Chapter 8. Methods

# Capítulo 8. Métodos

Aunque esto sea una locura, hay método en ello.

-WilliamShakespeare, *The Tragedy of Hamlet, Prince of Denmark*

JavaScript incluye un pequeño conjunto de métodos estándar que están disponibles en los tipos estándar.

**Matriz**

*array*.concat(*item*...)

El método concat produce un nuevo array que contiene una copia superficial de este *array* con la *item*s. Si *item* es una matriz, cada uno de sus elementos se añade individualmente. Consulta también *array*.push(*item*...) más adelante en este capítulo.

var a = ['a', 'b', 'c'];  
var b = ['x', 'y', 'z'];  
var c = a.concat(b, true);  
// c is ['a', 'b', 'c', 'x', 'y', 'z', true]

*array*.join(*separator*)

El método join hace una cadena a partir de una cadena *array* . Para ello, crea una cadena con cada uno de los elementos de . *array* y luego los concatena con un signo *separator* entre ellos. Por defecto *separator* es ','. Para unir sin separación, utiliza una cadena vacía como el elemento *separator*.

Si estás ensamblando una cadena a partir de un gran número de trozos, suele ser más rápido poner los trozos en una matriz y join que concatenarlos con el operador +:

var a = ['a', 'b', 'c'];  
a.push('d');  
var c = a.join(''); // c is 'abcd';

*array*.pop( )

Los métodos pop y push hacen que un *array* funcione como una pila. El método pop elimina y devuelve el último elemento de esta *array* . Si el elemento *array* es empty, devuelve undefined.

var a = ['a', 'b', 'c'];  
var c = a.pop( ); // a is ['a', 'b'] & c is 'c'

pop puede implementarse así:

Array.method('pop', function ( ) {  
 return this.splice(this.length - 1, 1)[0];  
});

*array*.push(*item*...)

El método push añade *items* al final de una matriz. A diferencia del método concat, éste modifica el elemento *array* y añade los elementos del array enteros. Devuelve la nueva longitud de la matriz *array*:

var a = ['a', 'b', 'c'];  
var b = ['x', 'y', 'z'];  
var c = a.push(b, true);  
// a is ['a', 'b', 'c', ['x', 'y', 'z'], true]  
// c is 5;

push puede implementarse así:

Array.method('push', function ( ) {  
 this.splice.apply(  
 this,  
 [this.length, 0].concat(Array.prototype.slice.apply(arguments)));  
 return this.length;  
});

*array*.inversa( )

El método reverse modifica el *array* invirtiendo el orden de los elementos. Devuelve la función *array*:

var a = ['a', 'b', 'c'];  
var b = a.reverse( );  
// both a and b are ['c', 'b', 'a']

*array*.shift( )

El método shift elimina el primer elemento de un elemento *array* y lo devuelve. Si *array* está vacío, devuelve undefined. shift suele ser mucho más lento que pop:

var a = ['a', 'b', 'c'];  
var c = a.shift( ); // a is ['b', 'c'] & c is 'a'

shift puede implementarse así:

Array.method('shift', function ( ) {  
 return this.splice(0, 1)[0];  
});

*array*.slice(*start*, *end* )

El método slice hace una copia superficial de una parte de un archivo *array* . El primer elemento copiado será *array* [ *start* ]. Se detendrá antes de copiar *array* [ *end* ]. El parámetro *end* es opcional, y por defecto es *array* .length. Si alguno de los parámetros es negativo *array* .length se les sumará para intentar que no sean negativos. Si *start* es mayor o igual que *array* .lengthel resultado será una nueva matriz vacía. No confundas slice con splice. Consulta también *string* .slice más adelante en este capítulo.

var a = ['a', 'b', 'c'];  
var b = a.slice(0, 1); // b is ['a']  
var c = a.slice(1); // c is ['b', 'c']  
var d = a.slice(1, 2); // d is ['b']

*array*.ordenar(*comparefn*)

El método sort ordena el contenido de un archivo *array* en su lugar. Ordena matrices de números incorrectamente:

var n = [4, 8, 15, 16, 23, 42];  
n.sort( );  
// n is [15, 16, 23, 4, 42, 8]

La función de comparación por defecto de JavaScript asume que los elementos a ordenar son cadenas. No es lo suficientemente inteligente como para comprobar el tipo de los elementos antes de compararlos, por lo que convierte los números en cadenas mientras los compara, garantizando un resultado escandalosamente incorrecto.

