function Prueba() As String

    Dim texto As String
    Using sw As New System.IO.StringWriter

        Dim input = "El rápido zorro marrón salta sobre el perezoso perro?"

        Dim match As Match = Regex.Match(input, "perro\?")
        sw.WriteLine("{0}: '{1}' {2} chars",
                     match.Index,
                     match.Value,
                     match.Length)
       sw.Flush()
       texto = sw.ToString
   End Using
   Return texto

   ' OUTPUT
   ' 47: 'perro?' 6 chars
   '
End Function
```

Nota: En el código anterior, la cadena de expresión regular está precedida por el carácter @. Esto especifica que el texto entre las comillas es un literal. Sin @, la barra diagonal inversa requeriría un escape, ya que es en sí mismo el carácter de escape para las cadenas de C#. Esto significa que el segundo argumento tendría que ser "perro\\?".

1.2 Caracteres de escape

En el artículo anterior de esta serie, examinamos los métodos básicos de coincidencia proporcionados por el motor de expresiones regulares de .NET. Vimos cómo hacer coincidir solo cadenas literales, en lugar de patrones que realizan búsquedas aproximadas. Al final del artículo, abordamos el uso del carácter de escape (\) y cómo se puede usar para prefijar el texto que desea hacer coincidir como literal pero que es un carácter especial en el lenguaje de expresiones regulares.

En este artículo, veremos los escapes de caracteres compatibles con el motor de expresiones regulares. Estos le permiten encontrar caracteres literales de diferentes maneras. Por ejemplo, puede hacer coincidir códigos de control, caracteres no imprimibles y cualquier símbolo ASCII o Unicode.

Todos los escapes de caracteres se definen en una expresión regular mediante una barra invertida (\) seguida de una o más letras, números o símbolos. Si se reconoce el símbolo después de la barra invertida, el motor busca el escape de caracteres deseado. Si no se trata de un escape de caracteres, el único carácter después de la barra diagonal inversa coincide con un carácter literal. Esto significa que puede hacer coincidir las barras invertidas con una barra diagonal inversa doble (\\), como se muestra a continuación:

```
string input = @"To match a \, you must use two backslashes (\\)";

foreach (Match match in Regex.Matches(input, @"\\"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT
Matched at index 11
Matched at index 44
Matched at index 45
*/
    Dim input = "To match a \, you must use two backslashes (\\)"

    For Each match As Match In Regex.Matches(input, "\\")
        Console.WriteLine("Matched at index {0}", match.Index)
    Next
```

1.2.1 Búsqueda de Tabuladores

Los archivos de texto pueden incluir caracteres de tabulación, que se utilizan para alinear el texto, especialmente en los archivos de texto que contienen listas. Puede utilizar escapes de

caracteres para buscar pestañas horizontales y verticales. Para tabulaciones horizontales, use el escape, \t. Hacer coincidir \v busca pestañas verticales.

En el ejemplo siguiente se encuentran las dos pestañas horizontales de la cadena de origen. Las tabulaciones se definen en la cadena mediante \t, ya que se usa la misma secuencia de escape en las cadenas de C# que en las expresiones regulares para las tabulaciones horizontales.

```
string input = "Tab\t\tTab";

foreach (Match match in Regex.Matches(input, @"\t"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT

Matched at index 3
Matched at index 4

*/
```

```
     Dim input = "Tabulador" & vbTab & vbTab & "Tabulador"

     For Each encontrado As Match In Regex.Matches(input, "\t")
         Console.WriteLine("Coincide en el índice {0}",
                           encontrado.Index)
     Next
 '
 ' OUTPUT
 ' Coincide en el índice 9
 ' Coincide en el índice 10
 '
```

1.2.2 Coincidencia de caracteres no imprimibles

Hay varios caracteres no imprimibles que se pueden hacer coincidir como literales mediante escapes de caracteres de expresión regular. Los tres elementos más utilizados son \r, que busca retornos de carro, \n, que busca un nuevo carácter de línea y \f, que busca alimentadores de formularios.

El siguiente código busca \r\n para encontrar el retorno de carro adyacente y el salto de línea que se inserta entre las dos palabras en el texto de origen.

```
string input =
@"Hello,
world";

foreach (Match match in Regex.Matches(input, @"\r\n"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT
Matched at index 6
*/
```

```
Dim input = "Hola" & Environment.NewLine & "mundo"
For Each match As Match In Regex.Matches(input, "\r\n")
    Console.WriteLine("Coincide en el índice {0}", match.Index)
Next
'
' OUTPUT
' Coincide en el índice 4
'
```

1.2.3 Coincidencia de caracteres de control

Además de los caracteres no imprimibles mencionados anteriormente, también puede hacer coincidir los caracteres de control no imprimibles. Esto puede ser especialmente útil cuando se trabaja con la entrada del teclado, donde el usuario utiliza caracteres de control.

Se proporcionan tres escapes de caracteres para caracteres de control específicos. \a coincide con el código de control de la campana. \b busca el carácter de retroceso y \e coincide con Escape. Los demás códigos de control se pueden hacer coincidir mediante el escape de caracteres \c. Esto debe ir seguido de la letra que representa el código de control. Por ejemplo, para hacer coincidir Ctrl-C, usaría la secuencia de escape de caracteres, \cC.

1.2.4 Coincidencia ASCII y caracteres Unicode

El último grupo de escapes de caracteres le permite hacer coincidir caracteres utilizando sus valores ASCII o Unicode. Hay varias opciones, dependiendo de si desea hacer coincidir con valores ASCII o Unicode, y si desea especificar los caracteres que desea buscar mediante octal o hexadecimal.

Para encontrar un carácter ASCII usando octal, simplemente use la barra invertida seguida de tres dígitos octales. Por ejemplo, el código siguiente busca todos los caracteres de la letra minúscula "e" con el código octal 157.

```
string input = "Hello World!";

foreach (Match match in Regex.Matches(input, @"\157"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT

Matched at index 4
Matched at index 7

*/
```

Para encontrar códigos ASCII usando hexadecimales, los dos dígitos hexadecimales deben seguir a \x. En el código siguiente se repite el ejemplo anterior con el código hexadecimal 6F.

```
string input = "Hello World!";

foreach (Match match in Regex.Matches(input, @"\x6F"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT

Matched at index 4
Matched at index 7

*/
```

Como el conjunto de caracteres Unicode es mucho más grande que ASCII, se requieren más dígitos para especificar un código. Puede buscar un carácter Unicode utilizando cuatro dígitos hexadecimales después del escape del carácter \u.

```
string input = "Hello World!";

foreach (Match match in Regex.Matches(input, @"\u006f"))
{
    Console.WriteLine("Matched at index {0}", match.Index);
}

/* OUTPUT

Matched at index 4
Matched at index 7

*/
```

1.3 Clases de caracteres

Hasta ahora, en el tutorial de expresiones regulares, hemos emparejado patrones basados solo en caracteres literales y secuencias de escape. Esto ha permitido una demostración de algunos de los métodos clave proporcionados por el motor de expresiones regulares y la clase Regex, pero ofrece pocas ventajas sobre la búsqueda mediante la clase de cadena. El verdadero poder de las expresiones regulares lo proporciona la capacidad de encontrar texto que coincida con un patrón más flexible.

En este artículo, comenzaremos a ver las formas en que puede crear un patrón que coincida con más que cadenas literales. Comenzaremos viendo las clases de caracteres, que a veces se conocen como conjuntos de caracteres. Permiten especificar que, al buscar un patrón, el carácter en la posición especificada en la expresión regular puede tener varios caracteres posibles en la posición correspondiente en la cadena de búsqueda. Un ejemplo común del uso de una clase de caracteres es hacer coincidir cualquier dígito numérico o hacer coincidir cualquier carácter. También son útiles para hacer coincidir palabras mal escritas o aquellas que tienen varias ortografías similares, como "serializar" y "serializar".

1.3.1 Comodín

El primero de los símbolos de clase de caracteres es el punto [.]. Se trata de un comodín que coincidirá con cualquier carácter del texto de origen. Es útil cuando desea hacer coincidir una cadena que contiene caracteres literales en posiciones específicas, pero donde los caracteres entre esos literales pueden ser cualquier cosa.

En el código de ejemplo siguiente, la variable de entrada se compara con una expresión regular que contiene tres caracteres comodín. Siempre que una 'r' va seguida de una 's' con tres caracteres en el medio, se encuentra una coincidencia. Tenga en cuenta que las dos letras literales distinguen entre mayúsculas y minúsculas, por lo que la operación de coincidencia encuentra "anillos" y "ruinas", pero no "Lluvias".

```
string input = "Rains ran rings around the Roman ruins";

foreach (Match match in Regex.Matches(input, "r...s"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}
/* OUTPUT
Matched 'rings' at index 10
Matched 'ruins' at index 33
*/
```

```
Dim input = "Las lluvias rodearon las ruinas romanas"

For Each match As Match In Regex.Matches(input, "r...n")
     Console.WriteLine("Coincide '{0}' en el índice {1}",
                       match.Value, match.Index)
Next
'
' OUTPUT
' Coincide 'roman' en el índice 32
'
```

1.3.2 Grupos de caracteres

Los grupos de caracteres actúan de manera similar a los comodines, lo que le permite hacer coincidir un solo carácter de la cadena de origen con uno de varios valores posibles. En lugar de hacer coincidir cualquier carácter, especifique exactamente qué caracteres se consideran coincidencias válidas.

Para crear un grupo de caracteres, escriba entre paréntesis todos los caracteres coincidentes aceptables. Por ejemplo, si desea hacer coincidir solo las vocales, puede usar el grupo de caracteres "[aeiou]".

Modifiquemos el ejemplo anterior para que encuentre todas las secuencias de cinco caracteres que comiencen con 'R' o 'r' y terminen con 's', eliminando la distinción entre mayúsculas y minúsculas de la primera letra:

```
string input = "Rains ran rings around the Roman ruins";

foreach (Match match in Regex.Matches(input, "[Rr]...s"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}
/* OUTPUT
Matched 'Rains' at index 0
Matched 'rings' at index 10
Matched 'ruins' at index 33
*/
```

1.3.3 Negación

Puede invertir el funcionamiento de un grupo de caracteres incluyendo un símbolo de intercalación (^) inmediatamente después del corchete de apertura. Un grupo de caracteres

negados coincide con cualquier símbolo único que no aparezca entre corchetes. Por ejemplo, para encontrar cualquier carácter que no sea una vocal, puedes usar "[^aeiou]".

En el ejemplo siguiente se buscan coincidencias de cinco caracteres que comienzan con 'R' o 'r' y terminan con 's' donde el tercer carácter no es una vocal.

```
string input = "Rains ran rings around the Roman ruins";

foreach (Match match in Regex.Matches(input, "[Rr].[^aeiou].s"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'rings' at index 10
*/
```

1.3.4 Rangos de caracteres

Los grupos de caracteres se vuelven difíciles de entender cuando contienen demasiados caracteres aceptables. Por ejemplo, si desea poder hacer coincidir cualquier letra minúscula, el grupo de caracteres, incluidos los corchetes, tendría 28 caracteres. En situaciones en las que los caracteres que deben coincidir son contiguos, puede utilizar un intervalo de caracteres en su lugar.

Con un intervalo de caracteres, se especifican el primer y el último carácter aceptable entre corchetes, separándolos con un guión. Por ejemplo, para hacer coincidir cualquier letra minúscula, puedes usar "[a-z]". En el caso de los dígitos numéricos, incluiría "[0-9]".

En el ejemplo siguiente se extraen números de teléfono de una cadena de origen mediante intervalos de caracteres. Los números deben tener un formato de cinco dígitos, seguido de un espacio y seis dígitos más.

