Artículo
Alberto Fuentes · 24 mayo, 2019 Lectura de 18 min

# Uso de expresiones regulares en ObjectScript

Al igual que con Pattern Matching, se pueden utilizar Expresiones Regulares para identificar patrones en textos en ObjectScript, sólo que con una potencia mucho mayor.

En este artículo se proporciona una breve introducción sobre las Expresiones Regulares y lo que puede hacerse con ellas en ObjectScript. La información que se proporciona aquí se basa en varias fuentes, entre las que destaca el libro *"Mastering Regular Expressions"* (Dominando las expresiones regulares) escrito por *Jeffrey Friedl* y, por supuesto, la documentación online de la plataforma. El procesamiento de textos utiliza patrones que algunas veces pueden ser complejos. Cuando utilizamos expresiones regulares, normalmente tenemos varias estructuras: el texto en el que buscamos los patrones, el patrón en sí mismo (la expresión regular) y las coincidencias (las partes del texto que coinciden con el patrón). Para que sea más sencillo distinguir entre estas estructuras, se utilizan las siguientes convenciones a lo largo de este documento:

Los ejemplos en el texto se muestran en monospace y sin comillas adicionales:

Esta es una "cadena de texto" en la cual deseamos encontrar "algo".

A menos que sean evidentes, las expresiones regulares que estén dentro del texto principal se visualizan en un fondo gris, como en el siguiente ejemplo: \".*?\".

Cuando sea necesario, las coincidencias se resaltarán mediante colores diferentes:

Esta es una "cadena de texto" en la cual deseamos encontrar "algo".

Los ejemplos de códigos que sean más grandes se mostrarán en cuadros:

```
set t="Esta es una ""cadena de texto"" en la cual queremos encontrar ""algo""."
set r="\"".*?\""
w $locate(t,r,,,tMatch)
```

# 1. Un poco de historia (y algunas curiosidades)

A principios de los años 40, los neurofisiólogos desarrollaron modelos para representar el sistema nervioso de los humanos. Unos años después, un matemático describió estos modelos mediante expresiones algebraicas que a las que llamó " conjuntos regulares ". La notación que se utilizó en estas expresiones algebraicas se denominó " expresiones regulares ".

En 1965, las expresiones regulares se mencionaron por primera vez en el contexto de la informática. Con *qed*, un editor que formó parte del sistema operativo UNIX, el uso de las expresiones regulares se extendió. Las siguientes versiones de ese editor proporcionan secuencias de comandos g/expresiones regulares/p (*global, expresión regular, imprimir*), las cuales realizan búsquedas de coincidencias de las expresiones regulares en todas las líneas del texto y muestran los resultados. Estas secuencias de comandos se convirtieron finalmente en la utilidad *grep*.

Hoy en día, existen varias implementaciones de expresiones regulares (RegEx) para muchos lenguajes de programación.

# 2. Regex 101

En esta sección se describen los componentes de las expresiones regulares, su evaluación y algunos de los motores disponibles. Los detalles de cómo utilizarlos se describen en la sección 4.

## 2.1. Componentes de las expresiones regulares

### 2.1.1. Metacaracteres

Los siguientes caracteres tienen un significado especial en las expresiones regulares.

. * + ? ( ) [ ] \ ^ $ |

Para utilizarlos como valores literales, deben utilizarse utilizando la contrabarra \. También pueden definirse explícitamente secuencias de valores literales utilizando \Q <literal sequence> \E.

### 2.1.2. Valores literales

El texto normal y caracteres de escape se tratan como valores literales, por ejemplo:

| | |
|---|---|
| • abc | abc |
| • \f | salto de página |
| • \n | salto de línea |
| • \r | retorno de carro |
| • \r | tabulación |
| • \0+tres dígitos (por ejemplo, \0101) | Número octal.<br>El motor de RegEx que utiliza Caché / IRIS (ICU), es compatible con los números octales hasta el \0377 (255 en el sistema decimal). Al migrar expresiones regulares desde otro motor<br>hay que considerar cómo procesa los octales dicho motor. |
| • \x+dos dígitos (por ejemplo, \x41) | Número hexadecimal.<br>La biblioteca ICU cuenta con varias opciones para procesar los números hexadecimales, consulte los documentos de apoyo sobre ICU (los enlaces están disponibles en la sección 5.8) |

### 2.1.3. Anclas

Las anclas (*anchors*) permiten encontrar ciertos puntos en un texto/cadena, por ejemplo:

- \A Inicio de la cadena
- \Z Final de la cadena
- ^ Inicio del texto o línea
- $ Final de un texto o línea
- \b Límite de palabras
- \B No en un límite de palabras
- \< Inicio de una palabra
- \> Final de una palabra

Algunos motores de regex se comportan de forma diferente, por ejemplo, al definir lo que constituye exactamente una palabra y cuáles caracteres se consideran delimitadores de palabras.