Afortunadamente, puedes sustituir la función de comparación por la tuya propia. Tu función de comparación debe tomar dos parámetros y devolver 0 si los dos parámetros son iguales, un número negativo si el primer parámetro debe ir primero, y un número positivo si el segundo parámetro debe ir primero. (A los veteranos puede que les recuerde a la sentencia aritmética IFde FORTRAN II).

n.sort(function (a, b) {  
 return a − b;  
});  
// n is [4, 8, 15, 16, 23, 42];

Esa función ordena números, pero no ordena cadenas. Si queremos poder ordenar cualquier matriz de valores simples, debemos esforzarnos más:

var m = ['aa', 'bb', 'a', 4, 8, 15, 16, 23, 42];  
m.sort(function (a, b) {  
 if (a === b) {  
 return 0;  
 }  
 if (typeof a === typeof b) {  
 return a < b ? −1 : 1;  
 }  
 return typeof a < typeof b ? −1 : 1;  
});  
// m is [4, 8, 15, 16, 23, 42, 'a', 'aa', 'bb']

Si las mayúsculas y minúsculas no son significativas, tu función de comparación debe convertir los operandos a minúsculas antes de compararlos. Consulta también *string* .localeCompare más adelante en este capítulo.

Con una función de comparación más inteligente, podemos ordenar una matriz de objetos. Para facilitar las cosas en el caso general, escribiremos una función que haga funciones de comparación:

// Function by takes a member name string and returns  
// a comparison function that can be used to sort an  
// array of objects that contain that member.  
  
var by = function (name) {  
 return function (o, p) {  
 var a, b;  
 if (typeof o === 'object' && typeof p === 'object' && o && p) {  
 a = o[name];  
 b = p[name];  
 if (a === b) {  
 return 0;  
 }  
 if (typeof a === typeof b) {  
 return a < b ? −1 : 1;  
 }  
 return typeof a < typeof b ? −1 : 1;  
 } else {  
 throw {  
 name: 'Error',  
 message: 'Expected an object when sorting by ' + name;  
 };  
 }  
 };  
};  
  
var s = [  
 {first: 'Joe', last: 'Besser'},  
 {first: 'Moe', last: 'Howard'},  
 {first: 'Joe', last: 'DeRita'},  
 {first: 'Shemp', last: 'Howard'},  
 {first: 'Larry', last: 'Fine'},  
 {first: 'Curly', last: 'Howard'}  
];  
s.sort(by('first')); // s is [  
// {first: 'Curly', last: 'Howard'},  
// {first: 'Joe', last: 'DeRita'},  
// {first: 'Joe', last: 'Besser'},  
// {first: 'Larry', last: 'Fine'},  
// {first: 'Moe', last: 'Howard'},  
// {first: 'Shemp', last: 'Howard'}  
// ]

El método sort no es estable, por lo que:

s.sort(by('first')).sort(by('last'));

no garantiza que produzca la secuencia correcta. Si quieres ordenar por varias claves, de nuevo tendrás que hacer más trabajo. Podemos modificar by para que tome un segundo parámetro, otro método compareque será llamado para romper los empates cuando la clave mayor produzca una coincidencia:

// Function by takes a member name string and an  
// optional minor comparison function and returns  
// a comparison function that can be used to sort an  
// array of objects that contain that member. The  
// minor comparison function is used to break ties  
// when the o[name] and p[name] are equal.  
  