```
string input = "Bob: 01234 567890, Mel: 01234 987654, Sue: 01432 123001";
string pattern = "[0-9][0-9][0-9][0-9][0-9] [0-9][0-9][0-9][0-9][0-9]";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value, match.Index);
}

/* OUTPUT
Matched '01234 567890' at index 5
Matched '01234 987654' at index 24
Matched '01432 123001' at index 43
*/
```

Nota: Al igual que con los grupos de caracteres, los rangos de caracteres se pueden negar utilizando el símbolo de intercalación. Para hacer coincidir cualquier carácter no numérico, puede usar "[^0-9]".

1.3.5 Combinando Clases de caracteres

Los rangos de caracteres se pueden combinar con grupos de caracteres para que coincidan con una gama más amplia de símbolos. Para combinarlos, incluye todas las opciones posibles entre paréntesis. Por ejemplo, para hacer coincidir cualquier carácter alfanumérico, puede combinar los rangos de letras minúsculas y mayúsculas con un rango de dígitos como "[a-zA-Z0-9]. Si también desea hacer coincidir un punto o una coma, puede agregarlos como caracteres individuales con "[a-zA-Z0-9.,]".

Para demostrarlo, pruebe el siguiente código. Esto extrae los precios de una lista de extras opcionales para un automóvil. La expresión regular utilizada es algo demasiado complicada. Podría simplificarse y mejorarse utilizando técnicas que aún no hemos visto en el tutorial.

```
string input = @"Option List
Metallic Paint        250.00
Alloy Wheels        1,050.00
Rear Spoiler          325.00
Leather Interior    2,350.00
Tinted Windows         99.00";

string pattern = "[0-9 ][, ][0-9 ][0-9][0-9].[0-9][0-9]";

foreach (Match match in Regex.Matches(input, pattern))
{
   Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched '  250.00' at index 35
Matched '1,050.00' at index 65
Matched '  325.00' at index 95
Matched '2,350.00' at index 125
Matched '   99.00' at index 155
*/
```

1.3.6 Clases de caracteres taquigrafía

Algunas clases de caracteres comunes se usan con tanta frecuencia que el lenguaje de expresiones regulares incluye versiones abreviadas. Esto puede hacer que los patrones sean más fáciles de leer y comprender rápidamente. Hay seis opciones clave de taquigrafía disponibles. Son los siguientes:

- **\d**. Coincide con cualquier dígito numérico. Este código es la abreviatura de "[0-9]".
- **\D**. La versión negada de \d. Coincide con cualquier carácter no numérico. Este código es la abreviatura de "[^0-9]".
- **\w**. Coincide con cualquier carácter de palabra. Un carácter de palabra es cualquier letra, dígito numérico o carácter de subrayado. Este código es la abreviatura de "[a-zA-Z0-9_]".
- **\W**. La versión negada de \w. Coincide con cualquier carácter que no sea una palabra. Este código es la abreviatura de "[^a-zA-Z0-9_]".
- **\s**. Coincide con cualquier carácter de espacio en blanco, incluidos espacios, tabulaciones, retornos de carro y saltos de línea.
- **\S**. La versión negada de \s. Coincide con cualquier carácter que no sea un espacio en blanco.

El código de ejemplo final recrea el ejemplo de número de teléfono anterior en el artículo. Esta vez, el código utiliza abreviaturas en lugar de rangos de caracteres.

```
string input = "Bob: 01234 567890, Mel: 01234 987654, Sue: 01432 123001";

foreach (Match match in Regex.Matches(input, @"\d\d\d\d\d \d\d\d\d\d\d"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched '01234 567890' at index 5
Matched '01234 987654' at index 24
Matched '01432 123001' at index 43
*/
```

1.4 Anclas

Al trabajar con expresiones regulares, puede utilizar un anclaje para indicar que debe producirse una coincidencia en una posición específica. Los anclajes se pueden usar para asegurarse de que las coincidencias se encuentran al principio o al final de la cadena de origen, o al principio del final de una línea en texto de varias líneas. También puede buscar patrones que se produzcan al principio o al final de una palabra o inmediatamente después de la coincidencia anterior.

Los anclajes de la expresión regular solo coinciden con una posición. No consumen caracteres del texto de origen y no aparecen en los resultados finales.

1.4.1 Inicio de la línea de anclaje

El primer anclaje que consideraremos coincide con el inicio de la cadena o el comienzo de una línea de texto. Se considera que las nuevas líneas comienzan inmediatamente después de un carácter de fin de línea (\n). Para establecer la posición necesaria para el carácter de inicio de línea, incluya un símbolo de intercalación (^) en la expresión regular.

Vamos a demostrarlo con una muestra. Primero, considere el código a continuación, que coincide con el texto "This" or "this". No se incluyen anclajes en la expresión regular, por lo que se encuentran cuatro elementos.

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, "[Tt]his"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched 'This' at index 0
Matched 'This' at index 41
Matched 'this' at index 87
Matched 'this' at index 133

*/
```

Para encontrar solo las palabras que aparecen al principio de la línea, puede incluir el anclaje antes del texto para que coincida, como se muestra a continuación. Tenga en cuenta el uso de RegexOptions.Multiline en la llamada al método Matches. Esto indica al motor de expresiones regulares que el texto debe procesarse como una cadena de varias líneas. Sin la opción, solo se encontrarán las coincidencias al principio del texto.

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, "^[Tt]his", RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched 'This' at index 0
Matched 'This' at index 41

*/
```

1.4.2 Anclaje de fin de línea

Puede buscar texto al final de la cadena o al final de una línea utilizando el anclaje de fin de línea ($). Con este anclaje, el resto del patrón debe coincidir inmediatamente antes de un carácter de salto de línea.

Intente ejecutar el siguiente código:

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, "string.$", RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'string.' at index 163
*/
```

En el ejemplo anterior se busca el texto "string.". Un anclaje de fin de línea especifica que solo se deben encontrar coincidencias al final de la cadena o al final de una línea. Sin embargo, aunque hay tres apariciones del texto, solo se coincide con la última. Esto se debe a que las cadenas de .NET terminan cada línea con un retorno de carro y un salto de línea combinados (\r\n), pero el motor de expresiones regulares solo busca el salto de línea.

Para hacer coincidir correctamente las cadenas de .NET con el delimitador de fin de línea, debe incluir el retorno de carro en el patrón. Para otras cadenas, es posible que el retorno de carro no esté presente. Para crear un patrón que coincida correctamente con o sin el retorno de carro, incluya el texto "\r?" en el patrón. El signo de interrogación indica que el patrón debe coincidir con \r cero o una vez. Es uno de los cuantificadores, que veremos más adelante en el tutorial.

Actualice el ejemplo anterior, como se muestra a continuación, y vuelva a ejecutarlo para ver los resultados. Nota: La llamada a Replace se incluye para evitar que los retornos de carro coincidentes se incluyan en la salida.

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, @"string.\r?$", RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value.Replace("\r", ""), match.Index);
}

/* OUTPUT
Matched 'string.' at index 32
Matched 'string.' at index 120
Matched 'string.' at index 163
*/
```

1.4.3 Comienzo de la cadena del ancla

Si solo desea encontrar coincidencias al principio del texto, puede usar el ancla "\A". Esto se demuestra en el siguiente código. Tenga en cuenta que, aunque la opción Multilínea todavía está en uso, solo se devuelve la primera coincidencia.

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, @"\A[Tt]his",
RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value, match.Index);
}

/* OUTPUT
Matched 'This' at index 0
*/
```

1.4.4 Fin de la cadena del ancla

Para hacer coincidir el texto al final del texto de origen solamente, puede usar el anclaje "\Z". La coincidencia debe estar al final de la cadena o inmediatamente antes de un carácter de salto de línea final. Al igual que con el delimitador de fin de línea, al trabajar con cadenas creadas con .NET Framework, debe comprobar si hay un retorno de carro opcional, como se muestra en el código de ejemplo siguiente:

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, @"string.\Z",
RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value,
match.Index);
}

/* OUTPUT
Matched 'string.' at index 163
*/
```

Si modifica el anclaje para usar una versión en minúsculas, "\z", la coincidencia debe aparecer solo al final de la cadena. Si va seguido de un salto de línea adicional, el patrón no coincidirá.

```
string input = @"This bit is at the start of the string.
This bit is at the start of a line.
However, this bit is in the middle of the string.
And this bit is at the end of the string.";

foreach (Match match in Regex.Matches(input, @"string.\z",
                                      RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value, match.Index);
}

/* OUTPUT
Matched 'string.' at index 163
*/
```

1.4.5 Anclaje de límite de palabra

Un elemento de uso común es la palabra ancla de límite (\b). Esto coincide con la posición en la que comienza o termina una palabra. Las palabras se definen como grupos de caracteres de palabras contiguos, que incluyen letras, dígitos numéricos y guiones bajos.

El siguiente código coincide con el texto "Can" o "can" donde aparece al principio de una palabra. Esto significa que "can" en "tucán" y el segundo "can" en "cancán" no coinciden.

```
string input = "Can the cantankerous toucan dance the cancan?";

foreach (Match match in Regex.Matches(input, @"\b[Cc]an"))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value, match.Index);
}

/* OUTPUT
Matched 'Can' at index 0
Matched 'can' at index 8
Matched 'can' at index 38
*/
```

1.4.6 Anclaje de límite sin palabras

El anclaje de límite que no es de palabra es lo opuesto al anclaje de límite de palabra. Garantiza que la posición coincidente esté entre dos caracteres de palabra o dos caracteres que no sean de palabra. El anclaje se especifica mediante "\B".

Intente ejecutar el siguiente código. Esto coincide con el texto "Can" o "can" donde no se encuentra al comienzo de una palabra.

```
string input = "Can the cantankerous toucan dance the cancan?";

foreach (Match match in Regex.Matches(input, @"\B[Cc]an"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'can' at index 24
Matched 'can' at index 41
*/
```

1.4.7 Ancla de coincidencia contigua

El último anclaje es el anclaje de coincidencia contiguo (\G). Esto coincide con la posición del personaje directamente después del final de la partida anterior. El ancla es ideal para extraer elementos de listas. La primera coincidencia se produce al principio de la cadena y también se devuelven las coincidencias que siguen inmediatamente.

Para demostrarlo, intente ejecutar el código siguiente. Aquí solo estamos haciendo coincidir caracteres numéricos y cada coincidencia debe aparecer directamente después de la anterior. Los primeros cinco dígitos coinciden. Los cinco números restantes no se devuelven porque el guión impide que coincidan con ellos.

```
string input = "12345-67890";

foreach (Match match in Regex.Matches(input, @"\G[0-9]"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched '1' at index 0
Matched '2' at index 1
Matched '3' at index 2
Matched '4' at index 3
Matched '5' at index 4

*/
```

1.5 Alternancia

Hasta ahora, en el tutorial de expresiones regulares, hemos trabajado con expresiones regulares que coinciden con un solo patrón. En algunos casos, es posible que desee hacer coincidir uno de varios patrones. Podrías realizar varios partidos y combinar los resultados. Sin embargo, el uso de la alternancia le permite buscar varios patrones de expresiones regulares con una sola operación.

Para utilizar la alternancia, utilice el carácter de barra vertical (|) para delimitar una serie de patrones. Puedes incluir tantos patrones como desees. Por ejemplo, el código siguiente coincide con "dog" o "fox" en la cadena de entrada:

```
string input = "The quick brown fox jumps over the lazy dog.";

foreach (Match match in Regex.Matches(input, @"dog|fox"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'fox' at index 16
Matched 'dog' at index 40
*/
```

El código de ejemplo anterior busca patrones que contengan solo caracteres literales. Sin embargo, puede incluir cualquier elemento de lenguaje de expresiones regulares dentro de cada uno de los patrones. Por ejemplo, la versión actualizada que se muestra a continuación coincide con tres patrones. La primera es "d.g", que coincide con cualquier secuencia de tres caracteres que comience con 'd' y termine con 'g'. El segundo patrón busca una 'f', seguida de cualquier carácter individual y una 'x'. El tercer patrón utiliza la clase de caracteres de palabra y los anclajes de límite de palabra para encontrar palabras de cuatro letras.

