
Mapas y conjuntos

Si bien los objetos y los arreglos son la forma más común de almacenar datos en JavaScript, tienen algunas limitaciones. Por ejemplo, ni los objetos ni los arreglos tienen la capacidad de crear una lista única sin duplicados. Del mismo modo, los objetos no pueden contener llaves que no son cadenas o símbolos. Para superar estos desafíos y más, existen dos tipos especiales adicionales de objetos conocidos como **mapas** y **conjuntos**. Ambos tienen una funcionalidad única, lo que los hace muy adecuados para ciertas tareas. En este documento, nos sumergiremos en ambos objetos y cuándo usarlos.

Conjuntos

Los conjuntos son listas mutables de valores, que son bastante similares a los arreglos. La diferencia entre conjuntos y arreglos es que los conjuntos no pueden contener duplicados. Esto los convierte en un rendimiento para las tareas donde necesitas tener una lista única. Si intentas agregar un elemento duplicado a un conjunto, se rechazará automáticamente sin lanzar un error.

Crear un conjunto es relativamente simple. Todo lo que necesitas es el constructor, `Set()` y luego métodos especiales como agregar y eliminar para modificarlo:

```
let mySet = new Set()
mySet.add(4)
mySet.add(4)
console.log(mySet) // Set(1) {4}
```

Nota: Si intentas agregar valores duplicados a un conjunto, solo se agregará uno. Los conjuntos rechazan automáticamente los valores duplicados.

Los conjuntos solo admitirán la adición de valores únicos para primitivas, pero con los objetos, usan la referencia de un objeto en lugar de su valor. Eso significa que es posible agregar dos objetos que tienen el mismo valor pero que tienen diferentes referencias. Esto se muestra en el siguiente ejemplo:

```
let mySet = new Set()
mySet.add({ "name" : "John" })
mySet.add({ "name" : "John" })
// Set(2) {{ "name" : "John" }, { "name" : "John" }}
```

```
console.log(mySet)
```

Los conjuntos también manejan la igualdad de una manera ligeramente diferente de la que hemos visto hasta ahora. Los conjuntos considerarán que NaN es igual a NaN a pesar de que en ejemplos anteriores deberíamos que `Nan === NaN` y `NaN == NaN` devuelve falso.

La razón por la cual los conjuntos funcionan de esta manera es porque usan un algoritmo de igualdad ligeramente diferente. Estos algoritmos de igualdad se resumen a continuación:

- El algoritmo de igualdad de `IsLooselyEqual` se aplica cuando usamos doble iguales (`==`) y coincide con el valor.
- El algoritmo de igualdad de `IsStrictlyEqual` se aplica cuando usamos triple iguales (`===`) y coincide con valor/tipo.
- El `SameValueZero` es utilizado por conjuntos. Considera tanto el valor como el tipo de igualdad como `IsStrictlyEqual`, pero también considera que NaN es igual a NaN.

Como tal, si intentamos agregar dos NaN a un conjunto, el resultado es que solo se agrega uno:

```
let mySet = new Set()
mySet.add(NaN)
mySet.add(NaN)
// Set(1) {NaN}
console.log(mySet)
```

Nota: Los conjuntos son realmente útiles cuando tienes un arreglo donde no están permitidos los duplicados. Siempre es posible que el código defectuoso deje que los duplicados se encuentren en un arreglo que no debería tener ninguno, pero los conjuntos aseguran que esto nunca suceda.

Modificación de conjuntos

Los conjuntos son mutables como otros objetos, y los mutamos a través de métodos de conjunto especiales. Los tres métodos principales para cambiar un conjunto son:

- `Set.add()` – Para agregar un elemento a tu conjunto
- `Set.delete()` – Para eliminar un elemento de tu conjunto
- `Set.clear()` – Para borrar todos los elementos de tu conjunto

En el siguiente ejemplo, utilizamos los tres métodos para crear un conjunto que contenga solo el valor 5 antes de limpiarlo para vaciar:

```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
mySet.delete(4)
console.log(mySet) // Set(1) {5}
mySet.clear()
console.log(mySet) // Set(0) {}
```