var by = function (name, minor) {  
 return function (o, p) {  
 var a, b;  
 if (o && p && typeof o === 'object' && typeof p === 'object') {  
 a = o[name];  
 b = p[name];  
 if (a === b) {  
 return typeof minor === 'function' ? minor(o, p) : 0;  
 }  
 if (typeof a === typeof b) {  
 return a < b ? −1 : 1;  
 }  
 return typeof a < typeof b ? −1 : 1;  
 } else {  
 throw {  
 name: 'Error',  
 message: 'Expected an object when sorting by ' + name;  
 };  
 }  
 };  
};  
  
s.sort(by('last', by('first'))); // s is [  
// {first: 'Joe', last: 'Besser'},  
// {first: 'Joe', last: 'DeRita'},  
// {first: 'Larry', last: 'Fine'},  
// {first: 'Curly', last: 'Howard'},  
// {first: 'Moe', last: 'Howard'},  
// {first: 'Shemp', last: 'Howard'}  
// ]

*array*.splice(*start*, *deleteCount*, *item*...)

El método splice elimina elementos de un *array*sustituyéndolos por nuevos *item* s. El parámetro *start* es el número de una posición dentro de *array* . El parámetro *deleteCount* es el número de elementos que hay que borrar a partir de esa posición. Si hay parámetros adicionales, esos *item* s se insertarán en la posición. Devuelve un array que contiene los elementos borrados.

El uso más popular de splice es borrar elementos de una matriz. No confundas splice con slice:

var a = ['a', 'b', 'c'];  
var r = a.splice(1, 1, 'ache', 'bug');  
// a is ['a', 'ache', 'bug', 'c']  
// r is ['b']

splice puede implementarse así:

Array.method('splice', function (start, deleteCount) {  
 var max = Math.max,  
 min = Math.min,  
 delta,  
 element,  
 insertCount = max(arguments.length - 2, 0),  
 k = 0,  
 len = this.length,  
 new\_len,  
 result = [],  
 shift\_count;  
  
 start = start || 0;  
 if (start < 0) {  
 start += len;  
 }  
 start = max(min(start, len), 0);  
 deleteCount = max(min(typeof deleteCount === 'number' ?  
 deleteCount : len, len − start), 0);  
 delta = insertCount − deleteCount;  
 new\_len = len + delta;  
 while (k < deleteCount) {  
 element = this[start + k];  
 if (element !== undefined) {  
 result[k] = element;  
 }  
 k += 1;  
 }  
 shift\_count = len - start - deleteCount;  
 if (delta < 0) {  
 k = start + insertCount;  
 while (shift\_count) {  
 this[k] = this[k − delta];  
 k += 1;  
 shift\_count −= 1;  
 }  
 this.length = new\_len;  
 } else if (delta > 0) {  
 k = 1;  
 while (shift\_count) {  
 this[new\_len − k] = this[len − k];  
 k += 1;  
 shift\_count −= 1;  
 }  
 this.length = new\_len;  
 }  
 for (k = 0; k < insertCount; k += 1) {  
 this[start + k] = arguments[k + 2];  
 }  
 return result;  
});

*array*.unshift(*item*...)

El método unshift es como el método push, salvo que coloca la *item* s al principio de este *array* en lugar de al final. Devuelve el *array* nuevo length:

var a = ['a', 'b', 'c'];  
var r = a.unshift('?', '@');  
// a is ['?', '@', 'a', 'b', 'c']  
// r is 5

unshift puede implementarse así:

Array.method('unshift', function ( ) {  
 this.splice.apply(this,  
 [0, 0].concat(Array.prototype.slice.apply(arguments)));  
 return this.length;  
});

**Función**

*function*.aplicar(*thisArg*, *argArray* )

El método apply invoca a *function*pasando el objeto que se vinculará a thisy una matriz opcional de argumentos. El método apply se utiliza en el patrón de invocación apply[(Capítulo 4](ch04.html)):

Function.method('bind', function (that) {  
  
// Return a function that will call this function as  
// though it is a method of that object.  
  