```
string input = "The quick brown fox jumps over the lazy dog.";

foreach (Match match in Regex.Matches(input, @"d.g|f.x|\b\w\w\w\w\b"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'fox' at index 16
Matched 'over' at index 26
Matched 'lazy' at index 35
Matched 'dog' at index 40
*/
```

1.5.1 La alternancia dentro de una expresión más grande

La alternancia es útil cuando se desea encontrar varios patrones. También se puede usar dentro de una expresión regular, donde parte del patrón debe coincidir con una de un conjunto de opciones. Por ejemplo, es posible que desee encontrar las palabras "perro", "zorro" o "saltar". Si usaras la expresión "perro|zorro|salto", coincidirías con esas palabras. Sin embargo, también encontraría coincidencias en las que cualquiera de esas palabras apareciera dentro de una palabra más grande. Puede agregar anclajes de límite de palabra a las tres palabras o, como en el ejemplo siguiente, puede contener los elementos de alternancia entre paréntesis:

```
string input = "The quick brown fox jumps over the lazy dog.";

foreach (Match match in Regex.Matches(input, @"\b(dog|fox|jump)\b"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched 'fox' at index 16
Matched 'dog' at index 40

*/
```

NB: Los paréntesis se puede utilizar en muchas situaciones en las que un grupo de personajes se debe procesar como un solo elemento.

1.6 Cuantificadores

Hasta ahora, en el tutorial de expresiones regulares, hemos visto cómo se pueden combinar literales y caracteres especiales para crear un patrón, que se puede hacer coincidir dentro de una cadena. En casi todos los casos, el patrón solo ha sido capaz de generar coincidencias de una longitud fija.

Los cuantificadores permiten definir un patrón que contiene caracteres, o grupos de caracteres, que se repiten. Permiten expresiones regulares mucho más potentes que incluyen caracteres opcionales, texto que debe aparecer pero que puede repetirse, y patrones que coinciden tanto con el texto que se repite como con los elementos que faltan. Incluso puede especificar el número de veces que un elemento debe repetirse para ser aceptado como coincidencia.

En este capitulo veremos ejemplos de todos los cuantificadores, incluida una expresión regular que coincide con precisión con las direcciones IP.

1.6.1 Emparejamiento opcional

En el artículo que describe los anclajes en expresiones regulares, vimos que el carácter de signo de interrogación (?) se usa para hacer coincidir opcionalmente un retorno de carro. Este es el primer cuantificador. Especifica que el carácter inmediatamente anterior al signo de interrogación debe coincidir con cero o una vez.

En el ejemplo siguiente se muestra el uso del cuantificador. Aquí se encuentra la palabra "color", con o sin la letra 'u'.

```
string input = "The USA spelling of colour is color.";
 foreach (Match match in Regex.Matches(input, "colou?r")){
  Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}
 /* OUTPUT
     Matched 'colour' at index 20
     Matched 'color' at index 30
* /
```

Todos los cuantificadores se pueden aplicar a un solo carácter o a un patrón contenido entre paréntesis. Por ejemplo, el siguiente código encuentra "fox" or "lazy fox".

```
string input = "The quick brown fox jumps over the lazy fox.";
foreach (Match match in Regex.Matches(input, @"(lazy )?fox"))
{
Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched 'fox' at index 16
Matched 'lazy fox' at index 35
*/
```

1.6.2 Coincidencia de caracteres repetidos

Si desea encontrar un carácter o patrón que se repita, puede usar el signo más (+). Esto especifica que el elemento inmediatamente anterior debe aparecer al menos una vez, pero puede aparecer muchas veces.

El siguiente código busca el patrón, "\d+". Esto busca todos los grupos de dígitos numéricos adyacentes.

```
string input = "1, 1, 2, 3, 5, 8, 13, 21, 34";

foreach (Match match in Regex.Matches(input, @"\d+"))
{
Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched '1' at index 0
Matched '1' at index 3
Matched '2' at index 6
Matched '3' at index 9
Matched '5' at index 12
Matched '8' at index 15
Matched '13' at index 18
Matched '21' at index 22
Matched '34' at index 26
*/
```

1.6.3 Coincidencia opcional de caracteres repetidos

El cuantificador de asterisco (*) también permite encontrar una serie de elementos que se repiten. Sin embargo, si el elemento que precede al asterisco no aparece, aún es posible que coincida. Esto se debe a que el cuantificador coincide con el elemento cero o más veces.

Considere el siguiente código. La expresión regular busca texto que contenga cero o más letras mayúsculas seguidas de cero o más dígitos numéricos. El texto de origen contiene cinco cadenas alfanuméricas que coinciden. Además, se encuentra una cadena de longitud cero inmediatamente después de cada una de esas cadenas coincidentes. Estas son coincidencias porque incluyen cero letras y cero números.

```
string input = "1234 A123 AB12 ABC1 ABCD";

foreach (Match match in Regex.Matches(input, @"[A-Z]*[0-9]*"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched '1234' at index 0
Matched '' at index 4
Matched 'A123' at index 5
Matched '' at index 9
Matched 'AB12' at index 10
Matched '' at index 14
Matched 'ABC1' at index 15
Matched '' at index 19
Matched 'ABCD' at index 20
Matched '' at index 24
*/
```

1.6.4 Hacer coincidir un número específico de elementos repetidos

Cuando necesite más control sobre el número de repeticiones, puede especificar exactamente cuántas veces debe aparecer un patrón. Para hacerlo, siga la parte repetida del patrón con un número contenido entre llaves. Por ejemplo, para encontrar un elemento tres veces, debe seguirlo con "{3}".

El código de ejemplo siguiente busca grupos de cuatro dígitos adyacentes. Solo hay una coincidencia de este tipo en la cadena de entrada.

```
string input = "1234 A123 AB12 ABC1 ABCD";

foreach (Match match in Regex.Matches(input, @"\d{4}"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT
Matched '1234' at index 0
*/
```

Puede utilizar llaves para especificar el número mínimo de repeticiones que deben coincidir con un patrón menos restrictivo. Para hacerlo, agregue una coma después del número dentro de las llaves. Por ejemplo, "{2,}" coincide con dos o más elementos.

En el ejemplo actualizado siguiente se encuentran dos o más dígitos numéricos adyacentes:

```
string input = "1234 A123 AB12 ABC1 ABCD";

foreach (Match match in Regex.Matches(input, @"\d{2,}"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched '1234' at index 0
Matched '123' at index 6
Matched '12' at index 12

*/
```

Por último, puede especificar un número mínimo y máximo de repeticiones dentro de las llaves. El mínimo aparece primero y está separado del máximo por una coma. Por ejemplo, para encontrar entre uno y tres elementos, usaría "{1,3}".

En el código de ejemplo siguiente se buscan elementos que contienen entre una y tres letras mayúsculas seguidas de uno a tres dígitos:

```
string input = "1234 A123 AB12 ABC1 ABCD";

foreach (Match match in Regex.Matches(input, "[A-Z]{1,3}[0-9]{1,3}"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched 'A123' at index 5
Matched 'AB12' at index 10
Matched 'ABC1' at index 15

*/
```

1.6.5 Cuantificadores codiciosos y perezosos

Los cuantificadores pueden ser codiciosos o perezosos. Todos los ejemplos anteriores utilizan cuantificadores codiciosos, lo que significa que coinciden con tantas repeticiones como sea posible. Por ejemplo, si se hace coincidir "\d+", se buscará un dígito numérico y se hará coincidir con él y con todos los caracteres siguientes hasta que se encuentre un carácter no numérico, o el final del texto.

Un cuantificador diferido funciona de manera diferente. Tan pronto como se hayan encontrado suficientes caracteres para corresponder al patrón, se devuelve una coincidencia. Esto significa que los cuantificadores perezosos devuelven el menor número posible de repeticiones. A menudo, esto significa que el texto de origen devolverá más coincidencias, pero cada una será más corta. Para especificar que un cuantificador debe ser diferido, anexe un signo de interrogación (?).

Intente ejecutar el siguiente código. En este caso, el mismo texto de entrada se compara con dos patrones. El primero busca un '2', seguido de uno o más dígitos. Como el cuantificador es codicioso, hay una sola coincidencia que consume el primer '2' y todos los dígitos posteriores.

El segundo patrón es similar al primero, pero utiliza un cuantificador diferido. Como tal, hay dos coincidencias, cada una de solo dos caracteres de longitud. Las coincidencias se producen en el índice de cada '2', coincidiendo con ese número y solo un dígito más.

```
string input = "A long number: 0123456789876543210";

foreach (Match match in Regex.Matches(input, @"2\d+"))
{
    Console.WriteLine("Greedy matched '{0}' at index {1}", match.Value, match.Index);
}

Console.WriteLine();

foreach (Match match in Regex.Matches(input, @"2\d+?"))
{
    Console.WriteLine("Lazy matched   '{0}' at index {1}", match.Value, match.Index);
}


/* OUTPUT

Greedy matched '23456789876543210' at index 17

Lazy matched   '23' at index 17
Lazy matched   '21' at index 31

*/
```

1.6.6 Coincidencia de una dirección IP

Para un ejercicio final, vamos a crear un patrón que coincida con precisión con las direcciones IP. A diferencia del patrón que se mencionó al principio del tutorial de expresiones regulares, no buscaremos simplemente tres números separados por puntos o puntos. Crearemos un patrón que garantice que cada uno de los cuatro números esté en el rango de 0 a 255.

El patrón que necesitamos tiene que coincidir con un número válido cuatro veces. Consideremos primero cómo podemos hacer coincidir un solo número que se encuentre dentro del rango correcto. Sabemos que cada número tendrá uno, dos o tres dígitos, por lo que un patrón ingenuo buscaría esta cantidad de números. La siguiente expresión regular logra esto con el uso de la clase de caracteres numéricos de dígitos y un cuantificador.