## 2.1.4. Cuantificadores

Con los cuantificadores, puede definir qué tan frecuentemente el elemento anterior puede crear una coincidencia:

- {x} exactamente x número de veces
- {x,y} mínimo x, máximo y número de veces
- * 0 ó más, es equivalente a {0,}
- + 1 ó más, es equivalente a {1,}
- ? 0 ó 1

### Codicia

Se dice que los cuantificadores son *"codiciosos" (greedy)* ya que toman tantos caracteres como sea posible. Supongamos que tenemos la siguiente cadena de texto y queremos encontrar el texto que está entre las comillas:

Este es "un texto" con "cuatro comillas".

Debido a la naturaleza codiciosa de los selectores, la expresión regular \".*\" encontrará demasiado texto:

Este es "un texto" con "cuatro comillas".

En este ejemplo, la expresión regular .* pretende incluir tantos caracteres que se encuentren entre comillas como sea posible. Sin embargo, debido a la presencia del selector punto ( . ) también buscará las comillas, de modo que no obtendremos el resultado que deseamos.

Con algunos motores de regex (incluyendo el que utiliza Caché / IRIS) se puede controlar la codicia de los cuantificadores, al incluir un signo de interrogación en ellos. Entonces, la expresión regular \".*?\" ahora hace que coincidan las dos partes del texto que se encuentran entre las comillas, es decir, exactamente lo que buscábamos:

Este es "un texto" con "cuatro comillas".

## 2.1.5. Clases de caracteres (rangos)

Los corchetes se utilizan para definir los rangos o conjuntos de caracteres, por ejemplo, [a-zA-Z0-9] o [abcd] . En el lenguaje de las expresiones regulares esta notación se refiere a una clase de caracteres. Un rango coincide con los caracteres individuales, de modo que el orden de los caracteres dentro de la definición del rango no es importante: [dbac] devuelve tantas coincidencias como [abcd].

Para excluir un rango de caracteres, simplemente se coloca el símbolo ^ frente a la definición del rango (¡dentro de los corchetes!): [^abc] buscará todas las coincidencias excepto con a, b o c.

Algunos motores de regex proporcionan clases de caracteres previamente definidas (POSIX), por ejemplo:

- [:alnum:] [a-zA-z0-9]
- [:alpha:] [a-zA-Z]
- [:blank:] [ \t]
- …

## 2.1.6. Grupos

Los componentes de una expresión regular pueden agruparse utilizando un par de paréntesis. Esto resulta útil para aplicar cuantificadores a un grupo de selectores, así como para referirse a los grupos tanto desde las expresiones regulares como desde ObjectScript. Los grupos también pueden anidarse.

Las siguientes coincidencias en las cadenas regex consisten en un número de tres dígitos, seguidos por un guion, a continuación, tres pares de letras mayúsculas y un dígito, seguidas por un guion, y para finalizar el mismo número de tres dígitos como en la primera parte:

([0-9]{3})-([A-Z][[0-9]){3}-\1

El siguiente ejemplo muestra cómo utilizar las *referencias previas* para que coincidan no solamente la estructura, sino también el contenido: el punto de referencia (en morado) le indica al motor que busque el mismo número de tres dígitos al principio y al final de la cadena (en amarillo). El ejemplo también muestra cómo aplicar un cuantificador a las estructuras más complejas (en verde).

La expresión regular de arriba coincidiría con la siguiente cadena:

123-D1E2F3-123

Pero no coincidiría con las siguientes cadenas:

123-D1E2F3-456 (los últimos tres dígitos son diferentes de los primeros tres)

123-1DE2F3-123 (la parte central no consiste en tres letras/pares de dígitos)

123-D1E2-123 (la parte central incluye únicamente dos letras/pares de dígitos)

Los grupos también puede accederse utilizando los búferes de captura desde ObjectScript ( sección 4.5.1). Esta es una característica muy útil que permite buscar y extraer información al mismo tiempo.