Comprobación de membresía del conjunto

Dado que el caso de uso principal de los conjuntos es crear listas únicas, se convierten en una herramienta poderosa para verificar si existe un cierto valor en un conjunto específico. Los conjuntos tienen un método incorporado llamado `Set.has()` para hacer esto:

```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
mySet.delete(4)
let checkSet = mySet.has(4)
console.log(checkSet) // false
```

Comprobar el tamaño del conjunto

Mientras que en los arreglos usamos `Array.length` para obtener el tamaño de un arreglo, los conjuntos usan una propiedad llamada `size` para lograr la misma funcionalidad:

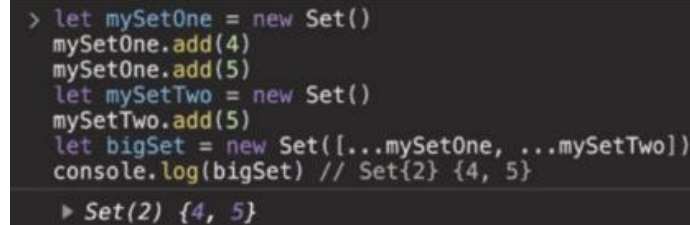
```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
mySet.add(6)
console.log(mySet.size) // Console logs 3
```

Mezcla de conjuntos

Dado que los conjuntos son iterables e implementan el protocolo iterator, se pueden fusionar (mezclar) utilizando el operador de tres puntos (...), como se muestra en el siguiente ejemplo. Los duplicados se eliminarán automáticamente si existen. Puedes ver cómo se ve esto en la figura 1.

```
let mySetOne = new Set()
mySetOne.add(4)
mySetOne.add(5)
```

```
let mySetTwo = new Set()
mySetTwo.add(5)
let bigSet = new Set([...mySetOne, ...mySetTwo])
console.log(bigSet) // Set{2} {4, 5}
```



```
> let mySetOne = new Set()
  mySetOne.add(4)
  mySetOne.add(5)
  let mySetTwo = new Set()
  mySetTwo.add(5)
  let bigSet = new Set([...mySetOne, ...mySetTwo])
  console.log(bigSet) // Set{2} {4, 5}

► Set(2) {4, 5}
```

Figura 1: Los conjuntos se pueden fusionar (mezclar) utilizando la sintaxis de tres puntos ya que son iterables. Al hacerlo, el nuevo conjunto tendrá todos los duplicados eliminados de él.

Iteración de conjunto y valores

Dado que los conjuntos son iterables, lo que significa que pueden ser alimentados con ciclo `for...de` como lo harías para los arreglos. Sin embargo, usar ciclos `for...en` no funcionará ya que los conjuntos no tienen índices. Dado que los conjuntos no tienen índices, también significa que no puedes acceder a un elemento establecido con notación de paréntesis cuadrado, como `mySet[2]`:

```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
mySet.add(6)
for(let x of mySet) {
  console.log(x) // 4, 5, 6
}
for(let x in mySet) {
  console.log(x) // undefined
}
```

Al igual que los arreglos, los conjuntos también tienen un método `forEach` integrado para una fácil iteración. Como hemos discutido anteriormente, ya que los ciclos son generalmente más rápidos que usar el método `forEach`, pero tienen algunas ventajas sobre un ciclo para la creación de un nuevo contexto, ya que usan funciones de devolución de llamada (callback):

```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
```

```
mySet.add(6)
mySet.forEach((x) => {
  console.log(x) // 4, 5, 6
})
```

Finalmente, si las limitaciones en los métodos de conjuntos o su falta de índices se vuelven demasiado, puedes convertir fácilmente un conjunto en un arreglo en su lugar utilizando `Array.from()`. Después de eso, podrás usar todos los métodos de arreglo en tu conjunto:

```
let mySet = new Set()
mySet.add(4)
mySet.add(5)
mySet.add(6)
let arrayFromSet = Array.from(mySet) // [ 4, 5, 6 ]
```

