4th February 2020

WeakMap and WeakSet

As we know from the chapter Garbage collection, JavaScript engine stores a value in memory while it is reachable (and can potentially be used).

For instance:

```
let john = { name: "John" };

// the object can be accessed, john is the reference to it

// overwrite the reference
john = null;

// the object will be removed from memory
```

Usually, properties of an object or elements of an array or another data structure are considered reachable and kept in memory while that data structure is in memory.

For instance, if we put an object into an array, then while the array is alive, the object will be alive as well, even if there are no other references to it.

Like this:

```
let john = { name: "John" };

let array = [ john ];

john = null; // overwrite the reference

// john is stored inside the array, so it won't be garbage-collected
// we can get it as array[0]
```

Similar to that, if we use an object as the key in a regular `Map`, then while the `Map` exists, that object exists as well. It occupies memory and may not be garbage collected.

For instance:

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // overwrite the reference
```

```
7
8 // john is stored inside the map,
9 // we can get it by using map.keys()
```

`WeakMap` is fundamentally different in this aspect. It doesn't prevent garbage-collection of key objects.

Let's see what it means on examples.

WeakMap

The first difference from `Map` is that `WeakMap` keys must be objects, not primitive values:

```
1 let weakMap = new WeakMap();
2
3 let obj = {};
4
5 weakMap.set(obj, "ok"); // works fine (object key)
6
7 // can't use a string as the key
8 weakMap.set("test", "Whoops"); // Error, because "test" is not an object
```

Now, if we use an object as the key in it, and there are no other references to that object – it will be removed from memory (and from the map) automatically.

```
1 let john = { name: "John" };
2
3 let weakMap = new WeakMap();
4 weakMap.set(john, "...");
5
6 john = null; // overwrite the reference
7
8 // john is removed from memory!
```

Compare it with the regular `Map` example above. Now if `john` only exists as the key of `WeakMap` – it will be automatically deleted from the map (and memory).

`WeakMap` does not support iteration and methods `keys()`, `values()`, `entries()`, so there's no way to get all keys or values from it.

`WeakMap` has only the following methods:

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

Why such a limitation? That's for technical reasons. If an object has lost all other references (like `john` in the code above), then it is to be garbage-collected automatically. But technically it's not exactly specified *when the cleanup happens*.

The JavaScript engine decides that. It may choose to perform the memory cleanup immediately or to wait and do the cleaning later when more deletions happen. So, technically the current element count of a `WeakMap` is

not known. The engine may have cleaned it up or not, or did it partially. For that reason, methods that access all keys/values are not supported.

Now where do we need such data structure?

Use case: additional data

The main area of application for `WeakMap` is an *additional data storage*.

If we're working with an object that "belongs" to another code, maybe even a third-party library, and would like to store some data associated with it, that should only exist while the object is alive – then `WeakMap` is exactly what's needed.

We put the data to a `WeakMap`, using the object as the key, and when the object is garbage collected, that data will automatically disappear as well.

```
weakMap.set(john, "secret documents");
// if john dies, secret documents will be destroyed automatically
```

Let's look at an example.

For instance, we have code that keeps a visit count for users. The information is stored in a map: a user object is the key and the visit count is the value. When a user leaves (its object gets garbage collected), we don't want to store their visit count anymore.

Here's an example of a counting function with `Map`:

```
// 📁 visitsCount.js
let visitsCountMap = new Map(); // map: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

And here's another part of the code, maybe another file using it:

```
// 📁 main.js
let john = { name: "John" };

countUser(john); // count his visits

// later john leaves us
john = null;
```

Now `john` object should be garbage collected, but remains in memory, as it's a key in `visitsCountMap`.

We need to clean `visitsCountMap` when we remove users, otherwise it will grow in memory indefinitely. Such cleaning can become a tedious task in complex architectures.