 var method = this,  
 slice = Array.prototype.slice,  
 args = slice.apply(arguments, [1]);  
 return function ( ) {  
 return method.apply(that,  
 args.concat(slice.apply(arguments, [0])));  
 };  
});  
  
var x = function ( ) {  
 return this.value;  
}.bind({value: 666});  
alert(x( )); // 666

**Número**

*number*.toExponencial(*fractionDigits*)

El método toExponential convierte este *number* en una cadena con forma exponencial. El parámetro opcional *fractionDigits* controla el número de decimales. Debe estar entre 0 y 20:

document.writeln(Math.PI.toExponential(0));  
document.writeln(Math.PI.toExponential(2));  
document.writeln(Math.PI.toExponential(7));  
document.writeln(Math.PI.toExponential(16));  
document.writeln(Math.PI.toExponential( ));  
  
// Produces  
  
3e+0  
3.14e+0  
3.1415927e+0  
3.1415926535897930e+0  
3.141592653589793e+0

*number*.toFixed(*fractionDigits*)

El método toFixed convierte esta *number* en una cadena con forma decimal. El parámetro opcional *fractionDigits* controla el número de decimales. Debe estar entre 0 y 20. El valor por defecto es 0:

document.writeln(Math.PI.toFixed(0));  
document.writeln(Math.PI.toFixed(2));  
document.writeln(Math.PI.toFixed(7));  
document.writeln(Math.PI.toFixed(16));  
document.writeln(Math.PI.toFixed( ));  
  
// Produces  
  
3  
3.14  
3.1415927  
3.1415926535897930  
3

*number*.toPrecision(*precision*)

El método toPrecision convierte esto *number* a una cadena en forma decimal. El parámetro opcional *precision* controla el número de dígitos de precisión. Debe estar entre 1 y 21:

document.writeln(Math.PI.toPrecision(2));  
document.writeln(Math.PI.toPrecision(7));  
document.writeln(Math.PI.toPrecision(16));  
document.writeln(Math.PI.toPrecision( ));  
  
// Produces  
  
3.1  
3.141593  
3.141592653589793  
3.141592653589793

*number*.toString(*radix*)

El método toString convierte esto *number* en una cadena. El parámetro opcional *radix* controla el radix, o base. Debe estar entre 2 y 36. Por defecto *radix* es la base 10. El parámetro *radix* se suele utilizar con números enteros, pero se puede utilizar con cualquier número.

El caso más común, *number* .toString( )puede escribirse más sencillamente como String(*number* ):

document.writeln(Math.PI.toString(2));  
document.writeln(Math.PI.toString(8));  
document.writeln(Math.PI.toString(16));  
document.writeln(Math.PI.toString( ));  
  
// Produces  
  
11.001001000011111101101010100010001000010110100011  
3.1103755242102643  
3.243f6a8885a3  
3.141592653589793

**Objeto**

*object*.hasPropiedad(*name*)

El método hasOwnProperty devuelve true si el objeto *object* contiene una propiedad *name* . No se examina la cadena de prototipos. Este método es inútil si la propiedad *name* es hasOwnProperty:

var a = {member: true};  
var b = Object.create(a); // from Chapter 3  
var t = a.hasOwnProperty('member'); // t is true  
var u = b.hasOwnProperty('member'); // u is false  
var v = b.member; // v is true

**RegExp**

*regexp*.exec(*string*)

El método exec es el más potente (y lento) de los métodos que utilizan expresiones regulares. Si coincide con éxito con el *regexp* y el *string*devuelve una matriz. El elemento 0 de la matriz contendrá la subcadena que coincida con el elemento *regexp* . El elemento 1 es el texto capturado por el grupo 1, el elemento 2 es el texto capturado por el grupo 2, y así sucesivamente. Si la coincidencia falla, devuelve null.