```
\d{1,3}
```

El patrón anterior no funciona correctamente porque el rango de valores encontrados es de cero a 999. Vamos a mejorarlo ligeramente. Si el número tiene uno o dos dígitos de longitud, lo aceptaremos. Para valores de tres dígitos, asegurémonos de que comience con '1' o '2'. Podemos utilizar una alternancia con los tres patrones aceptables, de la siguiente manera:

```
(2\d{2}|1\d{2}|\d{1,2})
```

Esto limita el rango potencial de valores entre cero y 299, lo que sigue siendo incorrecto. También significa que aceptaremos números de dos dígitos con ceros a la izquierda. Vamos a tratar primero los ceros a la izquierda utilizando una alternancia que permita un solo dígito o dos dígitos donde el primero está en el rango del 1 al 9:

```
(2\d{2}|1\d{2}|[1-9]\d|\d)
```

Para obtener el rango correcto de 0-255, eliminaremos la primera opción de la alternancia y la reemplazaremos con dos nuevos patrones posibles. El primero atenderá los valores 200-249 haciendo coincidir un '2', seguido de un rango de caracteres de "0-4" y cualquier dígito final. El segundo coincidirá con los valores de 250 a 255. Usaremos un patrón que comience con "25" y termine con un dígito entre cero y cuatro. El patrón actualizado se muestra a continuación:

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

Necesitamos hacer coincidir el patrón anterior cuatro veces con un punto entre cada elemento. Una forma de lograr esto es agrupando el patrón anterior y un punto, con un cuantificador que requiere exactamente tres copias del patrón, seguidas de otra copia del patrón numérico. La expresión regular completa es la siguiente:

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

En este ejemplo se puede ver que las expresiones regulares pueden ser muy eficaces, pero también pueden volverse complejas muy rápidamente. Para demostrar el patrón anterior, intente ejecutar el código siguiente. Esto extrae las direcciones IP válidas de la cadena de entrada e ignora las no válidas.

1.7 Construcciones de agrupación

En un artículo anterior del tutorial de expresiones regulares, hicimos coincidir grupos de caracteres rodeándolos con paréntesis. Esto le permite combinar una secuencia de literales y caracteres de patrón con un cuantificador para encontrar coincidencias repetidas u opcionales. En este artículo, veremos algunas características adicionales de las construcciones de agrupación y su uso con el motor de expresiones regulares de .NET.

Para empezar, recapitulemos con un programa de ejemplo. El siguiente código busca las etiquetas de anclaje dentro de algún código HTML:

1.7.1 Agrupación constructos

En un artículo anterior Buscamos [matched] grupos de caracteres rodeándolos con paréntesis. Esto le permite combinar una secuencia de literales y caracteres de patrones con un cuantificador para encontrar repetir búsquedas opcionales. En este artículo vamos a ver algunas de las características adicionales de las construcciones de la agrupación y su uso con el .NET motor de expresiones regulares.

Para empezar, vamos a recapitular con un programa de ejemplo. El siguiente código encuentra las etiquetas de anclaje dentro de un cierto código HTML:

```
string input = "For more information use the "
             + "<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a> "
             + "or check the list of "
             + "<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently "
             + "asked questions</a>.";

string pattern = "(<a href=')(.*?)('>)(.*?)(</a>)";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Matched '{0}'", match.Value);
}

/* OUTPUT
Matched '<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>'
Matched '<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently asked
questions</a>'
*/
```

La expresión regular utilizada en el código anterior es bastante ingenua, ya que solo encontrará anclajes que tengan el mismo formato que los de la cadena de entrada. Utiliza cinco subexpresiones agrupadas, cada una entre paréntesis, para realizar la coincidencia de patrones. Las subexpresiones son:

Subexpresión	Propósito
(a href = ')	Busca el texto literal, "<a href =" ", la identificación de la parte inicial de una etiqueta de ancla, hasta la posición de partida de la dirección URL de destino.
(. *?)	Encaja en una serie de caracteres consecutivos de una manera perezosa. Este grupo coincide con la URL definida dentro de un ancla.
(>)	Encuentra los dos caracteres literales que cierran etiqueta de apertura del ancla.
(. *?)	Coincide con otra serie de caracteres de manera perezosa. Este grupo busca el texto entre las etiquetas de apertura y cierre del anclaje. Este es el texto que se mostraría como un hipervínculo en un navegador web.
()	Coincide con la etiqueta de cierre para el ancla.

En el ejemplo, las construcciones de agrupación se usan únicamente para que coincidan correctamente con el patrón general. Sin embargo, tienen muchos otros usos. Por ejemplo, puede extraer el texto que coincida con cualquiera de las subexpresiones mediante clases de .NET Framework. También puede usar el texto coincidente de una subexpresión dentro de otra, o realizar la funcionalidad de búsqueda y reemplazo en los grupos, que veremos en futuros artículos.

1.7.2 Obtención de Grupos capturados

Al buscar una expresión regular que contiene grupos, la subexpresión de cada grupo coincide y estos resultados se pueden obtener individualmente. Se mantienen en una colección de objetos Group en la propiedad Groups del objeto Match. La colección Groups siempre contiene al menos un elemento en el índice cero. Este es el partido completo. Las subexpresiones coincidentes se derivan del índice uno. Aparecen en el orden de los grupos del patrón e incluyen los grupos anidados.

El código siguiente realiza la misma coincidencia que en el primer ejemplo. Esta vez, la salida incluye el texto de la coincidencia completa seguido de dos de las subexpresiones coincidentes. La primera es la URL del anclaje, que se encuentra en el índice 2. El segundo, en

el índice 4, es el texto que se mostraría como un hipervínculo cuando el anclaje se representa en un navegador web.

```
string input = "For more information use the "
               + "<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a> "
               + "or check the list of "
               + "<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently "
               + "asked questions</a>.";

string pattern = "(<a href=')(.*?)('>)(.*?)(</a>)";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Match: '{0}'", match.Value);
    Console.WriteLine("URL: '{0}'", match.Groups[2]);
    Console.WriteLine("Text: '{0}'", match.Groups[4]);
    Console.WriteLine();
}

/* OUTPUT

Match: '<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>'
URL: 'http://www.blackwasp.co.uk/Contact.aspx'
Text: 'contact form'

Match: '<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently asked questions</a>'
URL: 'http://www.blackwasp.co.uk/FAQ.aspx'
Text: 'frequently asked questions'

*/
```

1.7.3 Grupos capturados con nombre

Al hacer coincidir subexpresiones y extraer los resultados, a menudo puede hacer que el patrón sea más fácil de entender mediante el uso de grupos con nombre, en lugar de los numerados que se ven arriba. Agregar un nombre al patrón significa que puede recuperar la información por nombre en lugar de número. Esto es especialmente útil si está generando la expresión regular en el código y el número de grupos puede variar.

Para asignar un nombre a un grupo, agregue un signo de interrogación inmediatamente después del carácter de paréntesis inicial. El nombre debe seguir, entre corchetes angulares (<>). Por ejemplo, el código siguiente aplica nombres a los grupos que coinciden con la dirección URL dentro del anclaje y el contenido mostrado. Utiliza los nombres al generar la información.

```
string input = "For more information use the "
                + "<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>
"
                + "or check the list of "
                + "<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently "
                + "asked questions</a>.";

string pattern = "(<a href=')(?<url>.*?)('>)(?<text>.*?)(</a>)";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Match: '{0}'", match.Value);
    Console.WriteLine("URL: '{0}'", match.Groups["url"]);   // Matches extracted by
name
    Console.WriteLine("Text: '{0}'", match.Groups["text"]);
    Console.WriteLine();
}

/* OUTPUT

Match: '<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>'
URL: 'http://www.blackwasp.co.uk/Contact.aspx'
Text: 'contact form'

Match: '<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently asked questions
</a>'
URL: 'http://www.blackwasp.co.uk/FAQ.aspx'
Text: 'frequently asked questions'

*/
```

1.7.4 Grupos que no capturan

Si usa construcciones de grupo solo para generar un patrón y no necesita usar la coincidencia de subexpresión, ya sea de la colección Groups o en cualquier otro lugar de la expresión regular, se recomienda usar un grupo que no sea de captura. Este tipo de grupo coincide de la misma manera, pero no se le asigna un número ni un nombre, se excluye de la colección Groups y no genera la sobrecarga adicional de un grupo de captura estándar.

Para deshabilitar la captura de un grupo, agregue un signo de interrogación y dos puntos (?:) inmediatamente después del paréntesis de apertura. El código de ejemplo siguiente se actualiza para que solo se capturen los elementos de contenido de la dirección URL y el hipervínculo. Por lo tanto, los resultados de los grupos se extraen de los índices 1 y 2 de la propiedad de colección Groups.

```
string input = "For more information use the "
                + "<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>
"
                + "or check the list of "
                + "<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently "
                + "asked questions</a>.";

string pattern = "(?:<a href=')(.*?)(?:'>)(.*?)(?:</a>)";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Match: '{0}'", match.Value);
    Console.WriteLine("URL: '{0}'", match.Groups[1]);
    Console.WriteLine("Text: '{0}'", match.Groups[2]);
    Console.WriteLine();
}

/* OUTPUT

Match: '<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>'
URL: 'http://www.blackwasp.co.uk/Contact.aspx'
Text: 'contact form'

Match: '<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently asked questions
</a>'
URL: 'http://www.blackwasp.co.uk/FAQ.aspx'
Text: 'frequently asked questions'

*/
```

Hay muchas otras maneras en que usted puede utilizar numeradas y grupos nombrados. Vamos a ver algunas de las opciones en plazos posteriores del tutorial.

1.8 Referencias inversas

Al hacer coincidir patrones de cadena con expresiones regulares, es posible que desee hacer coincidir el mismo fragmento de texto más de una vez. Cuando el patrón utilizado para realizar la primera coincidencia incluye elementos no literales, puede buscar el texto repetido mediante una referencia inversa. Una referencia inversa en una expresión regular identifica un grupo previamente coincidente y vuelve a buscar exactamente el mismo texto.

Un ejemplo sencillo del uso de referencias inversas es cuando se desea buscar palabras adyacentes y repetidas en algún texto. La primera parte de la coincidencia podría usar un patrón que extraiga una sola palabra. La siguiente parte sería una referencia inversa que hace referencia al grupo capturado. Crearemos un ejemplo de este tipo en este artículo.

1.8.1 Referencias inversas numeradas

Una referencia inversa numerada coincide con el texto que ya se encuentra en un grupo. Simplemente agregue un carácter de barra invertida y el número del grupo para que coincida nuevamente. Por ejemplo, para encontrar el texto que coincide con el primer grupo en una expresión regular, debe incluir "\1" en el patrón de expresión regular. A continuación, el texto extraído en el grupo capturado se busca en la cadena de entrada en la posición de la referencia inversa.

Probemos un programa de ejemplo. Cree una nueva aplicación de consola y agregue el código siguiente al método Main:

```
string input = "An easy mistake to make when writing multiline text is "
             + "is typing the same word twice, once at the start of a "
             + "a line and again at the end. This duplication can't "
             + "can't always be easily spotted.";

string pattern = @"(\b\S+)\s+\1\b";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Duplication of '{0}' at index {1}.",
                      match.Groups[1], match.Index);
}

/* OUTPUT

Duplication of 'is' at index 52.
Duplication of 'a' at index 107.
Duplication of 'can't' at index 155.

*/
```

Considere la expresión regular que se mantiene en la variable de patrón. El primer grupo, "(\b\S+)", busca un anclaje de límite de palabra, seguido de una serie de caracteres que no sean espacios en blanco. Cuando se combina con lo siguiente, "\s", se encuentran palabras enteras seguidas de un carácter de espacio en blanco. El patrón encontrará caracteres que no sean palabras, lo que significa que también se incluyen símbolos.

La siguiente parte del patrón es una referencia inversa al primer grupo, seguida de un límite de palabra. La referencia inversa coincidirá con el texto capturado del primer grupo, por lo que el patrón completo identifica las palabras duplicadas y adyacentes.

Nota: La sintaxis de las referencias inversas es similar a la de los caracteres literales octales. El motor de expresiones regulares asume una referencia inversa si la barra diagonal inversa va seguida de un solo dígito numérico. Para números más largos, se utiliza una referencia inversa si hay suficientes grupos de captura. Si no los hay, se hace coincidir un literal.

1.8.2 Referencias inversas con nombre

Una forma de evitar la ambigüedad de las referencias inversas numeradas es utilizar referencias inversas con nombre. Estos le permiten hacer coincidir el texto que ha capturado un grupo con nombre. Si se ha utilizado el mismo nombre dos o más veces, la referencia

inversa coincidirá con el texto de la coincidencia más reciente. Para definir una referencia inversa con nombre, utilice "\k", seguido del nombre del grupo.

El siguiente código realiza la misma función que el ejemplo anterior. Esta vez la primera palabra es capturada en un grupo llamado "repetido". La referencia inversa utiliza este nombre para buscar las palabras duplicadas.

