Chapter 7. Regular Expressions

# Capítulo 7. Expresiones regulares Expresiones regulares

Mientras que lo contrario trae la dicha, Y es un modelo de paz celestial. A quién debemos emparejar con Enrique, siendo rey . . .

-WilliamShakespeare, *The First Part of Henry the Sixth*

Muchas de las características de JavaScript se tomaron prestadas de otros lenguajes. La sintaxis procede de Java, las funciones de Scheme, la herencia prototípica de Self y la función de Expresión Regular de JavaScript de Perl.

A *regular expression* es la especificación de la sintaxis de un lenguaje sencillo. Las expresiones regulares se utilizan con métodos para buscar, sustituir y extraer información de cadenas. Los métodos que funcionan con expresiones regulares son regexp.exec, regexp.test, string.match, string.replace, string.search y string.split. Todos ellos se describirán en el [Capítulo 8](ch08.html). Las expresiones regulares suelen tener una ventaja de rendimiento significativa sobre las operaciones de cadena equivalentes en JavaScript.

Las expresiones regulares proceden del estudio matemático de los lenguajes formales. Ken Thompson adaptó el trabajo teórico de Stephen Kleene sobre los lenguajes de tipo 3 a un comparador de patrones práctico que pudiera incorporarse a herramientas como editores de texto y lenguajes de programación.

La sintaxis de las expresiones regulares en JavaScript se ajusta estrechamente a las formulaciones originales de los Laboratorios Bell, con algunas reinterpretaciones y extensiones adoptadas de Perl. Las reglas para escribir expresiones regulares pueden ser sorprendentemente complejas porque interpretan caracteres en algunas posiciones como operadores, y en posiciones ligeramente diferentes como literales. Peor que ser difíciles de escribir, esto hace que las expresiones regulares sean difíciles de leer y peligrosas de modificar. Es necesario tener un conocimiento bastante completo de toda la complejidad de las expresiones regulares para leerlas correctamente. Para mitigar esto, he simplificado un poco las reglas. Tal como se presentan aquí, las expresiones regulares serán ligeramente menos concisas, pero también serán ligeramente más fáciles de utilizar correctamente. Y eso es bueno, porque las expresiones regulares pueden ser muy difíciles de mantener y depurar.

Las expresiones regulares actuales no son estrictamente regulares, pero pueden ser muy útiles. Las expresiones regulares tienden a ser extremadamente escuetas, incluso crípticas. Son fáciles de usar en su forma más simple, pero pueden volverse rápidamente desconcertantes. Las expresiones regulares de JavaScript son difíciles de leer en parte porque no permiten comentarios ni espacios en blanco. Todas las partes de una expresión regular están muy juntas, lo que las hace casi indescifrables. Esto es especialmente preocupante cuando se utilizan en aplicaciones de seguridad para escanear y validar. Si no puedes leer y comprender una expresión regular, ¿cómo puedes confiar en que funcionará correctamente para todas las entradas? Sin embargo, a pesar de sus evidentes inconvenientes, las expresiones regulares se utilizan ampliamente.

# Un ejemplo

He aquí un ejemplo. Se trata de una expresión regular que empareja URLs. Las páginas de este libro no son infinitamente anchas, así que lo he dividido en dos líneas. En un programa JavaScript, la expresión regular debe estar en una sola línea. Los espacios en blanco son importantes:

var parse\_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)  
(?::(\d+))?(?:\/([^?#]\*))?(?:\?([^#]\*))?(?:#(.\*))?$/;  
  
var url = "http://www.ora.com:80/goodparts?q#fragment";

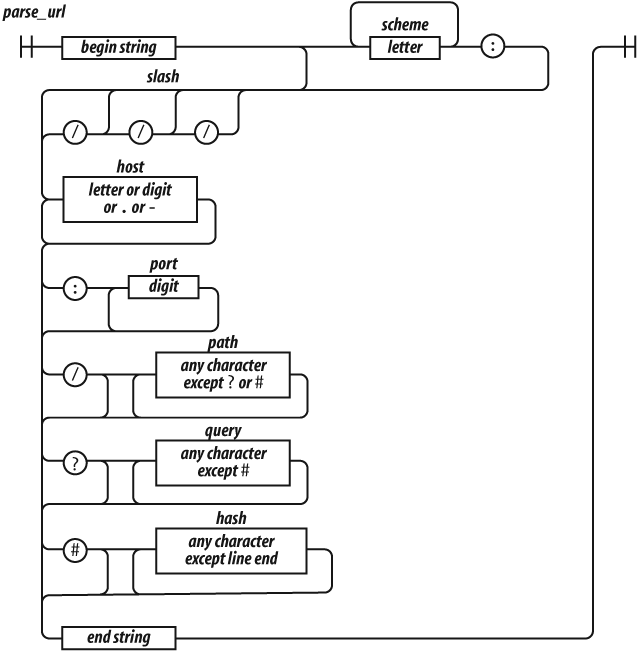
Llamemos al método exec de parse\_url. Si coincide con éxito con la cadena que le pasemos, devolverá una matriz que contiene fragmentos extraídos del url:

var url = "http://www.ora.com:80/goodparts?q#fragment";  
  
var result = parse\_url.exec(url);  
  
var names = ['url', 'scheme', 'slash', 'host', 'port',  
 'path', 'query', 'hash'];  
  
var blanks = ' ';  
var i;  
  
for (i = 0; i < names.length; i += 1) {  
 document.writeln(names[i] + ':' +  
 blanks.substring(names[i].length), result[i]);  
}

Esto produce:

url: http://www.ora.com:80/goodparts?q#fragment  
scheme: http  
slash: //  
host: www.ora.com  
port: 80  
path: goodparts  
query: q  
hash: fragment

En el [Capítulo 2](ch02.html), utilizamos diagramas de ferrocarril para describir el lenguaje JavaScript. También podemos utilizarlos para describir los lenguajes definidos por expresiones regulares. Esto puede facilitar la comprensión de lo que hace una expresión regular. Éste es un diagrama de ferrocarril para parse\_url.



Las expresiones regulares no pueden dividirse en trozos más pequeños como las funciones, por lo que la vía que representa a parse\_url es larga.

Vamos a dividir parse\_url en sus partes para ver cómo funciona.

^

El carácter ^ indica el principio de la cadena. Es un ancla que impide que exec se salte un prefijo que no sea URL.

(?:([A-Za-z]+):)?

Este factor coincide con un nombre de esquema, pero sólo si va seguido de : (dos puntos). El (?:. . . ) indica un grupo no capturador. El sufijo ? indica que el grupo es opcional.

Significa *repeat zero or one time*. El ( . . . ) indica un grupo de captura. Un grupo de captura copia el texto con el que coincide y lo coloca en la matriz result. A cada grupo de captura se le asigna un número. Este primer grupo de captura es el 1, por lo que una copia del texto coincidente con este grupo de captura aparecerá en result[1]. El [ . . . ] indica una clase de caracteres. Esta clase de caracteres, A-Za-z, contiene 26 letras mayúsculas y 26 letras minúsculas. Los guiones indican rangos, de la A a la Z. El sufijo + indica que la clase de caracteres coincidirá una o más veces. El grupo va seguido del carácter :, que se emparejará literalmente.

(\/{0,3})

El siguiente factor es el grupo de captura 2. \/ indica que debe coincidir un carácter / (barra oblicua). Se escapa con \ (barra invertida) para que no se malinterprete como el final del literal de la expresión regular. El sufijo {0,3} indica que el / se emparejará 0 ó 1 ó 2 ó 3 veces.

([0-9.\-A-Za-z]+)

El siguiente factor es el grupo de captura 3. Coincidirá con un nombre de host, que esté formado por uno o más dígitos, letras o . o -. El - se ha escapado as \- para evitar que se confunda con un guión de rango.

(?::(\d+))?

El siguiente factor coincide opcionalmente con un número de puerto, que es una secuencia de uno o más dígitos precedida por un :. \d representa un carácter de dígito. La serie de uno o más dígitos será el grupo de captura 4.

(?:\/([^?#]\*))?

Tenemos otro grupo opcional. Éste comienza con un /. La clase de caracteres [^?#]comienza con un ^, que indica que la clase incluye todos los caracteres excepto ? y #. El \* indica que la clase de caracteres coincide cero o más veces.

Ten en cuenta que aquí estoy siendo descuidado. La clase de todos los caracteres *except ? and #* incluye caracteres de fin de línea, caracteres de control y muchos otros caracteres que realmente no deberían coincidir aquí. La mayoría de las veces esto hará lo que queremos, pero existe el riesgo de que se cuele algún texto incorrecto. Las expresiones regulares descuidadas son una fuente popular de fallos de seguridad. Es mucho más fácil escribir expresiones regulares descuidadas que expresiones regulares rigurosas.