Llaves de conjuntos y valores

Al igual que los objetos, los conjuntos heredan los métodos `keys()`, `values()` y `entries()`. En conjuntos, tanto `keys()` como los `values()` hacen lo mismo y simplemente devolverán una lista de todos los miembros en el conjunto como un objeto especial conocido como `setIterator`:

```
let mySet = new Set()
mySet.add(5)
mySet.add(10)
let getKeys = mySet.keys() // SetIterator{5, 10}
```

Los `SetIterators` son diferentes de los conjuntos, y solo tienen un método, que es el método `next()`. El método `next()` te permite iterar a través de los conjuntos de un elemento a la vez. Cada vez que lo haces, se devuelve un objeto que consiste en el valor del elemento establecido y si la iteración está completa.

Puedes ver cómo se ve eso en el siguiente ejemplo:

```
let mySet = new Set()
mySet.add(5)
mySet.add(10)
let getKeys = mySet.keys() // SetIterator{5, 10}
console.log(getKeys.next()) // { value: 5, done: false }
console.log(getKeys.next()) // { value: 10, done: false }
console.log(getKeys.next()) // { value: undefined, done: false }
```

Mientras que `keys()` y `values()` proporcionan una forma abreviada de generar `SetIterators`, también se pueden crear refiriéndose a la propiedad del iterador `sets` para lograr el mismo resultado:

```
let mySet = new Set()
mySet.add(5)
mySet.add(10)
let getKeys = mySet[Symbol.iterator]() // SetIterator{5, 10}
console.log(getKeys.next()) // { value: 5, done: false }
console.log(getKeys.next()) // { value: 10, done: false }
console.log(getKeys.next()) // { value: undefined, done: false }
```

Nota: También puedes iterar a través de `SetIterators` usando un ciclo `for...de`, ya que son iterables.

El método `entries()` también está disponible para todos los conjuntos. En los objetos, el método de entradas devuelve arreglos en la forma `[llave, valor]`. Dado que los conjuntos no tienen llaves, el valor se devuelve como la llave y el valor.

La razón principal por la cual este método está disponible en conjuntos es garantizar la consistencia con los mapas, que tienen llaves y valores. `Set.entries()` todavía devuelve un `setIterator`, al igual que `keys()` y `values()`:

```
let mySet = new Set()
mySet.add(5)
mySet.add(10)
let setEntries = mySet.entries()
for(let x of setEntries) {
  console.log(x)
  // Returns [ 5, 5 ], [ 10, 10 ]
}
```

Mapas

Si bien los conjuntos son estructuras similares al arreglo que también proporcionan la eliminación duplicada automática, los mapas son estructuras similares a objetos con algunas mejoras adicionales. Si bien son similares a los objetos, también difieren de algunas maneras muy importantes:

- Los mapas no tienen un prototipo: Mientras que los objetos de mapa heredan un prototipo, el mapa en sí no contiene llaves a menos que insertemos una en él. En los

objetos JavaScript, tenemos que usar el `Object.hasOwnProperty()` para averiguar si una propiedad es del objeto en sí o de su prototipo. Los mapas no buscan en el prototipo, lo que significa que sabemos que los mapas solo tendrán lo que les ponemos explícitamente en ellos y no heredarán otras propiedades.

- Los mapas garantizan orden por cronología: Se garantiza que se ordenan por el pedido que fueron insertados. Dado que los mapas usan un método `set()` específico para crear nuevas entradas, el orden de las entradas corresponde cronológicamente a cuando se usó `set()`.
- Los mapas permiten que cualquier cosa se confiera como llave: Las llaves de un mapa pueden ser cualquier cosa, incluida una función o incluso un objeto. En los objetos, debe ser una cadena o símbolo.
- Los mapas tienen beneficios de rendimiento: Tienen un mejor rendimiento que los objetos en tareas que requieren eliminación/adición rápida o frecuente de datos.
- Los mapas son iterables de forma predeterminada, a diferencia de los objetos.