## 2.1.7. Alternancia

Con el carácter de barra vertical se especifica alternancia entre opciones, por ejemplo, skyfall|done. Esto permite comparar expresiones más complejas, como las clases de caracteres que se describieron en la sección 3.1.5.

### 2.1.8. Referencias

Los referencias permiten consultar grupos previamente definidos (selectores dentro de los paréntesis). En el siguiente ejemplo se muestra una expresión regular que coincide con tres caracteres consecutivos, los cuales deben ser iguales:

([a-zA-Z]) \1 \1

Los puntos de referencia se especifican con \x, donde x representa la expresión x-ésima entre paréntesis.

### 2.1.9. Reglas de precedencia

1. [] antes de ()
2. , + y ? antes de una secuencia: ab es equivalente a a(b*), y no a (ab)*
3. Secuencia antes de una alternancia: ab¦c es equivalente a (ab)¦c, y no a a(b¦c)

## 2.2. Un poco de teoría

La evaluación de las expresiones regulares, por lo general, se realiza mediante la implementación de alguno de los dos siguientes métodos (las descripciones se han simplificado, en las referencias que se mencionan en el capítulo 5 se pueden encontrar más detalles):

1. Orientado por el texto (DFA – *Autómata finito determinista*) El motor de búsqueda avanza a través del texto de entrada, carácter por carácter, e intenta encontrar coincidencias en el texto recorrido hasta el momento. Cuando llega al final del texto que de entrada, declara que el proceso se realizó con éxito.
2. Orientado por las expresiones regulares (NFA – *Autómata finito no determinista*) El motor de búsqueda avanza a través de la expresión regular, token por token, e intenta aplicarla al texto. Cuando llega al último token (y encuentra todas las coincidencias), declara que el proceso se realizó con éxito.

El método 1 es determinista, su tiempo de ejecución depende únicamente de la longitud del texto introducido. El orden de los selectores en las expresiones regulares no influye en el tiempo de ejecución.

El método 2 no es determinista, el motor reproduce todas las combinaciones posibles de los selectores en la expresión regular, hasta que encuentre una coincidencia o se produzca algún error. Por ende, este método es particularmente lento cuando *no* encuentra una coincidencia (debido a que tiene que reproducir todas la combinaciones posibles). El orden de los selectores *influye* en el tiempo de ejecución. Sin embargo, este método le permite retroceder y capturar los búferes.

## 2.3. Motores de búsqueda

Existen muchos motores de regex disponibles, algunos ya están incorporados en los lenguajes de programación o en los sistemas operativos, otros son librerías que pueden utilizarse en casi cualquier parte. Aquí se muestran algunos motores de regex, los cuales se agruparon por el tipo de método de evaluación:

- DFA: grep, awk, lex
- NFA: Perl, Tcl, Python, Emacs, sed, vi, ICU

En la siguiente tabla se realiza una comparación de las características que están disponibles en regex, entre varias bibliotecas y en lenguajes de programación:

| | "+" quantifier | Negated character classes | Non-greedy quantifiers | Shy groups | Recursion | Look-ahead | Look-behind | Backreferences | >9 indexable captures | Directives | Conditionals | Atomic groups | Named capture | Comments | Embedded code | Unicode property support | Balancing groups | Variable-length look-behinds |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| .NET | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | Yes | Yes |
| Boost.Regex | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | No | No |
| Boost.Xpressive | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | No | No | No |
| CL-PPCRE | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Some | No | No |
| GLib/GRegex | Yes | ? | Yes | ? | No | ? | ? | ? | ? | Yes | Yes | Yes | Yes | Yes | No | Some | No | No |
| GNU grep | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | ? | Yes | Yes | ? | Yes | Yes | No | No | No | No |
| Haskell | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | ? | ? | ? | ? | ? | No | No | No | No |
| ICU Regex | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | No | Yes | No | Yes | No | No |
| Java | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | Some | No | No |
| JavaScript | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | No | No | No | No | No | No | No | No | No |
| JGsoft | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Some | No | Yes |
| Lua | Yes | Yes | Yes | No | No | No | No | Yes | No | No | No | No | No | No | No | No | No | No |
| OCaml | Yes | Yes | No | No | No | No | No | Yes | No | No | No | No | No | No | No | No | No | No |
| PCRE | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No |
| Perl | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No |
| PHP | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No | No |
| Python | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | No | No | No | No |
| Qt/QRegExp | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | No | No | No | No | No | No | No | No | No |
| R | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| RE2 | Yes | Yes | Yes | Yes | No | No | No | No | Yes | Yes | No | ? | Yes | No | No | Some | No | No |
| RGX | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | No | No |
| Ruby | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Some | No | No |
| Tcl | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | No | Yes | No | Yes | No | Yes | No | No |
| TRE | Yes | Yes | Yes | Yes | No | No | No | Yes | No | Yes | No | No | No | Yes | No | ? | No | No |
| Vim | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | No | Yes | No | Yes | No | No | No | No | No | Yes |
| XRegExp | Yes | Yes | Yes | Yes | No | Yes | No | Yes | Yes | Some | No | No | Yes | Yes | No | Yes | No | No |