(?:\?([^#]\*))?

A continuación, tenemos un grupo opcional que empieza por ?. Contiene el grupo de captura 6, que contiene cero o más caracteres que no son #.

(?:#(.\*))?

Tenemos un último grupo opcional que empieza por #. El . coincidirá con cualquier carácter excepto con un carácter de final de línea.

$

El $ representa el final de la cadena. Nos asegura que no hay material extra tras el final de la URL.

Ésos son los factores de la expresión regular parse\_url.[[1](#ftn.CHP-7-FN-1)]

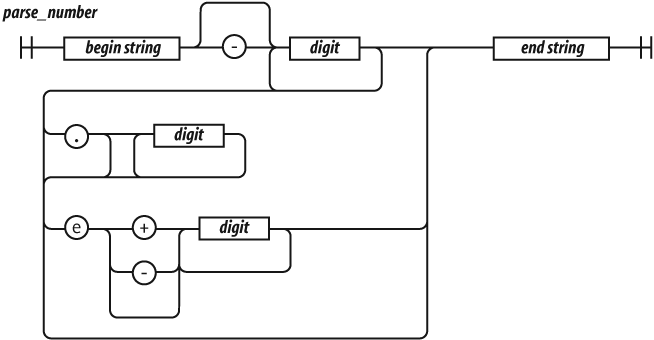
Es posible hacer expresiones regulares más complejas que parse\_url, pero yo no lo recomendaría. Las expresiones regulares son mejores cuando son cortas y sencillas. Sólo así podemos tener la seguridad de que funcionan correctamente y de que podrían modificarse con éxito si fuera necesario.

Existe un grado muy alto de compatibilidad entre los procesadores del lenguaje JavaScript. La parte del lenguaje que es *least* portable es la implementación de expresiones regulares. Las expresiones regulares muy complicadas o enrevesadas tienen más probabilidades de tener problemas de portabilidad. Las expresiones regulares anidadas también pueden sufrir horribles problemas de rendimiento en algunas implementaciones. La simplicidad es la mejor estrategia.

Veamos otro ejemplo: una expresión regular que coincide con números. Los números pueden tener una parte entera con un signo menos opcional, una parte fraccionaria opcional y una parte de exponente opcional.

var parse\_number = /^-?\d+(?:\.\d\*)?(?:e[+\-]?\d+)?$/i;  
  
var test = function (num) {  
 document.writeln(parse\_number.test(num));  
};  
  
test('1'); // true  
test('number'); // false  
test('98.6'); // true  
test('132.21.86.100'); // false  
test('123.45E-67'); // true  
test('123.45D-67'); // false

parse\_number identificó con éxito las cadenas que se ajustaban a nuestra especificación y las que no, pero para las que no, no nos da ninguna información sobre por qué o dónde fallaron la prueba de números.



Vamos a desglosar parse\_number.

/^ $/i

Volvemos a utilizar ^ y $ para anclar la expresión regular. Esto hace que todos los caracteres del texto se comparen con la expresión regular. Si hubiéramos omitido las anclas, la expresión regular nos diría si una cadena contiene un número. Con las anclas, nos dice si la cadena sólo contiene un número. Si sólo incluyéramos ^, coincidiría con las cadenas que empezaran por un número. Si incluimos sólo $, coincidirá con las cadenas que terminen con un número.

La bandera i hace que se ignoren las mayúsculas y minúsculas al buscar letras. La única letra de nuestro patrón es e. Queremos que e coincida también con E. Podríamos haber escrito el factor e como [Ee] o (?:E|e), pero no tuvimos que hacerlo porque utilizamos la bandera i:

-?

El sufijo ? en el signo menos indica que el signo menos es opcional:

\d+

\d significa lo mismo que [0-9]. Coincide con un dígito. El sufijo +hace que coincida con uno o más dígitos:

(?:\.\d\*)?

El (?: . . . )? indica un grupo opcional de no captura. Suele ser mejor utilizar grupos de no captura en lugar de los grupos de captura, menos feos, porque la captura tiene una penalización de rendimiento. El grupo coincidirá con un punto decimal seguido de cero o más dígitos:

(?:e[+\-]?\d+)?

Este es otro grupo opcional de no captura. Coincide con e (o E), un signo opcional y uno o más dígitos.

[Cuando los pulsas todos juntos de nuevo, resulta visualmente bastante confuso: /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]\*))?(?:\?([^#]\*))?(?:#(.\*))?$/