We can avoid it by switching to `WeakMap` instead:

```
// 📁 visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => visits count

// increase the visits count
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

Now we don't have to clean `visitsCountMap`. After `john` object becomes unreachable by all means except as a key of `WeakMap`, it gets removed from memory, along with the information by that key from `WeakMap`.

Use case: caching

Another common example is caching: when a function result should be remembered ("cached"), so that future calls on the same object reuse it.

We can use `Map` to store results, like this:

```
// 📁 cache.js
let cache = new Map();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculations of the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// Now we use process() in another file:

// 📁 main.js
let obj = {/* let's say we have an object */};

let result1 = process(obj); // calculated

// ...later, from another place of the code...
let result2 = process(obj); // remembered result taken from cache

// ...later, when the object is not needed any more:
obj = null;

alert(cache.size); // 1 (Ouch! The object is still in cache, taking memory!)
```

For multiple calls of `process(obj)` with the same object, it only calculates the result the first time, and then just takes it from `cache`. The downside is that we need to clean `cache` when the object is not needed any more.

If we replace `Map` with `WeakMap`, then this problem disappears: the cached result will be removed from memory automatically after the object gets garbage collected.

```
// 📁 cache.js
let cache = new WeakMap();

// calculate and remember the result
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculate the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// 📁 main.js
let obj = {/* some object */};

let result1 = process(obj);
let result2 = process(obj);

// ...later, when the object is not needed any more:
obj = null;

// Can't get cache.size, as it's a WeakMap,
// but it's 0 or soon be 0
// When obj gets garbage collected, cached data will be removed as well
```

WeakSet

`WeakSet` behaves similarly:

- It is analogous to `Set`, but we may only add objects to `WeakSet` (not primitives).
- An object exists in the set while it is reachable from somewhere else.
- Like `Set`, it supports `add`, `has` and `delete`, but not `size`, `keys()` and no iterations.

Being "weak", it also serves as an additional storage. But not for an arbitrary data, but rather for "yes/no" facts. A membership in `WeakSet` may mean something about the object.

For instance, we can add users to `WeakSet` to keep track of those who visited our site:

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John visited us
visitedSet.add(pete); // Then Pete
visitedSet.add(john); // John again

// visitedSet has 2 users now

// check if John visited?
alert(visitedSet.has(john)); // true

// check if Mary visited?
```

```
alert(visitedSet.has(mary)); // false

john = null;

// visitedSet will be cleaned automatically
```

The most notable limitation of `WeakMap` and `WeakSet` is the absence of iterations, and inability to get all current content. That may appear inconvenient, but does not prevent `WeakMap/WeakSet` from doing their main job – be an “additional” storage of data for objects which are stored/managed at another place.

Summary

`WeakMap` is `Map` -like collection that allows only objects as keys and removes them together with associated value once they become inaccessible by other means.

`WeakSet` is `Set` -like collection that stores only objects and removes them once they become inaccessible by other means.

Both of them do not support methods and properties that refer to all keys or their count. Only individual operations are allowed.

`WeakMap` and `WeakSet` are used as “secondary” data structures in addition to the “main” object storage. Once the object is removed from the main storage, if it is only found as the key of `WeakMap` or in a `WeakSet`, it will be cleaned up automatically.

Tasks

Store "unread" flags

importance: 5

There’s an array of messages:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

Your code can access it, but the messages are managed by someone else’s code. New messages are added, old ones are removed regularly by that code, and you don’t know the exact moments when it happens.

Now, which data structure could you use to store information about whether the message “has been read”? The structure must be well-suited to give the answer “was it read?” for the given message object.

P.S. When a message is removed from `messages`, it should disappear from your structure as well.

P.P.S. We shouldn’t modify message objects, add our properties to them. As they are managed by someone else’s code, that may lead to bad consequences.

solution

Store read dates

importance: 5

There's an array of messages as in the previous task. The situation is similar.

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

The question now is: which data structure you'd suggest to store the information: "when the message was read?".

In the previous task we only needed to store the "yes/no" fact. Now we need to store the date, and it should only remain in memory until the message is garbage collected.

P.S. Dates can be stored as objects of built-in `Date` class, that we'll cover later.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)