Similar a los conjuntos, se inicia un mapa utilizando el constructor `new Map()`:

```
let myMap = new Map()
```

Y al igual que los conjuntos, los mapas vienen con métodos `add()`, `delete()` y `clear()`. La diferencia clave aquí es que, para agregar elementos de mapa, debes especificar una llave y un valor y eliminar un elemento de mapa, debes especificar el nombre de llave que deseas eliminar. Puedes ver cómo funcionan estos métodos en el siguiente ejemplo y en la figura 2.

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
myMap.delete("key")
console.log(myMap) // Map(1) {'secondKey' => 'value'}
myMap.clear()
console.log(myMap) // Map(0) {}
```



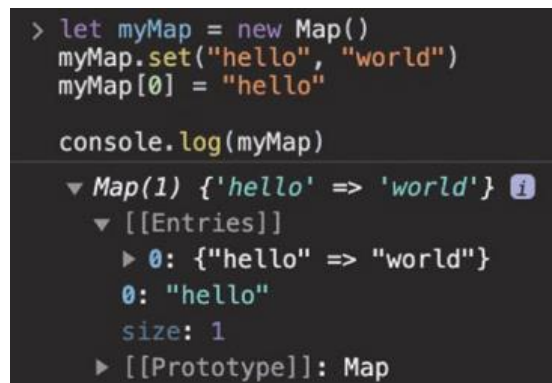
Figura 2: Agregar una entrada al mapa con notación de paréntesis cuadrado la agrega al objeto, al mismo nivel que el prototipo. No agrega una entrada al mapa en sí, lo que significa que pierdes todos los beneficios de los mapas.

Dado que no es posible establecer una llave antes que otra con `set()`, se supone que el orden de los mapas es de manera confiable al ser cronológica.

Algo confusamente, los mapas parecen ser capaces de tener elementos establecidos utilizando la notación de paréntesis cuadrado, pero esta no es la forma correcta de establecer nuevas entradas del mapa. De hecho, establecer una propiedad en un mapa utilizando paréntesis cuadrados da como resultado que la propiedad se define en el objeto del mapa, y no en el mapa en sí:

```
let myMap = new Map()
myMap.set("hello", "world")
myMap[0] = "hello"
console.log(myMap)
```

La separación entre las entradas de mapa establecidas con la notación de paréntesis cuadrado y el objeto que contiene el mapa es por qué los mapas no heredan las propiedades de un prototipo. Puedes ver esta separación en la figura 3, donde existe una sección `[[Entries]]` para contener entradas de mapa reales, mientras que el prototipo y otras propiedades se establecen en el objeto de mapa.



```
> let myMap = new Map()
myMap.set("hello", "world")
myMap[0] = "hello"

console.log(myMap)

▼ Map(1) {'hello' => 'world'} ⓘ
  ▼ [[Entries]]
    ► 0: {"hello" => "world"}
    0: "hello"
    size: 1
    ► [[Prototype]]: Map
```

Figura 3: Los mapas no heredan un prototipo ya que sus propiedades están contenidas en una propiedad especial llamada `[[Entries]]`. Intentar establecer una propiedad en el mapa en sí con los paréntesis cuadrados no funcionará ya que la propiedad se establecerá en el objeto map.

Mientras que los objetos solo permiten cadenas, números o símbolos para que se establezcan como llaves, los mapas permiten que las llaves sean de cualquier tipo. Como tal, una llave de función es posible en los mapas, mientras que es imposible en los objetos:

```
let myMap = new Map()
myMap.set(() => { return true }, "value")
console.log(myMap) // Map(1) {{f => 'value'}}
```


Recuperación de propiedades del mapa

Para recuperar elementos de un mapa, usamos un método llamado `get()`:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
console.log(myMap.get("key"))
```

Dado que los mapas no heredan prototipos, no nos encontramos con conflictos de llave. Por ejemplo, los objetos típicamente heredarán llaves como `valueOf()` de su prototipo. Eso significa que podemos usar este método en cualquier objeto, como se muestra en el siguiente ejemplo:

```
let myObject = { "key" : "value" }
myObject.valueOf() // Returns value of myObject
```

Con los mapas, estos métodos no son heredados y, por lo tanto, no existirán. Esto evita cualquier posibilidad de conflicto de llave.