```
string input = "An easy mistake to make when writing multiline text is "
             + "is typing the same word twice, once at the start of a "
             + "a line and again at the end. This duplication can't "
             + "can't always be easily spotted.";

string pattern = @"(?<repeated>\b\S+)\s+\k<repeated>\b";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Duplication of '{0}' at index {1}.", match.Groups[1],
match.Index);
}

/* OUTPUT

Duplication of 'is' at index 52.
Duplication of 'a' at index 107.
Duplication of 'can't' at index 155.

*/
```

1.9 Mirar hacia adelante y hacia atrás

Las aserciones lookahead y lookbehind de longitud cero, a veces conocidas como aserciones lookaround, son tipos especiales de grupos que no son de captura. Permiten realizar coincidencias complejas basadas en información que sigue o precede a un patrón, sin que la información de la aserción de búsqueda anticipada forme parte del texto devuelto.

1.9.1 Mirada positiva

El primer tipo de aserción de búsqueda es la búsqueda anticipada positiva. Esta construcción aparece después de un patrón inicial que se va a coincidir. Afirma que la primera parte del patrón debe ser seguida directamente por el elemento lookahead. Sin embargo, la coincidencia devuelta solo contiene el texto que coincide con la primera parte.

Para definir una aserción de búsqueda anticipada positiva, cree un grupo, rodeado de paréntesis, que comience con un signo de interrogación y un signo igual (?=). El texto entre paréntesis, después del signo igual inicial, es el patrón de la búsqueda anticipada.

Para demostrarlo, considere el siguiente código simple:

```
string input = "Andy Smith\n"
            + "Jim Brown\n"
            + "Lisa Smith\n"
               + "Sue Brown";

string pattern = @"^\w+(?=\sBrown)";

foreach (Match match in Regex.Matches(input, pattern,
                                       RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}.", match, match.Index);
}

/* OUTPUT
Matched 'Jim' at index 11.
Matched 'Sue' at index 32.
*/
```

El patrón anterior coincide con los nombres de pila de las personas con el apellido "Brown". La primera parte de la expresión regular es "^\w+". Captura uno o más caracteres de palabra al principio de una línea, extrayendo los nombres. Sin la búsqueda anticipada, el patrón coincidiría con los cuatro nombres de la lista de la cadena de entrada.

La segunda parte, "(?=\sBrown)" es la aserción de búsqueda anticipada de longitud cero. El patrón que debe coincidir es un carácter de espacio en blanco, seguido del apellido "Brown". Esto significa que toda la expresión regular devuelve los nombres "Jim" y "Sue".

1.9.2 Anticipación negativa

Podría decirse que las afirmaciones negativas de anticipación son más poderosas que sus contrapartes positivas. Afirman que el patrón en la búsqueda anticipada no debe seguir el texto que coincide con la parte inicial del patrón. Para crear una búsqueda anticipada negativa, sustituya el signo igual de la variación positiva por un signo de exclamación (!).

El siguiente ejemplo coincide con la primera palabra al principio de cada línea. La búsqueda anticipada negativa elimina las coincidencias si van seguidas de un espacio y el apellido "Brown".

```
string input = "Andy Smith\n"
             + "Jim Brown\n"
             + "Lisa Smith\n"
             + "Sue Brown";

string pattern = @"^\w+\b(?! Brown)";

foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}.", match, match.Index);
}

/* OUTPUT
Matched 'Andy' at index 0.
Matched 'Lisa' at index 21.
*/
```

1.9.3 Búsqueda retrospectiva positiva

La búsqueda retrospectiva positiva invierte el orden de la búsqueda anticipada positiva. La parte de búsqueda posterior del patrón, que suele aparecer al principio de una expresión regular, especifica el texto que debe aparecer antes del texto que se devolverá. La aserción lookbehind se define como un grupo entre paréntesis. Después del paréntesis de apertura, la cadena "?<=" prefija el patrón para que coincida.

En el ejemplo siguiente, el elemento lookbehind busca el nombre "Lisa" al principio de una línea y seguido de un espacio. A esto le sigue un patrón que captura uno o más caracteres de palabra. El resultado es que los apellidos de las personas llamadas "Lisa" coinciden.

```
string input = "Andy Smith\n"
             + "Jim Brown\n"
             + "Lisa Smith\n"
             + "Sue Brown";

string pattern = @"(?<=^Lisa )\w+";

foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}.", match, match.Index);
}

/* OUTPUT
Matched 'Smith' at index 26.
*/
```

1.9.4 Búsqueda retrospectiva negativa

Al igual que con la búsqueda anticipada, puede crear aserciones de búsqueda tardía negativas. Especifican que una coincidencia solo es válida si va precedida del texto del grupo lookbehind. De nuevo, el signo igual de la variante positiva se sustituye por un signo de exclamación.

El siguiente código busca todos los apellidos que no siguen al nombre "Lisa" al principio de una nueva línea:

```
string input = "Andy Smith\n"
             + "Jim Brown\n"
             + "Lisa Smith\n"
              + "Sue Brown";

string pattern = @"(?<!^Lisa )\b\w+$";

foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
{
    Console.WriteLine("Matched '{0}' at index {1}.", match, match.Index);
}

/* OUTPUT

Matched 'Smith' at index 5.
Matched 'Brown' at index 15.
Matched 'Brown' at index 36.

*/
```

1.10 Coincidencia condicional de expresiones regulares

Se examina la construcción de coincidencia condicional. Esto permite que una expresión regular incluya una condición de estilo 'if-then-else'.

1.10.1 Coincidencia condicional

La construcción de coincidencia condicional proporcionada por el motor de expresiones regulares le permite hacer coincidir el texto de diferentes maneras, de acuerdo con una coincidencia inicial. La construcción incluye tres grupos de caracteres de patrón. El primer grupo, que no es de captura, define el elemento condicional. A continuación, se muestran dos grupos de captura con patrones que coinciden y devuelven texto. Si la condición coincide correctamente, se utiliza el primer grupo de captura. Si no es así, se utiliza el segundo.

La sintaxis de una coincidencia condicional se muestra a continuación:

```
(? (expresión) entonces|más)
```

El grupo de expresiones es la parte que determina cuál de los dos elementos siguientes debe coincidir. Si el grupo de expresiones coincide correctamente con el texto, se coincide con el patrón "then". Si no es así, se coincide con la parte 'else'. El grupo condicional es una aserción de longitud cero, muy parecida a una búsqueda anticipada o retrospectiva. Esto significa que no consume ningún carácter y no se incluye en el texto coincidente.

Considere el siguiente programa. La cadena de entrada representa parte de un archivo de inicialización simple que contiene categorías y configuraciones, donde cada configuración incluye una clave y un valor. La expresión regular extrae las categorías y los ajustes regulares

```
string input = "[WindowPosition]\n"
             + "x=200\n"
             + "y=150\n"
             + "\n"
             + "[WindowSize]\n"
             + "height=200\n"
             + "width=150\n";

string groupPattern = @"(?:^\[)(?<Group>.+)(?:\]$)";
string valuePattern = @"(?:^)(?<Key>.+)=(?<Value>.+)(?:$)";

string pattern = string.Format(@"(?(^\[){0}|{1})", groupPattern, valuePattern);

Console.WriteLine("Full Pattern: {0}\n", pattern);

foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
{
    if (match.Groups["Group"].Success)
        Console.WriteLine("Group: {0}", match.Groups[1]);
    else
        Console.WriteLine("Value: '{0}' is set to {1}",
                          match.Groups[2], match.Groups[3]);
}

/* OUTPUT

Full Pattern: (?(^\[)(?:^\[)(?<Group>.+)(?:\]$)|(?:^)(?<Key>.+)=(?<Value>.+)(?:$))

Group: WindowPosition
Value: 'x' is set to 200
Value: 'y' is set to 150
Group: WindowSize
Value: 'height' is set to 200
Value: 'width' is set to 150

*/
```

Para que el código anterior sea más legible, la expresión regular se crea en varias etapas. El primer patrón, contenido en la variable groupPattern, extrae categorías de la cadena de entrada. Busca los nombres entre corchetes, utilizando grupos que no son de captura para identificar los corchetes y un grupo con nombre para el nombre de la categoría.

La cadena valuePattern contiene un patrón que extrae la clave y el valor de una configuración, utilizando dos grupos con nombre, donde una línea contiene dos elementos separados por un signo igual (=). Esto se combina con el valor groupPattern en la expresión regular final de la variable pattern. Esto agrega el elemento condicional, que especifica que se debe usar el

primer patrón si se encuentra una cadena entre corchetes y que el segundo patrón debe coincidir de otra manera.

A continuación se muestra el patrón completo de expresión regular. Puedes ver que las expresiones regulares condicionales pueden volverse complejas y difíciles de leer muy rápidamente. El problema aumenta cuando se utilizan más condiciones. Siempre que sea posible, es mejor utilizar enfoques alternativos y más simples.

```
(? (^\[)(?:^\[)(?<Group>.+)(?:\]$)|(?:^) (?<Clave>.+)=(?<Valor>.+)(?:$))
```

1.11 Las sustituciones de expresiones regulares

Las sustituciones permiten que los elementos de una cadena de entrada coincidan y se reemplacen con texto alternativo. El patrón de reemplazo puede incluir caracteres literales y elementos de la coincidencia original.

1.11.1 Sustituciones

Una característica muy poderosa del motor de expresiones regulares es la funcionalidad de búsqueda y reemplazo proporcionada por las sustituciones. Te permiten realizar una coincidencia usando todos los caracteres de patrón que ya hemos visto en el tutorial. En lugar de simplemente devolver las coincidencias, se reemplazan con otro texto y se devuelve la cadena actualizada.

La cadena de reemplazo puede ser texto literal en casos simples. Para operaciones más complejas, puede incluir información de la cadena de entrada en la sustitución, incluida la coincidencia completa o el contenido de un grupo capturado.

1.11.2 Sustitución simple

Para realizar una sustitución, se usa el método Replace de la clase Regex, en lugar del método Match que hemos visto en artículos anteriores. Este método es similar a Match, excepto que incluye un parámetro de cadena adicional para recibir el valor de reemplazo.

Para demostrarlo, intente ejecutar el siguiente código. La llamada a Replace incluye tres argumentos. La primera es la cadena de entrada, que incluye dos números que podrían ser datos de la tarjeta de crédito. El segundo es el patrón a coincidir, que busca cuatro grupos de cuatro dígitos numéricos. El tercero proporciona la cadena de reemplazo. En este caso, el texto de reemplazo solo contiene caracteres literales; Reemplazan los números de las tarjetas de crédito con asteriscos.

