Contenido

Parámetros Rest y operador Spread

Muchas funciones nativas de JavaScript soportan un número arbitrario de argumentos.

Por ejemplo:

- `Math.max(arg1, arg2, ..., argN)` – devuelve el argumento más grande.
- `Object.assign(dest, src1, ..., srcN)` – copia las propiedades de `src1..N` en `dest`.
- ...y otros más

En este capítulo aprenderemos como hacer lo mismo. Y, además, cómo trabajar cómodamente con dichas funciones y arrays.

Parámetros Rest . . .

Una función puede ser llamada con cualquier número de argumentos sin importar cómo sea definida.

Por ejemplo::

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

No habrá ningún error por "exceso" de argumentos. Pero, por supuesto, en el resultado solo los dos primeros serán tomados en cuenta, entonces el resultado del código anterior es `3`.

El resto de los parámetros pueden ser referenciados en la definición de una función con 3 puntos `...` seguidos por el nombre del array que los contendrá. Literalmente significan "Reunir los parámetros restantes en un array".

Por ejemplo, para reunir todos los parámetros en un array `args`:

```
function sumAll(...args) { // args es el nombre del array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Podemos elegir obtener los primeros parámetros como variables, y juntar solo el resto.

Aquí los primeros dos argumentos van a variables y el resto va al array `titles`:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julio Cesar

  // el resto va en el array titles
  // por ejemplo titles = ["Cónsul", "Emperador"]
  alert( titles[0] ); // Cónsul
  alert( titles[1] ); // Emperador
  alert( titles.length ); // 2
}

showName("Julio", "Cesar", "Cónsul", "Emperador");
```

Los parámetros rest deben ir al final

Los parámetros rest recogen todos los argumentos sobrantes, por lo que el siguiente código no tiene sentido y causa un error:

```
function f(arg1, ...rest, arg2) { // arg2 después de
...rest ?!
  // error
}
```

`...rest` debe ir siempre último.

La variable "arguments"

También existe un objeto símil-array especial llamado **`arguments`** que contiene todos los argumentos indexados.

Por ejemplo:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // arguments es iterable
  // for(let arg of arguments) alert(arg);
}

// muestra: 2, Julio, Cesar
showName("Julio", "Cesar");

// muestra: 1, Ilya, undefined (no hay segundo
argumento)
showName("Ilya");
```

Antiguamente, los parámetros rest no existían en el lenguaje, y usar `arguments` era la única manera de obtener todos los argumentos de una función. Y aún funciona, podemos encontrarlo en código antiguo.

Pero la desventaja es que a pesar de que `arguments` es símil-array e iterable, no es un array. No soporta los métodos de array, no podemos ejecutar `arguments.map(...)` por ejemplo.

Además, siempre contiene todos los argumentos. No podemos capturarlos parcialmente como hicimos con los parámetros rest.

Por lo tanto, cuando necesitemos estas funcionalidades, los parámetros rest son preferidos.

Las funciones flecha no poseen `"arguments"`
Si accedemos el objeto `arguments` desde una función flecha, toma su valor dela función "normal" externa.

Aquí hay un ejemplo:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

Como recordamos, las funciones de flecha no tienen su propio `this`. Ahora sabemos que tampoco tienen el objeto especial `arguments`.

Sintaxis Spread

Acabamos de ver cómo obtener un array de la lista de parámetros.

Pero a veces necesitamos hacer exactamente lo opuesto.

Por ejemplo, existe una función nativa Math.max que devuelve el número más grande de una lista:

```
alert( Math.max(3, 5, 1) ); // 5
```

Ahora bien, supongamos que tenemos un array `[3, 5, 1]`. ¿Cómo ejecutamos `Math.max` con él?

Pasando la variable no funcionará, porque `Math.max` espera una lista de argumentos numéricos, no un único array:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

Y seguramente no podremos listar manualmente los ítems en el código `Math.max(arr[0], arr[1], arr[2])`, porque tal vez no

sepamos cuántos son. A medida que nuestro script se ejecuta, podría haber muchos elementos, o podría no haber ninguno. Y eso podría ponerse feo.

¡*Operador Spread* al rescate! Es similar a los parámetros rest, también usa `...`, pero hace exactamente lo opuesto.

Cuando `...arr` es usado en la llamada a una función, "expande" el objeto iterable `arr` en una lista de argumentos.

Para `Math.max`:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread convierte el
array en una lista de argumentos)
```

También podemos pasar múltiples iterables de esta manera:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

Incluso podemos combinar el operador spread con valores normales:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Además, el operador spread puede ser usado para combinar arrays:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, luego arr,
después 2, después arr2)
```

En los ejemplos de arriba utilizamos un array para demostrar el operador spread, pero cualquier iterable funcionará también.

Por ejemplo, aquí usamos el operador spread para convertir la cadena en un array de caracteres:

```
let str = "Hola";

alert( [...str] ); // H,o,l,a
```

El operador spread utiliza internamente iteradores para iterar los elementos, de la misma manera que `for..of` hace.

Entonces, para una cadena `for..of` retorna caracteres y `...str` se convierte en `"H","o","l","a"`. La lista de caracteres es pasada a la inicialización del array `[...str]`.

Para esta tarea en particular también podríamos haber usado `Array.from`, ya que convierte un iterable (como una cadena de caracteres) en un array:

```
let str = "Hola";

// Array.from convierte un iterable en un array
alert( Array.from(str) ); // H,o,l,a
```

El resultado es el mismo que `[...str]`.

Pero hay una sutil diferencia entre `Array.from(obj)` y `[...obj]`:

- `Array.from` opera con símil-arrays e iterables.
- El operador spread solo opera con iterables.

Por lo tanto, para la tarea de convertir algo en un array, `Array.from` tiende a ser más universal.

Copia de un objeto array

Es posible hacer una compia de un array con la sintaxis de spread

```
let arr = [1, 2, 3];

let arrCopy = [...arr]; // separa el array en una lista de parameters
                        // luego pone el resultado en un nuevo array
```

```
// ¿los arrays tienen el mismo contenido?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy));
// true

// ¿los arrays son iguales?
alert(arr === arrCopy); // false (no es la misma
referencia)

// modificando nuestro array inicial no modifica la
copia:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Nota que es posible hacer lo mismo para hacer una copia de un objeto:

```
let obj = { a: 1, b: 2, c: 3 };

let objCopy = { ...obj }; // separa el objeto en una
lista de parámetros
                          // luego devuelve el
resultado en un nuevo objeto

// ¿tienen los objetos el mismo contenido?
alert(JSON.stringify(obj) === JSON.stringify(objCopy));
// true

// ¿son iguales los objetos?
alert(obj === objCopy); // false (no es la misma
referencia)

// modificando el objeto inicial no modifica la copia:
obj.d = 4;
alert(JSON.stringify(obj)); //
{"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

Esta manera de copiar un objeto es mucho más corta que `let objCopy = Object.assign({}, obj);` o para un array `let arrCopy = Object.assign([], arr);` por lo que preferimos usarla siempre que podemos.

Resumen

Cuando veamos `"..."` en el código, son los parámetros rest o el operador spread.

Hay una manera fácil de distinguir entre ellos:

- Cuando `...` se encuentra al final de los parámetros de una función, son los "parámetros rest" y recogen el resto de la lista de argumentos en un array.
- Cuando `...` está en la llamada de una función o similar, se llama "operador spread" y expande un array en una lista.

Patrones de uso:

- Los parámetros rest son usados para crear funciones que acepten cualquier número de argumentos.
- El operador spread es usado para pasar un array a funciones que normalmente requieren una lista de muchos argumentos.

Todos los argumentos de una llamado a una función están también disponibles en el "viejo" **`arguments`: un objeto símil-array** iterable.