Si el elemento *regexp* tiene una bandera g, las cosas son un poco más complicadas. La búsqueda no empieza en la posición 0 de la cadena, sino en la posición *regexp* .lastIndex (que inicialmente es cero). Si la coincidencia es correcta, entonces *regexp* .lastIndex se pondrá en la posición del primer carácter después de la coincidencia. Una coincidencia fallida vuelve a *regexp* .lastIndex a 0.

Esto te permite buscar varias apariciones de un patrón en una cadena llamando a execen un bucle. Hay que tener cuidado con un par de cosas. Si sales del bucle antes de tiempo, debes poner a 0 *regexp* .lastIndex a 0 antes de volver a entrar en el bucle. Además, el factor ^ sólo coincide cuando *regexp* .lastIndex es 0:

// Break a simple html text into tags and texts.  
// (See string.replace for the entityify method.)  
  
// For each tag or text, produce an array containing  
// [0] The full matched tag or text  
// [1] The /, if there is one  
// [2] The tag name  
// [3] The attributes, if any  
  
var text = '<html><body bgcolor=linen><p>' +  
 'This is <b>bold<\/b>!<\/p><\/body><\/html>';  
var tags = /[^<>]+|<(\/?)([A-Za-z]+)([^<>]\*)>/g;  
var a, i;  
  
while ((a = tags.exec(text))) {  
 for (i = 0; i < a.length; i += 1) {  
 document.writeln(('// [' + i + '] ' + a[i]).entityify( ));  
 }  
 document.writeln( );  
}  
  
// Result:  
  
// [0] <html>  
// [1]  
// [2] html  
// [3]  
  
// [0] <body bgcolor=linen>  
// [1]  
// [2] body  
// [3] bgcolor=linen  
  
// [0] <p>  
// [1]  
// [2] p  
// [3]  
  
// [0] This is  
// [1] undefined  
// [2] undefined  
// [3] undefined  
  
// [0] <b>  
// [1]  
// [2] b  
// [3]  
  
// [0] bold  
// [1] undefined  
// [2] undefined  
// [3] undefined  
  
// [0] </b>  
// [1] /  
// [2] b  
// [3]  
  
// [0] !  
// [1] undefined  
// [2] undefined  
// [3] undefined  
  
// [0] </p>  
// [1] /  
// [2] p  
// [3]  
  
// [0] </body>  
// [1] /  
// [2] body  
// [3]  
  
// [0] </html>  
// [1] /  
// [2] html  
// [3]

*regexp*.test(*string*)

El método test es el más sencillo (y rápido) de los métodos que utilizan expresiones regulares. Si el factor *regexp* coincide con *string*devuelve true; en caso contrario, devuelve false. No utilices la bandera g con este método:

var b = /&.+;/.test('frank &amp; beans');  
// b is true

test podría implementarse como

RegExp.method('test', function (string) {  
 return this.exec(string) !== null;  
});

**Cadena**

*string*.charAt(*pos*)

El método charAt devuelve el carácter en la posición *pos* de este *string*. Si *pos* es menor que cero o mayor o igual que *string*.length , devuelve la cadena vacía. JavaScript no tiene un tipo de carácter.

resultado de este método es una cadena:

var name = 'Curly';  
var initial = name.charAt(0); // initial is 'C'

charAt podría implementarse como:

String.method('charAt', function (pos) {  
return this.slice(pos, pos + 1);  
});

*string*.charCodeAt(*pos*)

El método charCodeAt es igual que charAt excepto que, en lugar de devolver una cadena, devuelve una representación entera del valor del punto de código del carácter en la posición *pos* de ese *string*. Si *pos* es menor que cero o mayor o igual que *string*.length

NaN:

var name = 'Curly';  
var initial = name.charCodeAt(0); // initial is 67

*string*.concat(*string*...