```
string input = "Don't give away your credit card number, "
             + "be it 1111 1111 1111 1111 or 8493 2349 5173 8495.";

string find = @"\d\d\d\d \d\d\d\d \d\d\d\d \d\d\d\d";
string replace = "**** **** **** ****";

string result = Regex.Replace(input, find, replace);

Console.WriteLine(result);

/* OUTPUT
Don't give away your credit card number, be it **** **** **** **** or **** **** ****
****.
*/
```

1.11.3 Inclusión de grupos capturados en una cadena de reemplazo

Las sustituciones se vuelven mucho más eficaces cuando se incluyen elementos de la cadena de entrada en el reemplazo. Un enfoque común es copiar información de un grupo capturado en el texto de reemplazo. Puede hacerlo utilizando grupos capturados numerados o con nombre. En el caso de los grupos numerados, incluya un signo de dólar ($) y el número de grupo en la cadena de sustitución. Este marcador de posición se reemplazará con la información de la partida.

Para demostrarlo, ejecute el siguiente programa. En este caso, la cadena de entrada contiene una lista de nombres, cada uno en una nueva línea. Algunos de los nombres se proporcionan con el nombre antes del apellido, mientras que otros tienen el apellido primero, separado del nombre con una coma y un espacio.

El código coincide con los nombres en los que el apellido aparece antes del nombre. El apellido se incluye en el primer grupo capturado y el nombre en el segundo. Todas las coincidencias se reemplazan usando el patrón "$2 $1". Esto cambia el orden de los nombres y elimina la coma.

```
string input = "Bob Smith\n"
             + "Green, Mel\n"
             + "Sam Jones\n"
             + "Black, Liz\n"
             + "White, Tim\n";

string find = @"^(\w+), (\w+)$";
string replace = "$2 $1";

string result = Regex.Replace(input, find, replace, RegexOptions.Multiline);

Console.WriteLine(result);

/* OUTPUT

Bob Smith
Mel Green
Sam Jones
Liz Black
Tim White

*/
```

Si está utilizando grupos con nombre, puede usar el nombre en lugar del número después del signo de dólar si lo rodea con llaves. El siguiente ejemplo es funcionalmente equivalente al anterior, pero utiliza grupos con nombre.

```
string input = "Bob Smith\n"
             + "Green, Mel\n"
             + "Sam Jones\n"
             + "Black, Liz\n"
             + "White, Tim";

string find = @"^(?<Surname>\w+), (?<Forename>\w+)$";
string replace = "${Forename} ${Surname}";

string result = Regex.Replace(input, find, replace, RegexOptions.Multiline);

Console.WriteLine(result);

/* OUTPUT

Bob Smith
Mel Green
Sam Jones
Liz Black
Tim White

*/
```

Nota: Si la cadena de reemplazo incluye dígitos numéricos, pueden volverse ambiguos si aparecen junto a un grupo numerado. Puede rodear el número de grupo con llaves para evitar confusiones. Por ejemplo, "$11" coincidiría con el grupo 11 si está presente, pero "${1}1" coincidirá con el grupo uno y lo seguirá con el carácter '1' en la sustitución.

Otra opción es incluir los detalles del último grupo capturado en la cadena de reemplazo, independientemente de su nombre o número. Para ello, incluya el marcador de posición, "$+", como en el siguiente ejemplo:

```
string input = "Bob Smith\n"
             + "Green, Mel\n"
             + "Sam Jones\n"
             + "Black, Liz\n"
             + "White, Tim";

string find = @"^(?<Surname>\w+), (?<Forename>\w+)$";
string replace = "$+ ${Surname}";

string result = Regex.Replace(input, find, replace, RegexOptions.Multiline);

Console.WriteLine(result);

/* OUTPUT

Bob Smith
Mel Green
Sam Jones
Liz Black
Tim White

*/
```

1.12 Sustituciones de expresiones regulares

Se examinan las sustituciones. Estas permiten que los elementos de una cadena de entrada coincidan y se reemplacen con texto alternativo. El patrón de reemplazo puede incluir caracteres literales y elementos de la coincidencia original.

1.12.1 Incluir toda la coincidencia en una cadena de reemplazo

Si desea incluir toda la coincidencia en el texto de reemplazo, puede usar el marcador de posición, "$&". Esto es útil cuando desea agregar texto antes o después de la coincidencia. Por ejemplo, el código siguiente busca tiempos en la cadena de entrada. El texto de reemplazo agrega corchetes alrededor de esos tiempos coincidentes.

```
string input = "9:00 Start\n"
             + "10:30 Coffee\n"
             + "10:45 Session 2\n"
             + "12:30 Lunch";

string find = @"\d{1,2}:\d{2}";
string replace = "[$&]";

string result = Regex.Replace(input, find, replace);

Console.WriteLine(result);

/* OUTPUT

[9:00] Start
[10:30] Coffee
[10:45] Session 2
[12:30] Lunch

*/
```

1.12.2 Incluir signos de dólar en una cadena de reemplazo

El signo de dólar se utiliza para todas las sustituciones. Esto significa que si desea incluir un signo de dólar literal en el texto de reemplazo, debe omitirlo. Para ello, utilice dos signos de dólar adyacentes para incluir un solo símbolo en el reemplazo.

En el último ejemplo se ajusta el texto de un menú de bebidas. Busca valores que parecen ser precios y los prefija con un signo de dólar. Los dos primeros símbolos de dólar de la variable replace generan un único símbolo en el resultado. El tercero es parte del marcador de posición que incluye la coincidencia completa en la sustitución.

```
string input = "Tea 1.00\n"
             + "Coffee 1.30\n"
             + "Cappucino 2.15\n"
             + "Latte 2.15";

string find = @"\d+.\d{2}";
string replace = "$$$&";

string result = Regex.Replace(input, find, replace);

Console.WriteLine(result);

/* OUTPUT
Tea $1.00
Coffee $1.30
Cappucino $2.15
Latte $2.15
*/
```

1.13 Categorías y bloques Unicode de expresiones regulares

En este artículo se examina la coincidencia de caracteres de categorías generales y bloques de puntos de código Unicode específicos.

1.13.1 Unicode

Unicode proporciona un sistema de codificación estándar de la industria para caracteres. A diferencia de los sistemas más simples, como ASCII, Unicode le permite representar letras, números, espacios en blanco, signos de puntuación y otros símbolos para muchos idiomas diferentes, tanto modernos como históricos. El estándar define códigos para más de 120.000 símbolos o puntos de código diferentes.

1.13.2 Categorías generales de Unicode

Además de definir el símbolo que se mostrará o imprimirá, Unicode agrega propiedades adicionales para cada punto de código. Una de estas propiedades se conoce como la categoría general. Esta propiedad se puede utilizar para organizar caracteres en grupos y subgrupos que no están necesariamente vinculados al idioma del conjunto de caracteres. Por ejemplo, las letras mayúsculas siempre están contenidas en la categoría "Lu" y la categoría "Sm" se aplica a los puntos de código que representan símbolos matemáticos. Hay muchas categorías generales disponibles y los puntos de código individuales se pueden vincular a más de una categoría.

1.13.3 Bloques Unicode

Otra de las propiedades de los puntos de código Unicode organiza los caracteres en bloques contiguos. A diferencia de las categorías generales, los bloques tienen nombres únicos que no se superponen; Cualquier punto de código único solo puede aparecer en un bloque. Los bloques de código de ejemplo incluyen "árabe", "bengalí" y "mongol".

1.13.4 Coincidencia de categorías y bloques generales

Al utilizar expresiones regulares, puede hacer coincidir los caracteres en función de las categorías generales que tienen o del bloque en el que aparecen. Para hacer coincidir un solo carácter en cualquiera de las agrupaciones, utilice el patrón, "/p", seguido del nombre de la categoría o bloque entre llaves. Por ejemplo, para hacer coincidir símbolos matemáticos, puede usar "/p{Sm}", como en el siguiente ejemplo:

```
string input = "5+5=10";

foreach (Match match in Regex.Matches(input, @"\p{Sm}"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched '+' at index 1
Matched '=' at index 3

*/
```

Al igual que con las clases de caracteres abreviados, puede negar una coincidencia poniendo en mayúscula la letra 'p'. Para demostrarlo, intente ejecutar el código de ejemplo modificado a continuación, que coincide con todos los caracteres que no existen en la categoría de símbolos matemáticos.

```
string input = "5+5=10";

foreach (Match match in Regex.Matches(input, @"\P{Sm}"))
{
    Console.WriteLine("Matched '{0}' at index {1}",
                      match.Value, match.Index);
}

/* OUTPUT
Matched '5' at index 0
Matched '5' at index 2
Matched '1' at index 4
Matched '0' at index 5
*/
```

1.14 Opciones en línea de expresiones regulares

Se examinan las opciones que se pueden aplicar a un patrón mediante caracteres dentro de una expresión regular. Hay cinco opciones de este tipo, que se pueden aplicar a todo o parte de un patrón.

1.14.1 Opciones de expresión regular

Hay una serie de opciones que puede aplicar a las expresiones regulares para modificar la forma en que se hacen coincidir los patrones. Las opciones se pueden especificar en línea, mediante caracteres dentro de una expresión regular o mediante una enumeración que se pasa a un parámetro de uno de los métodos de la clase Regex. Por ejemplo, en el artículo que describe los anclajes de expresiones regulares, usamos un parámetro para especificar que la coincidencia debe aplicarse a una cadena de varias líneas. En este artículo, veremos las opciones en línea.

Hay cinco opciones que puede incluir dentro de una expresión regular. Cada uno está representado por una sola letra. Son los siguientes:

- i. Especifica que la coincidencia no debe distinguir entre mayúsculas y minúsculas. Con esta opción habilitada, las letras mayúsculas coincidirán cuando se busquen elementos en minúsculas y viceversa.

- m. Habilita la coincidencia de varias líneas. Cuando se habilitan, los anclajes ^ y $ coinciden con el inicio y el final de una línea. Sin la opción, coinciden con el inicio y el final de toda la cadena.

- n. Especifica que los grupos no deben capturarse a menos que se les asigne un nombre explícito.

- Esta opción permite la coincidencia de una sola línea. En este modo, el carácter comodín (.) coincide con cualquier carácter. Cuando no está habilitado, el comodín coincide con cualquier carácter, excepto con los saltos de línea.

- x. La aplicación de esta opción hace que se omitan los espacios en blanco en el patrón de expresión regular a menos que se escape. Esto puede hacer que las expresiones regulares complejas sean más fáciles de leer.

1.14.2 Aplicación de opciones en línea

Hay dos formas de aplicar una opción en línea. Para demostrarlo, primero considere el siguiente código de ejemplo. Busca el texto "Apple", precedido por cero o más caracteres de

palabra. Como la expresión regular distingue entre mayúsculas y minúsculas, coincide con "Apple" pero no con "Pineapple".

```
string input = "Apple, Pineapple, Orange";

foreach (Match match in Regex.Matches(input, @"\w*Apple"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match, match.Index);
}

/* OUTPUT

Matched 'Apple' at index 0

*/
```

La primera forma de aplicar una opción es colocarla en la posición dentro de la expresión regular en la que desea que comience a surtir efecto. Una vez aplicado, se utiliza hasta que se llega al final del patrón, o hasta que se deshabilita con otro código en línea.

Para aplicar una opción de esta manera, prefíjela con un signo de interrogación y rodee ambos caracteres entre paréntesis. Por ejemplo, "(?i)" desactivaría la distinción entre mayúsculas y minúsculas, como en el siguiente código de ejemplo:

```
string input = "Apple, Pineapple, Orange";

foreach (Match match in Regex.Matches(input, @"\w*(?i)Apple"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match, match.Index);
}

/* OUTPUT
Matched 'Apple' at index 0
Matched 'Pineapple' at index 7
*/
```

También puede aplicar opciones en línea a un solo grupo. Para ello, prepárelos con un signo de interrogación y sepárelos de los caracteres capturadores del grupo con dos puntos (:). En el ejemplo siguiente se aplica la opción que no distingue entre mayúsculas y minúsculas al grupo que contiene el patrón, "Apple":

```
string input = "Apple, Pineapple, Orange";

foreach (Match match in Regex.Matches(input, @"\w*(?i:Apple)"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match, match.Index);
}

/* OUTPUT
Matched 'Apple' at index 0
Matched 'Pineapple' at index 7
*/
```

1.14.3 Eliminación (Borrado) de opciones en línea

A menudo, deberá deshabilitar una opción después de habilitarla. Puede eliminar una opción con la misma sintaxis, pero con un signo menos (-) antes de la letra de la opción. En el ejemplo siguiente se muestra un patrón con la opción que no distingue entre mayúsculas y minúsculas activada primero y luego desactivada.

