JavaScript Map Object

Summary: in this tutorial, you will learn about the JavaScript Map object that maps a key to a value.

Introduction to JavaScript Map object

Before ES6, we often used an object to emulate a map by mapping a key to a value of any type. But using an object as a map has some side effects:

1. An object always has a default key like the prototype.
2. A key of an object must be a string or a symbol, you cannot use an object as a key.
3. An object does not have a property that represents the size of the map.

ES6 provides a new collection type called `Map` that addresses these deficiencies.

By definition, a `Map` object holds key-value pairs. Keys are unique in a Map's collection. In other words, a key in a Map object only appears once.

Keys and values of a Map can be any values.

When iterating a `Map` object, each iteration returns a 2-member array of `[key, value]`. The iteration order follows the insertion order which corresponds to the order in which each key-value pair was first inserted into the Map by the `set()` method.

To create a new `Map`, you use the following syntax:

```
let map = new Map([iterable]);
```

The `Map()` accepts an optional iterable object whose elements are key-value pairs.

Useful JavaScript Map methods

- `clear()` – removes all elements from the map object.

- `delete(key)` – removes an element specified by the key. It returns if the element is in the map, or false if it does not.
- `entries()` – returns a new Iterator object that contains an array of `[key, value]` for each element in the map object. The order of objects in the map is the same as the insertion order.
- `forEach(callback[, thisArg])` – invokes a callback for each key-value pair in the map in the insertion order. The optional thisArg parameter sets the `this` value for each callback.
- get(key) – returns the value associated with the key. If the key does not exist, it returns undefined.
- has(key) – returns true if a value associated with the key exists or false otherwise.
- `keys()` – returns a new Iterator that contains the keys for elements in insertion order.
- `set(key, value)` – sets the value for the key in the map object. It returns the map object itself therefore you can chain this method with other methods.
- `values()` returns a new iterator object that contains values for each element in insertion order.

JavaScript Map examples

Let's take some examples of using a Map object.

Create a new Map object

Suppose you have a list of `user` objects as follows:

```
let john = {name: 'John Doe'},
    lily = {name: 'Lily Bush'},
    peter = {name: 'Peter Drucker'};
```

Assuming that you have to create a map of users and roles. In this case, you use the following code:

```
let userRoles = new Map();
```

The `userRoles` is an instance of the `Map` object and its type is an object as illustrated in the following example:

```
console.log(typeof(userRoles)); // object
console.log(userRoles instanceof Map); // true
```

Add elements to a Map

To assign a role to a user, you use the `set()` method:

```
userRoles.set(john, 'admin');
```

The `set()` method maps user `john` with the `admin` role. Since the `set()` method is chainable, you can save some typing as shown in this example:

```
userRoles.set(lily, 'editor')
          .set(peter, 'subscriber');
```

Initialize a map with an iterable object

As mentioned earlier, you can pass an iterable object to the `Map()` constructor:

```
let userRoles = new Map([
    [john, 'admin'],
    [lily, 'editor'],
    [peter, 'subscriber']
]);
```

Get an element from a map by key

If you want to see the roles of `John` , you use the `get()` method:

```
userRoles.get(john); // admin
```

If you pass a key that does not exist, the `get()` method will return `undefined`.

```
let foo = {name: 'Foo'};
userRoles.get(foo); //undefined
```

Check the existence of an element by key

To check if a key exists in the map, you use the `has()` method.

```
userRoles.has(foo); // false
userRoles.has(lily); // true
```

Get the number of elements in the map

The `size` property returns the number of entries of the Map object.

```
console.log(userRoles.size); // 3
```

Iterate over map keys

To get the keys of a `Map` object, you use the `keys()` method. The `keys()` returns a new iterator object that contains the keys of elements in the map.

The following example displays the username of the users in the `userRoles` map object.

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (const user of userRoles.keys()) {
  console.log(user.name);
}
```

Output:

```
John Doe
Lily Bush
Peter Drucker
```

Iterate over map values

Similarly, you can use the `values()` method to get an iterator object that contains values for all the elements in the map:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (let role of userRoles.values()) {
  console.log(role);
}
```

Output:

```
admin
editor
subscriber
```

Iterate over map elements

Also, the `entries()` method returns an iterator object that contains an array of `[key,value]` of each element in the `Map` object:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (const role of userRoles.entries()) {
  console.log(`${role[0].name}: ${role[1]}`);
}
```

To make the iteration more natural, you can use destructuring as follows:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (let [user, role] of userRoles.entries()) {
  console.log(`${user.name}: ${role}`);
}
```

In addition to `for...of` loop, you can use the `forEach()` method of the map object:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

userRoles.forEach((role, user) => console.log(`${user.name}: ${role}`));
```

Convert map keys or values to an array

Sometimes, you want to work with an array instead of an iterable object, in this case, you can use the spread operator.

The following example converts keys for each element into an array of keys:

```
var keys = [...userRoles.keys()];
console.log(keys);
```

Output:

```
[ { name: 'John Doe' },
  { name: 'Lily Bush' },
  { name: 'Peter Drucker' } ]
```

The following converts the values of elements to an array:

```
let roles = [...userRoles.values()];
console.log(roles);
```

Output

```
[ 'admin', 'editor', 'subscriber' ]
```

Delete an element by key

To delete an entry in the map, you use the `delete()` method.

```
userRoles.delete(john);
```

Delete all elements in the map

To delete all entries in the `Map` object, you use the `clear()` method:

```
userRoles.clear();
```

Hence, the size of the map now is zero.

```
console.log(userRoles.size); // 0
```

WeakMap

A `WeakMap` is similar to a `Map` except for the keys of a `WeakMap` must be objects. It means that when a reference to a key (an object) is out of scope, the corresponding value is automatically released from the memory.

A `WeakMap` only has subset methods of a `Map` object:

- `get(key)`
- `set(key, value)`
- `has(key)`
- `delete(key)`

Here are the main differences between a `Map` and a `WeekMap` :

- Elements of a WeakMap cannot be iterated.
- Cannot clear all elements at once.
- Cannot check the size of a WeakMap.

In this tutorial, you have learned how to work with the JavaScript Map object and its useful methods to manipulate entries in the map.