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
console.log(myMap.get("myValue")) // undefined
```

Dado que los mapas pueden tener llaves de cualquier tipo, recuperar las llaves no primitivas puede volverse un poco complicado. Para hacer eso, debemos mencionar la referencia original a la llave, lo que significa que necesitamos almacenar la llave en una variable para recuperarla de manera confiable. Puedes ver cómo funciona en el siguiente ejemplo:

```
let myMap = new Map()
let someArray = [ 1, 2 ]
myMap.set(someArray, "value")
```

Para recuperar la entrada de `someArray`, hacemos referencia a la variable, `someArray`:

```
myMap.get(someArray) // value
```

La funcionalidad que nos permite establecer las llaves del mapa para cualquier valor conduce a algunos problemas interesantes. Dado que el mapa tiene una referencia al objeto, `someArray`, `someArray` nunca será recolectada de la basura. Normalmente, cuando los objetos no se realizan, JavaScript los eliminará automáticamente de la memoria.

Esto significa que, si haces lo que hicimos en el ejemplo anterior, podrías encontrar problemas de memoria y, en última instancia, errores de memoria en tu código.

Para superar ese problema en particular, existe otro tipo de mapa llamado mapa débil (WeakMap). Los mapas débiles tienen un conjunto más limitado de métodos (get, set, has y delete), pero permitirán la recolección de basura de objetos donde la única referencia a ese objeto está en el mapa débil. Los mapas débiles no son iterables, lo que significa que acceder a ellos se basa únicamente en get(). Esto los hace mucho más limitados que los mapas, pero útiles al resolver este problema en particular.

Puedes crear un mapa débil de la misma manera que un mapa, pero utilizando el constructor de mapas débiles en su lugar:

```
let myMap = new WeakMap()
let someArray = [ 1, 2 ]
myMap.set(someArray, "value")
```

Verificar la existencia de la llave en un mapa

Podemos verificar la membresía de una llave en un mapa utilizando el método HAS ():

```
let myMap = new Map()
myMap.set('firstKey', 'someValue')
myMap.has('firstKey') // true
```

Decir cuán grande es un mapa de JavaScript

Al igual que en los conjuntos, los mapas usan la propiedad de tamaño en lugar de la longitud para obtener el número de llaves que existen dentro de ellas:

```
let myMap = new Map()
myMap.set('firstKey', 'someValue')
myMap.size // 1
```

Esto es en realidad una mejora en los objetos. Dado que los objetos no tienen una longitud por defecto, generalmente usamos una mezcla de Object.keys() para convertirlos en arreglos para encontrar su tamaño:

```
let myObj = { "name" : "John" };
let sizeOfObj = Object.keys(myObj).length; // 1
Using Maps with non-string keys
```

Iterando, mezclando y accediendo a mapas

Otra característica útil de los mapas es el hecho de que son iterables de forma predeterminada. En primer lugar, eso significa que se pueden mezclar utilizando el operador de tres puntos (...), como se muestra en el siguiente ejemplo:

```
let myMapOne = new Map()
myMapOne.set("key", "value")
let myMapTwo = new Map()
myMapTwo.set("secondKey", "value")
// Map{2} {4, 5}
let bigMap = new Map([...myMapOne, ...myMapTwo])
console.log(bigMap) // Map(2) {'key' => 'value', 'secondKey' => 'value'}
```

Esto es muy diferente de los objetos, donde normalmente necesitamos usar `Object.keys()`, `Object.entries()` u `Object.values()` para iterar a través de ellos

```
let myObject = { "item-1": false, "item-2": true, "item-3": false }
let getKeys = Object.keys(myObject)
for(let item of getKeys) {
  console.log(myObject[item])
}
```

Con los mapas, las cosas son más simples. Podemos iterar directamente en el mapa en sí.

Cuando hacemos esto con un ciclo `for...de`, esto devuelve un elemento de arreglo de valores de llave útil. También tenemos la opción de usar el método `forEach` incorporado:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
for(let item of myMap) {
  console.log(item) // [ 'key', 'value' ], [ 'secondKey', 'value' ]
  console.log(item[1]) // 'value', 'value'
}
myMap.forEach((x) => {
  console.log(x) // 'value', 'value'
})
```

Llaves y valores en mapas

Los mapas también tienen métodos `keys()`, `values()` y `entries()`. Todos estos métodos devuelven un tipo especial de objeto conocido como `MapIterator`. Esto es esencialmente lo mismo que cuando miramos los conjuntos, que devolvieron un `SetIterator` para estos métodos.