```
string input = "Apple, Pineapple, Orange";

foreach (Match match in Regex.Matches(input, @"(?i)\w*(?-i)Apple"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match, match.Index);
}

/* OUTPUT
Matched 'Apple' at index 0
*/
```

1.14.4 Combinación de opciones en línea

Las opciones en línea se pueden aplicar en grupos especificando más de una letra a la vez; No es necesario incluir cada opción individualmente. Por ejemplo, en el ejemplo de código final se aplican dos opciones. La 'i' indica que no distingue entre mayúsculas y minúsculas y la 'x' especifica que se debe ignorar el espacio en el patrón.

```
string input = "Apple, Pineapple, Orange";

foreach (Match match in Regex.Matches(input, @"(?ix)\w* Apple"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match, match.Index);
}

/* OUTPUT
Matched 'Apple' at index 0
Matched 'Pineapple' at index 7
*/
```

Nota: Puede activar y desactivar las opciones al mismo tiempo. Las opciones para habilitar deben aparecer antes de cualquier signo menos y las que se deshabilitan deben aparecer después de uno. Por ejemplo, (?i-mx) habilitará la indistinción entre mayúsculas y minúsculas y deshabilitará el modo multilínea e ignorará los espacios en blanco sin escape.

1.15 Opciones de Expresiones Regulares

Este artículo analiza el uso de la enumeración RegexOptions.

1.15.1 RegexOptions

En el artículo anterior del tutorial, vimos las opciones en línea que puede aplicar dentro de una expresión regular. Las cinco opciones disponibles le permiten modificar el comportamiento de coincidencia para un patrón completo, un grupo o desde cualquier punto de partida dentro de la cadena.

Las mismas cinco opciones se pueden aplicar sin usar caracteres insertados, pasando un valor de la enumeración RegexOptions al método que se usa para hacer coincidir el texto o realizar sustituciones. Con este enfoque, se aplican las opciones a todo el patrón, excepto cuando las opciones insertadas invalidan el valor de RegexOptions.

Además de las cinco opciones que vimos anteriormente, hay varios valores que solo están disponibles mediante la enumeración. Estos le permiten aplicar opciones que no están permitidas con caracteres en línea. Proporcionan características como invertir el orden de búsqueda de la operación, omitir la referencia cultural actual del usuario y compilar expresiones regulares para mejorar el rendimiento.

1.15.2 Opciones básicas

Las opciones básicas funcionan de la misma manera que cuando se aplican con caracteres en línea. Se representan mediante las siguientes constantes en la enumeración RegexOptions:

- **None (Ninguno)**. La configuración predeterminada que se utiliza cuando no se proporciona ninguna opción.
- **IgnoreCase**. Esta opción funciona de la misma manera que la opción en línea 'i'. Hace que se omitan las mayúsculas y minúsculas de los caracteres individuales de la cadena de entrada.
- **Multiline (Multilínea)**. Esta opción especifica que la cadena de entrada debe tratarse como texto de varias líneas, lo que afecta al funcionamiento de los anclajes de inicio y fin de línea. Es el equivalente a la opción en línea 'm'.
- **Singleline (Línea única)**. Al igual que con la opción 's', esta constante indica que la cadena de entrada debe considerarse como una sola línea de texto. Esto cambia el comportamiento de los anclajes de inicio y final de línea.

- **ExplicitCapture (Captura explícita)**. Esta opción impide la captura de grupos sin nombre. Su comportamiento es similar al de la opción 'n'.

- **IgnorePatternWhitespace**. Este valor, el equivalente de la opción 'x' en línea, indica que el espacio en blanco en la expresión regular debe ignorarse a menos que se escape.

Nota: La enumeración RegexOptions se define como un campo de bits mediante el atributo Flags, por lo que puede combinar varias opciones mediante operadores lógicos bit a bit.

1.15.3 Aplicación de opciones

Para aplicar las opciones anteriores, pase un valor RegexOptions al parámetro options del método del motor de expresiones regulares que desee usar. En el código siguiente se muestra esto con el método Match. La primera llamada a Match no proporciona ninguna opción, por lo que el método usa el valor RegexOptions.Default. La segunda llamada especifica que la operación debe omitir el uso de mayúsculas y minúsculas, por lo que se hacen coincidir más palabras.

```
string input = "Banana, banana, BANANA!";

Console.WriteLine("Default Matching");
foreach (Match match in Regex.Matches(input, "banana"))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

Console.WriteLine("\nCase-Insensitive Matching");
foreach (Match match in Regex.Matches(input, "banana", RegexOptions.IgnoreCase))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}


/* OUTPUT

Default Matching
Matched 'banana' at index 8

Case-Insensitive Matching
Matched 'Banana' at index 0
Matched 'banana' at index 8
Matched 'BANANA' at index 16

*/
```

Puede utilizar las opciones de forma similar con el método Replace. Por ejemplo, el código siguiente utiliza la coincidencia que distingue entre mayúsculas y minúsculas y que no distingue entre mayúsculas y minúsculas durante una operación de sustitución. La segunda llamada, que ignora el caso, rodea las tres palabras entre paréntesis.

```
string input = "Banana, banana, BANANA!";

Console.WriteLine("Default Substitution");
Console.WriteLine(Regex.Replace(input, "banana", "[$0]"));

Console.WriteLine("\nCase-Insensitive Substitution");
Console.WriteLine(Regex.Replace(input, "banana", "[$0]", RegexOptions.IgnoreCase));


/* OUTPUT

Default Substitution
Banana, [banana], BANANA!

Case-Insensitive Substitution
[Banana], [banana], [BANANA]!

*/
```

1.15.4 CultureInvariant (Opción)

La realización de una operación de coincidencia o sustitución que no distingue entre mayúsculas y minúsculas puede tener efectos inesperados. De forma predeterminada, el motor de expresiones regulares usa la referencia cultural actual del usuario. Esto significa que el motor es consciente de los vínculos entre las letras minúsculas y mayúsculas en el idioma del usuario. Sin embargo, si coincide con texto que tiene una referencia cultural fija, como palabras clave en un lenguaje de programación, esto puede dar coincidencias falsas cuando la referencia cultural del usuario incluye diferentes convenciones de uso de mayúsculas y minúsculas.

Un ejemplo bien conocido de un problema de envoltura cultural es el problema de la I turca. Mientras que las culturas inglesas ven la "i" como la versión mayúscula de la "i", la cultura turca no lo hace. En su lugar, hay cuatro caracteres de letra 'i'; 'ı' es la versión minúscula de 'I' y 'i' se escribe en mayúsculas a 'İ'. Si coincide con términos como "if" o "IDisposable", esto puede causar problemas.

Puede evitar algunos de estos problemas mediante la opción CultureInvariant. Esto impide que el motor de expresiones regulares use las convenciones culturales del usuario y, en su lugar, use la referencia cultural invariable bien documentada.

1.15.5 Opción ECMAScript

ECMAScript es una definición de lenguaje de scripting estandarizada internacionalmente que se implementa en lenguajes como JavaScript y ActionScript. Incluye un motor de expresiones regulares. Sin embargo, la funcionalidad de ese motor no es idéntica a la de las expresiones regulares de .NET Framework. Si necesita imitar el motor ECMAScript, aplique la opción ECMAScript de la enumeración RegexOptions.

La opción modifica varios comportamientos coincidentes. Por ejemplo, Unicode no es compatible, por lo que las clases de caracteres cambian. Además, no todas las demás opciones se pueden combinar con el valor ECMAScript. Cuando se incluye, solo puede agregar las opciones IgnoreCase, Multiline y Compilado. Si usa cualquier otro valor, la operación producirá una excepción.

1.15.6 Opción RightToLeft

La opción RightToLeft invierte la dirección del orden de búsqueda al aplicar un patrón. En lugar de buscar primero la coincidencia situada más a la izquierda, la opción busca la coincidencia situada más a la derecha antes que las que aparecen más a la izquierda. El funcionamiento del patrón de la expresión regular no se ve afectado.

Intente ejecutar el siguiente código de ejemplo para demostrarlo. Los resultados muestran que las coincidencias se encuentran de derecha a izquierda. Tenga en cuenta que la opción RightToLeft se combina con la opción IgnoreCase mediante un operador lógico OR.

```
string input = "Banana, banana, BANANA!";

foreach (Match match in Regex.Matches(
    input, "banana", RegexOptions.IgnoreCase | RegexOptions.RightToLeft))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched 'BANANA' at index 16
Matched 'banana' at index 8
Matched 'Banana' at index 0

*/
```

1.15.7 Opción Compilada

Normalmente, cuando se hace coincidir una expresión regular, el patrón se convierte en un conjunto de códigos de operación personalizados que se interpretan cada vez que se utiliza la expresión. Si usa repetidamente el mismo patrón, esto puede agregar una sobrecarga que reduce el rendimiento de las operaciones.

Para mejorar la velocidad de coincidencia, puede aplicar la opción Compilado. La primera vez que se utiliza una expresión regular compilada, se compila en lenguaje intermedio (MSIL). Esto da una compensación; La compilación es más lenta que la conversión a códigos de operación personalizados, pero la expresión se puede procesar más rápidamente.

Solo debe compilar expresiones regulares que se usarán muchas veces, ya que el proceso de inicialización tarda más que para los patrones que se interpretan. Si utiliza los métodos estáticos de la clase Regex, las expresiones compiladas se almacenan en caché para que se puedan reutilizar. Si utiliza métodos de instancia, es importante tener en cuenta que la versión compilada se pierde cuando la instancia sale del ámbito.

1.16 Comentarios de expresiones regulares

Se describe cómo se pueden agregar comentarios a un patrón de expresión regular. Esto permite que los patrones complejos incluyan texto explicativo.

1.16.1 Comentarios

Al escribir código, suele ser recomendable minimizar el uso de comentarios que describan la funcionalidad. Cuando el código es difícil de entender, es mejor refactorizarlo para que sea más fácil de mantener, en lugar de agregar un comentario. Si no lo hace, es posible que el código se actualice en el futuro sin cambiar el comentario. Esto lleva a comentarios desactualizados que pueden causar confusión.

Cuando se trabaja con expresiones regulares, el lenguaje es tan conciso que la creación de patrones legibles se vuelve increíblemente difícil o incluso imposible. En tales situaciones, un comentario bien redactado puede ser esencial. Puede agregar comentarios cerca de la expresión regular utilizando la sintaxis del lenguaje que está utilizando para llamar al motor de expresiones regulares. También puedes incluirlos dentro del propio patrón.

Hay dos formas de agregar un comentario a una expresión regular. La primera es crear un grupo específicamente para el comentario. Al igual que con otros grupos, esto se define con un par de paréntesis. El comentario se coloca entre paréntesis y va precedido de un signo de interrogación y un símbolo de almohadilla. Por lo tanto, la sintaxis de un comentario es:

```
(?# Comment)
```

En el código siguiente se muestra esta sintaxis. En el ejemplo se buscan hipervínculos en un documento HTML. Los tres comentarios muestran las partes de la expresión regular que localizan las etiquetas de apertura y cierre del hipervínculo y la información que se mostraría en un navegador web.