El método concat crea una nueva cadena concatenando otras cadenas. Se utiliza poco porque el operador + es más cómodo:

var s = 'C'.concat('a', 't'); // s is 'Cat'

*string* .  
indexOf(*searchString*  
 *position* )

indexOf busca una *searchString* dentro de una *string*. Si lo encuentra, devuelve la posición del primer carácter coincidente; en caso contrario, devuelve -1. El parámetro opcional *position*hace que la búsqueda comience en la posición especificada en *string*

var text = 'Mississippi';  
var p = text.indexOf('ss'); // p is 2  
p = text.indexOf('ss', 3); // p is 5  
p = text.indexOf('ss', 6); // p is −1

*string*.lastIndexOf(*searchString*  
 *position* )

El método lastIndexOf es como el método indexOf, salvo que busca desde el final de la cadena en lugar de desde el principio:

var text = 'Mississippi';  
var p = text.lastIndexOf('ss'); // p is 5  
p = text.lastIndexOf('ss', 3); // p is 2  
p = text.lastIndexOf('ss', 6); // p is 5

*string*.localeCompare(*that*)

El método localeCompare compara dos cadenas. No se especifican las reglas para comparar las cadenas. Si la cadena *string* es menor que *that*, el resultado es negativo. Si son iguales, el resultado es cero. Esto es similar a la convención para la función de comparación *array*.sort :

var m = ['AAA', 'A', 'aa', 'a', 'Aa', 'aaa'];  
m.sort(function (a, b) {  
 return a.localeCompare(b);  
});  
// m (in some locale) is  
// ['a', 'A', 'aa', 'Aa', 'aaa', 'AAA']

*string*.match(*regexp*)

El método match compara una cadena y una expresión regular. La forma de hacerlo depende de la bandera g. Si no existe la bandera g, el resultado de llamar a *string*.match( *regexp* ) es el mismo que llamar a *regexp*.exec( *string* ) .

embargo, si *regexp* tiene la bandera g, entonces produce una matriz de todas las coincidencias pero excluye los grupos de captura:

var text = '<html><body bgcolor=linen><p>' +  
 'This is <b>bold<\/b>!<\/p><\/body><\/html>';  
var tags = /[^<>]+|<(\/?)([A-Za-z]+)([^<>]\*)>/g;  
var a, i;  
  
a = text.match(tags);  
for (i = 0; i < a.length; i += 1) {  
 document.writeln(('// [' + i + '] ' + a[i]).entityify( ));  
}  
  
// The result is  
  
// [0] <html>  
// [1] <body bgcolor=linen>  
// [2] <p>  
// [3] This is  
// [4] <b>  
// [5] bold  
// [6] </b>  
// [7] !  
// [8] </p>  
// [9] </body>  
// [10] </html>

*string*.replace(*searchValue*  
 *replaceValue* )

El método replace realiza una operación de búsqueda y reemplazo en este *string*, produciendo una nueva cadena. El argumento *searchValue* puede ser una cadena o un objeto de expresión regular.

Si es una cadena

, *only* reemplazará la primera

ocurrencia

de *searchValue*, por lo que:

var result = "mother\_in\_law".replace('\_', '-');

producirá "mother-in\_law"

,

que puede ser una decepción.

Si *searchValue* es una expresión regular y si tiene la bandera g, entonces reemplazará todas las ocurrencias.

Si no tiene el indicador g, sólo sustituirá la primera aparición

.

*replaceValue* puede ser una cadena o una función.

Si *replaceValue* es una cadena, el carácter $ tiene un significado especial:

// Capture 3 digits within parens  
  
var oldareacode = /\((\d{3})\)/g;  
var p = '(555)666-1212'.replace(oldareacode, '$1-');  
// p is '555-666-1212'

| Sustitución de |
| --- |

la

| secuencia del dólar |
| --- |

|  |  |
| --- | --- |
| $& | El texto coincidente |

|  |
| --- |
| Texto del grupo de captura |

El texto que precede a la

*string*.search(*regexp*)