Puede encontrar más información aquí: **https://en.wikipedia.org/wiki/Comparisonofregularexpressionengines**

# 3. RegEx y Caché

InterSystems Caché/IRIS utiliza la biblioteca ICU para buscar expresiones regulares, en los documentación online se describen muchas de estas características. El objetivo de las siguientes secciones es hacer una introducción rápida sobre cómo utilizarlas.

# 3.4. $match() y $locate()

En ObjectScript (COS), las funciones $match() y $locate() brindan acceso directo a la mayoría de las características de regex a las que se tiene acceso desde la biblioteca de ICU. $match(String, Regex) busca en la cadena de entrada un patrón de regex. Cuando la función encuentra una coincidencia devuelve 1, en caso contrario devuelve 0.

Ejemplos:

- w $match("baaacd",".*(a)\1\1.*") devuelve 1
- w $match("bbaacd",".*(a)\1\1.*") devuelve 0

$locate(String,Regex,Start,End,Value) busca en la cadena de entrada un patrón de regex, del mismo modo que lo hace $match(). Sin embargo, $locate() proporciona mayor control del proceso y también devuelve más información. En Start, se puede indicar a $locate en qué punto de la cadena de entrada debe comenzar la búsqueda. Cuando $locate() encuentra una coincidencia, regresa al punto donde se encuentra el primer carácter de la coincidencia y establece End para la siguiente posición del carácter después de la coincidencia. El contenido de la coincidencia es devuelto en Value.

Si $locate() no encuentra alguna coincidencia devuelve 0 y no tocará los contenidos de End y Value (si se especificaron). End y Value se pasan por referencia, por lo que hay que ser cuidadoso si utilizan continuamente (por ejemplo en bucles).

Ejemplo:

- w $locate("abcdexyz",".d.",1,e,x) devuelve 3, *e* se establece en 6, *x* se establece en "cde"

$locate() realiza la búsqueda de coincidencias en los patrones y puede devolver el contenido de la primera coincidencia, al mismo tiempo. Si necesita extraer el contenido de todas las coincidencias, puede llamar a $locate() continuamente mediante un bucle o puede utilizar los métodos que se proporcionan en %Regex.Matcher (siguiente sección).

# 3.5.% Regex.Matcher

%Regex.Matcher proporciona acceso a funciones de regex de la librería ICU del mismo modo que lo hacen las funciones $match() y $locate() pero además cuenta con características avanzadas. En las siguientes secciones se retomará el concepto de los búferes de captura, se analizará la posibilidad de sustituir cadenas con expresiones regulares y las formas para controlar el comportamiento del tiempo de ejecución.

## 3.5.1. Búferes de captura

Como ya hemos visto con los grupos, las referencias y $locate(), las expresiones regulares le permiten buscar simultáneamente los patrones en el texto y devolver el contenido con las coincidencias. Esto funciona al colocar las partes del patrón que desea extraer entre un par de paréntesis (agrupación). Si la búsqueda de coincidencias es exitosa, los búferes de captura tendrán el contenido de todos los grupos en los que se hayan encontrado coincidencias. Hay que tener en cuenta que este procedimiento es un poco diferente de lo que $locate() proporciona con sus parámetros de valores: $locate() devuelve el contenido de todas la coincidencias en sí, mientras que los búferes de captura le dan acceso a algunas *partes* de las coincidencias (los grupos).