Dado que las llaves de objetos solo pueden ser símbolos, números o cadenas, generalmente solo usamos la llave para acceder a los valores de los objetos. Sin embargo, en los mapas, las llaves pueden ser de cualquier tipo, por lo que el método `keys()` adquiere un papel más útil para darnos el valor llave, que en realidad puede ser algo útil como una función.

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
let mapKeys = myMap.keys()
console.log(mapKeys.next()) // {value: 'key', done: false}
console.log(mapKeys.next()) // {value: 'secondKey', done: false}
console.log(mapKeys.next()) // {value: undefined, done: true}
```

Nota: Los `MapIterators` siguen siendo iterables, por lo que también puedes usar un ciclo `for...de` en ellos.

La misma funcionalidad que hemos mostrado arriba para las llaves también se puede lograr para los valores, usando `Map.values()`:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
let mapValues = myMap.values()
console.log(mapValues.next()) // {value: "value", done: false}
console.log(mapValues.next()) // {value: "value", done: false}
console.log(mapValues.next()) // {value: undefined, done: true}
```

Finalmente, podemos recuperar la llave y el valor en este formato usando `Map.entries()`. Devuelve cada par de valores de llave como `[llave, valor]`:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
let mapValues = myMap.entries()
console.log(mapValues.next()) // {value: [ "key", "value" ], done: false}
```

```
console.log(mapValues.next()) // { value: [ "secondKey", "value"], done: false }
console.log(mapValues.next()) // { value: undefined, done: true }
```

Serialización de mapas en JavaScript

Cuando queremos serializar un objeto JavaScript, usamos `JSON.parse()` y `JSON.stringify()`. Dado que los mapas son objetos vacíos con entradas dentro de ellos, tratar de tritararlos (`stringify`) resulta en una cadena de objeto vacía:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
console.log(JSON.stringify(myMap)) // "{}"
```

La única forma de serializar un mapa para la transferencia a través de una API es convertirlo en un objeto o arreglo. Para hacer eso, podemos usar `Array.from()` para convertir nuestro mapa en un arreglo y luego usar `JSON.stringify()` para serializarlo:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
myMap.set("thirdKey", "value")

let objConversion = Array.from(myMap)
// Returns [ ["some", "value"], ["someOther", "value"], ["aFinal", "value"] ]
console.log(JSON.stringify(objConversion))
```

Para volver a cambiarlo a un mapa, podemos usar `JSON.parse()` junto con `new Map()`. En este caso, un arreglo de entradas pasadas al constructor del mapa dará como resultado un objeto de mapa completamente formado:

```
let myMap = new Map()
myMap.set("key", "value")
myMap.set("secondKey", "value")
myMap.set("thirdKey", "value")
let objConversion = JSON.stringify(Array.from(myMap))
// Returns [ ["some", "value"], ["someOther", "value"], ["aFinal", "value"] ]
console.log(objConversion)
let toObj = JSON.parse(objConversion)
let toMap = new Map(toObj)
// Map(3) { 'key' => 'value', 'secondKey' => 'value', 'thirdKey' => 'value' }
console.log(toMap)
```

Resumen

En este documento, hemos cubierto conjuntos y mapas. Hemos revisado la utilidad de ambos y cuando los usarías sobre objetos regulares. Los conjuntos permiten listas únicas, mientras que los mapas tienen mucha utilidad adicional que los objetos simplemente no lo hacen.

Hasta este punto, hemos dependido completamente de objetos y arreglos para manejar nuestros datos. Ahora puedes ser más específico y crear conjuntos y mapas cuando también surja la necesidad. Dado que estos objetos están optimizados para las tareas que hemos descrito en este documento, también pueden proporcionar una optimización adicional a tu código.