```
string input = "For more information use the "
             + "<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a> "
             + "or check the list of "
             + "<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently "
             + "asked questions</a>.";

string pattern = "(<a href=')(.*?)('>)(?#Opening tag)(.*?)(?#Display)(</a>)(?#Closing
tag)";

foreach (Match match in Regex.Matches(input, pattern))
{
    Console.WriteLine("Matched '{0}'", match.Value);
}

/* OUTPUT

Matched '<a href='http://www.blackwasp.co.uk/Contact.aspx'>contact form</a>'
Matched '<a href='http://www.blackwasp.co.uk/FAQ.aspx'>frequently asked
questions</a>'

*/
```

Si está usando la opción para omitir el espacio en blanco sin escape dentro de una expresión regular, usando el código 'x' o la opción IgnorePatternWhitespace, puede usar una segunda sintaxis de comentario. En este modo, los comentarios van precedidos de un símbolo de almohadilla (#). El comentario se extiende hasta el final de la línea.

```
# Comment
```

En el ejemplo siguiente se incluye una expresión regular que extrae direcciones IP. Incluye un comentario definido solo por el símbolo de almohadilla. Si quitara el argumento RegexOptions.IgnorePatternWhitespace de la llamada a Match, no se encontrarían coincidencias porque el comentario se vería como parte del patrón.

```
string input = @"Some of these are valid IP addresses:

0.1.2.3
10.20.30.40
100.200.300.400
192.0.0.256
192.0.0.255
192.0.0.249
10.1.30.150
1.01.1.1
99.00.99.00";

string pattern = @"((2[0-4]\d|25[0-5]|1\d{2}|[1-9]\d|\d)\.){3}"
               + @"(2[0-4]\d|25[0-5]|1\d{2}|[1-9]\d|\d) #Extracts IP addresses";

foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnorePatternWhitespace))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched '0.1.2.3' at index 41
Matched '10.20.30.40' at index 50
Matched '192.0.0.25' at index 80
Matched '192.0.0.255' at index 93
Matched '192.0.0.249' at index 106
Matched '10.1.30.150' at index 119

*/
```

1.17 División de cadenas con expresiones regulares

Se examina el método Split. De forma similar al método Split de la clase de cadena, este miembro divide una cadena en una matriz de elementos, en función de un delimitador.

1.17.1 Regex.Split

Anteriormente en el tutorial de expresiones regulares vimos dos métodos clave. Coincidencias le permite encontrar las partes de una cadena de entrada que coinciden con un patrón y devolverlas. Reemplazar realiza sustituciones, lo que le permite encontrar coincidencias y reemplazarlas con texto alternativo. En este artículo, comenzaremos a echar un vistazo a algunos otros miembros de la clase Regex. Comenzaremos con el método Split.

Al igual que con el método Split proporcionado por la clase string, la versión del motor de expresiones regulares toma una cadena de entrada y la divide en una matriz de cadenas más cortas en función de la posición de los caracteres delimitadores. Para la clase de cadena, el delimitador es un solo carácter o una cadena. Con la clase Regex, los delimitadores se encuentran mediante una expresión regular.

La forma básica de llamar a Split es mediante dos parámetros de cadena. El primero contiene la cadena de entrada y el segundo contiene la expresión regular que debe coincidir. Cada vez que se encuentra una coincidencia para el patrón, marca el final de una subcadena que se incluirá en la matriz resultante. Si el patrón coincide al principio de la cadena de entrada, o si el patrón coincide dos veces seguidas, la matriz incluirá una o más cadenas vacías. La cadena final de la matriz devuelta contiene el texto que sigue a la última coincidencia.

Para demostrarlo, intente ejecutar el siguiente código de ejemplo. La expresión regular de la variable splitOn busca uno o varios caracteres de un grupo de caracteres, que busca espacios en blanco y algunos signos de puntuación. Esto divide el texto de entrada en una matriz de palabras individuales.

```
string input = "Next day he was drunk, and he went to Judge Thatcher's and
bullyragged "
             + "him, and tried to make him give up the money; but he couldn't, and
then "
             + "he swore he'd make the law force him.";

string splitOn = @"[\s.,;]+";

string[] words = Regex.Split(input, splitOn);

foreach (var word in words)
{
    Console.WriteLine("'" + word + "'");
}

/* OUTPUT

'Next'
'day'
'he'
'was'
'drunk'
'and'
'he'
'went'
'to'
'Judge'
'Thatcher's'
'and'
'bullyragged'
'him'
'and'
'tried'
'to'
'make'
'him'
'give'
'up'
'the'
'money'
'but'
'he'
'couldn't'
'and'
'then'
'he'
'swore'
'he'd'
'make'
```

```
'the'
'law'
'force'
'him'
''

*/
```

1.18 Texto de escape para expresiones regulares

Este artículo considera el proceso de escape y des escape del texto.

1.18.1 Texto de escape

A veces, querrá crear una expresión regular basada en la entrada del usuario. Por ejemplo, es posible que desee realizar una búsqueda en varios documentos para encontrar los que contienen el texto introducido. Esto se puede lograr con un patrón que contenga la entrada del usuario y algunos caracteres comodín y cuantificadores.

En tal situación, es importante que la información introducida se trate como texto literal. Sin embargo, el usuario puede usar elementos que el motor de expresiones regulares reconoce y trata como caracteres de control, lo que provoca resultados inesperados cuando se coincide con el patrón. Para evitar esto, puede escapar de los caracteres de control.

Para que no necesite escribir su propio código para escapar de los caracteres, la clase Regex incluye un método estático para este propósito. Puede llamar al método Escape y pasar la cadena a process a su parámetro. El texto de escape se devuelve como una nueva cadena.

Para demostrarlo, intente ejecutar el código siguiente. Como se muestra en el resultado, la coma y el punto (punto) se escapan con barras invertidas.

```
string input = "Hello, world.";
string escaped = Regex.Escape(input);

Console.WriteLine(escaped);

// Outputs: "Hello,\ world\."
```

1.18.2 Texto sin escape

Puede revertir el proceso con el método Unescape. Busca caracteres de escape y los reemplaza con el texto original. En la mayoría de los casos, esto funciona perfectamente. Sin embargo, como algunas secuencias de caracteres pueden ser ambiguas, hay situaciones en las que la cadena sin escape no es idéntica a la original. Además, si la cadena para anular el escape no pudo haber sido creada por Escape, el método puede producir una excepción.

En el ejemplo siguiente se escapa una cadena y, a continuación, se anula el escape. El resultado muestra que el texto original y la versión sin escape coinciden.

```
string input = "Hello, world.";
string escaped = Regex.Escape(input);
string unescaped = Regex.Unescape(escaped);

Console.WriteLine(unescaped);

// Outputs: "Hello, world."
```

1.19 Métodos de instancia de expresiones regulares

Se examina una alternativa a los métodos estáticos empleados anteriormente en la serie. Los métodos de instancia proporcionan la misma funcionalidad, pero con diferentes características de almacenamiento en caché.

1.19.1 Clase Regex

A lo largo del tutorial de expresiones regulares, hemos utilizado métodos estáticos para realizar operaciones de coincidencia, sustituciones y división de cadenas. También puede ejecutar la misma funcionalidad mediante instancias de la clase Regex. Los resultados son exactamente los mismos, pero el uso y el almacenamiento en caché de los patrones compilados varían un poco.

Para crear una instancia de la clase Regex, llame a uno de varios constructores. Cada uno requiere que proporcione una cadena que contenga la expresión regular que se usará para la coincidencia. También puede agregar un segundo parámetro del tipo RegexOptions que especifique las opciones que debe aplicar Regex. Nota: Una vez instanciados, los objetos Regex son inmutables; no se puede modificar ni el patrón ni las opciones.

```
Regex pattern1 = new Regex("[A-Z]{1,3}[0-9]{1,3}");
Regex pattern2 = new Regex("[A-Z]{1,3}[0-9]{1,3}", RegexOptions.IgnoreCase);
```

Los métodos de instancia para la coincidencia, la sustitución y la división utilizan los mismos nombres que sus homólogos estáticos. Como el objeto Regex ya define la expresión regular y las opciones, estos argumentos se omiten.

El siguiente código de ejemplo proporciona la misma funcionalidad que el ejemplo de coincidencia de direcciones IP del artículo en el que se describen los cuantificadores. La única diferencia es el uso de una instancia Regex en lugar de llamadas a métodos estáticos.

```
string input = @"Some of these are valid IP addresses:

0.1.2.3
10.20.30.40
100.200.300.400
192.0.0.256
192.0.0.255
192.0.0.249
10.1.30.150
1.01.1.1
99.00.99.00";

string ipAddress =
    @"((2[0-4]\d|25[0-5]|1\d{2}|[1-9]\d|\d)\.){3}(2[0-4]\d|25[0-5]|1\d{2}|[1-9]\d|\
d)";
Regex ipRegex = new Regex(ipAddress);

foreach (Match match in ipRegex.Matches(input))
{
    Console.WriteLine("Matched '{0}' at index {1}", match.Value, match.Index);
}

/* OUTPUT

Matched '0.1.2.3' at index 41
Matched '10.20.30.40' at index 50
Matched '192.0.0.25' at index 80
Matched '192.0.0.255' at index 93
Matched '192.0.0.249' at index 106
Matched '10.1.30.150' at index 119

*/
```

1.19.2 Almacenamiento en caché de expresiones regulares

Como se menciona en el artículo que describe las opciones de expresiones regulares, puede compilar las expresiones regulares o puede permitir que el motor las convierta en códigos de operación personalizados. Cuando se utilizan los métodos estáticos, las expresiones regulares procesadas se almacenan en caché de forma centralizada para que se puedan reutilizar más rápidamente.

Cuando se utilizan instancias de Regex, se produce la misma conversión o compilación, pero la expresión regular procesada se almacena en caché dentro de la instancia. Esto es importante porque cuando el objeto sale del ámbito, se pierde la versión almacenada en caché. Si crea varios objetos para el mismo patrón, la memoria caché no se utilizará y cada uno de ellos deberá compilarse individualmente. Debe tener esto en cuenta, especialmente si usa

instancias Regex dentro de bucles, donde debe crear la instancia fuera del bucle y reutilizarla siempre que sea posible.

Esta diferencia en el comportamiento del almacenamiento en caché puede llevarle a pensar que los métodos estáticos siempre ofrecen un mejor rendimiento. Sin embargo, debe tener en cuenta que el tamaño de la caché es limitado. De forma predeterminada, solo se almacenarán en caché quince expresiones regulares. Cuando sepa cuántos elementos necesita conservar, puede modificar el tamaño mediante la propiedad estática CacheSize. También puede leer esta propiedad para encontrar el tamaño actual de la memoria caché. Si usa un mayor número de expresiones regulares con métodos estáticos, los elementos de caché más antiguos se quitarán para dejar espacio para los más nuevos y es posible que sea necesario volver a compilar algunas llamadas repetidas. Este problema no existe cuando se utilizan métodos de instancia.

1.20 Bibliografía

- http://www.blackwasp.co.uk/RegexAnchors.aspx
- http://www.blackwasp.co.uk/RegexCharacterClasses.aspx
- http://www.blackwasp.co.uk/RegexCharacterEscapes.aspx
- http://www.blackwasp.co.uk/RegexMatches.aspx
- http://www.blackwasp.co.uk/RegularExpressions.aspx
- http://www.blackwasp.co.uk/RegexQuantifiers.aspx
- http://www.blackwasp.co.uk/RegexSubstitutions.aspx
-

por Richard Carr, publicado en http://www.blackwasp.co.uk/RegexGrouping.aspx