Para utilizarlo, hay crear un objeto a partir de la clase %Regex.Matcher y pasar la expresión regular y la cadena de entrada. Después, puede llamar a uno de los métodos proporcionados por %Regex.Matcher para

que realice el trabajo.

Ejemplo 1 (grupos simples):

```
set m=##class(%Regex.Matcher).%New("(a|b).*(de)", "abcdeabcde")

w m.Locate() returns 1

w m.Group(1) returns a

w m.Group(2) returns de
```

Ejemplo 2 (grupos anidados y referencias):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(de))(\1)", "abcdeabcde")

w m.Match()          returns    1

w m.GroupCount       returns    4

w m.Group(1)         returns    abcde

w m.Group(2)         returns    a

w m.Group(3)         returns    de

w m.Group(4)         returns    abcde
```

(hay que tener en cuenta el orden de los grupos anidados, porque el paréntesis de apertura marca el inicio de un grupo, los grupos internos tienen un índice numérico mayor que los externos)

Como se mencionó anteriormente, los búferes de captura son una característica muy poderosa, ya que permiten buscar coincidencias en los patrones y extraer el contenido de las coincidencias al mismo tiempo. Sin las expresiones regulares, tendría que buscar las coincidencias como se indica en el paso uno (por ejemplo, utilizando el operador para coincidencias de patrones) y extraer el contenido de las coincidencias (o partes de las mismas) basándose en algunos de los criterios que se indican en el paso dos.

Si necesita agrupar alguna parte de su patrón (por ejemplo, para aplicarle un cuantificador a esa parte), pero no desea rellenar un búfer de captura con el contenido de la parte coincidente, puede definir el grupo como "sin captura" o "pasivo" al colocarle un signo de interrogación seguido de dos puntos adelante del grupo, como en el ejemplo 3, el cual se muestra a continuación.

### Ejemplo 3 (grupo pasivo):

```
set m=##class(%Regex.Matcher).%New("((a|b).*?(?:de))(\1)","abcdeabcde")

w m.Match()          returns    1

w m.Group(1)         returns    abcde

w m.Group(2)         returns    a

w m.Group(3)         returns    abcde

w m.Group(4)         returns    <REGULAR EXPRESSION>zGroupGet+3^%Regex.Matcher.1
```

## 3.5.2. Reemplazar

%Regex.Matcher también proporciona métodos para reemplazar rápidamente el contenido de las coincidencias: ReplaceAll() y ReplaceFirst():

```
set m=##class(%Regex.Matcher).%New(".c.","abcdeabcde")

w m.ReplaceAll("xxxx")        returns    axxxxeaxxxxe

w m.ReplaceFirst("xxxx")      returns    axxxxeabcde
```

También puede referirse a los grupos en sus cadenas de reemplazo. Si añadimos un grupo al patrón del ejemplo anterior, podemos referirnos a su contenido al incluir $1 en la cadena de reemplazo:

```
set m=##class(%Regex.Matcher).%New(".(c).","abcdeabcde")

w m.ReplaceFirst("xx$1xx")   returns    axxcxxeabcde
```

Se puede utilizar $0 para incluir todo el contenido de la coincidencia en la cadena de reemplazo:

```
w m.ReplaceFirst("xx$0xx")   returns    axxbcdxxeabcde
```

## 3.5.3. La propiedad OperationLimit

En la sección 3.2 hablamos sobre los dos métodos para evaluar una expresión regular (DFA y NFA). El motor regex que se utiliza en Caché es un Autómata finito no determinista (NFA, por sus siglas en inglés). Por lo tanto, el tiempo requerido para evaluar varias expresiones regulares en una determinada cadena de entrada puede variar. **[1]**

Se puede utilizar la propiedad *OperationLimit* de un objeto %Regex.Matcher para limitar el número de unidades de ejecución (llamados *clusters*). El tiempo exacto para la ejecución de un cluster depende del entorno. Por lo general, el tiempo requerido para la ejecución de un cluster es de unos cuantos milisegundos. La propiedad *OperationLimit* se establece de manera predeterminada en 0 (sin límites).

# 3.6. Un ejemplo de la vida real: la migración de Perl hacia Caché

Esta sección describe la parte relacionada con regex en un caso de migración de Perl a Caché. En este caso, el script en Perl consiste en docenas de expresiones regulares con mayor o menor complejidad, las cuales se utilizaban tanto para buscar coincidencias como para extraer contenido.

Si no existieran funciones de regex disponibles en Caché, el proyecto de migración implicaría un enorme esfuerzo. Sin embargo, las funciones de regex están disponibles en ObjectScript, y las expresiones regulares de los scripts en Perl pueden utilizarse casi sin realizar modificaciones.

Aquí se muestra una parte del script en Perl:

```
sub readJournal{
    my $vcs_file = shift;

    my $vcs = file($vcs_file)->absolute;
    my $domain = undef;
    my $domain_prefix = undef;
    if ($vcs =~ /[\\\/]([^\\^\/]+)[\\\/]ProjectDB[\\\/](.+)[\\\/]archives[\\\/]/i){
            $domain_prefix = uc $1;
            $domain = uc $2;
        }
```

El único cambio necesario en las expresiones regulares para la migración de Perl hacia Caché fue en el modificador /i- (este provoca que regex no distinga entre las mayúsculas y minúsculas), el cual tuvo que moverse desde el final hacia el principio en las expresiones regulares.

En Perl, el contenido de los búferes de captura se copia en variables especiales ($1 y $2 en el código de Perl que vimos anteriormente). Casi todas las expresiones regulares en el proyecto Perl utilizaban este mecanismo. Para emularlo, se escribió un simple método de encapsulamiento en ObjectScript. Este método utiliza a %Regex.Matcher para evaluar una expresión regular contra una cadena de texto y devuelve el contenido del buffer de captura en forma de una lista ($lb()).

El código correspondiente en ObjectScript es:

```
if ..RegexMatch(

    tVCSFullName,

    "(?i)[\\\/]([^\\^\/]+)[\\\/]ProjectDB[\\\/](.+)[\\\/]archives[\\\/]",

    .tCaptureBufferList)

    {

        set tDomainPrefix=$zcvt($lg(tCaptureBufferList,1), "U")

        set tDomain=$zcvt($lg(tCaptureBufferList,2), "U")

    }

...

Classmethod RegexMatch(pString as %String, pRegex as %String, Output pCaptureBuffer="") {



    #Dim tRetVal as %Boolean=0

    set m=##class(%Regex.Matcher).%New(pRegex,pString)

while m.Locate() {

        set tRetVal=1
```

```
for i=1:1:m.GroupCount {

  set pCaptureBuffer=pCaptureBuffer$lb(m.Group(i))

        }

    }

    quit tRetVal

}
```

# 4. Información de referencia

## 4.7. Información general

Información general y tutoriales:

- **http://www.regular-expressions.info/engine.html**

Tutoriales y ejemplos:

- **http://www.sitepoint.com/demystifying-regex-with-practical-examples/**

Comparaciones entre varios motores regex:

- **https://en.wikipedia.org/wiki/Comparisonofregularexpressionengines**

Hoja de referencia:

- **https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/pdf/**

Libros:

- Jeffrey E. F. Friedl: " Mastering Regular Expressions (Dominando las expresiones regulares) " (consulte **http://regex.info/book.html**)

## 4.8. Documentación online

- Resumen sobre "El uso de las expresiones regulares en Caché": **http://docs.intersystems.com/latest/csp/docbook/ DocBook.UI.Page.cls?KEY=GCOSregexp**
- Documentación sobre $match(): **http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RCOSfmatch**
- Documentación sobre $locate(): **http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RCOSflocate**
- Documentación de la clase %Regex.Matcher: **http://docs.intersystems.com/latest/csp/documatic/%25CSP.Documatic.cls?APP=1&LIBRARY=%25SYS&CLASSNAME=%25Regex.Matcher**

# 4.9. ICU

Como se mencionó anteriormente, InterSystems Caché/IRIS utiliza los motores de ICU. La información completa está disponible en línea:

- **http://userguide.icu-project.org/strings/regexp**
- **http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Metacharacters**
- **http://userguide.icu-project.org/strings/regexp#TOC-Regular-Expression-Operators**
- **http://userguide.icu-project.org/strings/regexp#TOC-Replacement-Text**
- **http://userguide.icu-project.org/strings/regexp#TOC-Flag-Options**

#Mejores prácticas #ObjectScript #Caché #InterSystems IRIS

**URL de fuente:**https://es.community.intersystems.com/post/uso-de-expresiones-regulares-en-objectscript