El método search es como el método indexOf, salvo que toma un objeto de expresión regular en lugar de una cadena. Devuelve la posición del primer carácter de la primera coincidencia, si la hay, o -1 si la búsqueda falla. La bandera g se ignora. No hay parámetro *position*:

var text = 'and in it he says "Any damn fool could';  
var pos = text.search(/["']/); // pos is 18

*string*.slice(*start*  
 *end* )

El método slice crea una nueva cadena copiando una parte de otra *string*. Si el parámetro *start* es negativo, le añade *string*.length . El parámetro *end* es opcional, y su valor por defecto es *string*.length . Si el parámetro *end* es negativo, se le añade *string*.length . El parámetro *end* es uno mayor que la posición del último carácter. Para obtener n caracteres a partir de la posición p, utiliza *string*.slice(p, p + n) . Consulta también *string*.substring y *array*.slice , más adelante y anteriormente en este capítulo, respectivamente.

var text = 'and in it he says "Any damn fool could';  
var a = text.slice(18);  
// a is '"Any damn fool could'  
var b = text.slice(0, 3);  
// b is 'and'  
var c = text.slice(−5);  
// c is 'could'  
var d = text.slice(19, 32);  
// d is 'Any damn fool'

*string*.split(*separator*  
 *limit* )

El método split crea una matriz de cadenas dividiendo este *string* en trozos. El parámetro opcional *limit* puede limitar el número de trozos que se dividirán.

El parámetro

*separator* puede ser una cadena o una expresión regular.

Si *separator* es la cadena vacía, se produce una matriz de caracteres simples

:

var digits = '0123456789';  
var a = digits.split('', 5);  
// a is ['0', '1', '2', '3', '4']

De lo contrario, se buscan en *string* todas las apariciones de *separator*. Cada unidad de texto entre los separadores se copia en la matriz.

La bandera g se ignora:

var ip = '192.168.1.0';  
var b = ip.split('.');  
// b is ['192', '168', '1', '0']  
  
var c = '|a|b|c|'.split('|');  
// c is ['', 'a', 'b', 'c', '']  
  
var text = 'last, first ,middle';  
var d = text.split(/\s\*,\s\*/);  
// d is [  
// 'last',  
// 'first',  
// 'middle'  
// ]

Hay que tener cuidado con algunos casos especiales.

El

texto de los grupos de captura se incluirá en la división

:

var e = text.split(/\s\*(,)\s\*/);  
// e is [  
// 'last',  
// ',',  
// 'first',  
// ',',  
// 'middle'  
// ]

Algunas implementaciones suprimen las cadenas vacías en la matriz de salida cuando *separator* es una expresión regular

:

var f = '|a|b|c|'.split(/\|/);  
// f is ['a', 'b', 'c'] on some systems, and  
// f is ['', 'a', 'b', 'c', ''] on others

*string*.substring(*start*, *end* </span> )

El método substring es igual que el método slice, salvo que no gestiona el ajuste para parámetros negativos. No hay razón para utilizar el método substring. Utiliza en su lugar slice.

*string*.toLocaleMinúsculas( )

El método toLocaleLowerCaseproduce una nueva cadena que se hace convirtiendo esta *string* a minúsculas utilizando las reglas de la configuración regional. Esto es principalmente en beneficio del turco, porque en ese idioma `I' se convierte a 1, no a `i'.

*string*.toLocaleMayúsculas( )

El método toLocaleUpperCaseproduce una nueva cadena que se hace convirtiendo esta *string* a mayúsculas utilizando las reglas de la configuración regional. Esto es principalmente en beneficio del turco, porque en ese idioma `i' se convierte a `y no a "I".

*string*.aMayúsculas( )

El método toLowerCase produce una nueva cadena que se hace convirtiendo esta *string* a minúsculas.

*string*.aMayúsculas( )

El método toUpperCase produce una nueva cadena convirtiendo esto *string* a mayúsculas.

String.fromCharCode(*char*...)

La función String.fromCharCodeproduce una cadena a partir de una serie de números.

var a = String.fromCharCode(67, 97, 116);  
// a is 'Cat'