An adventure in sparse arrays

I offered to coach JavaScript recently, and an exercise I dreamt up was to implement every array (prototype) method, and write tests that they work.

Since I'd come down with tonsillitis which apparently comes with a full on bedridden fever, I thought I'd have a go myself. And I learnt a few things along the way. Today, I wanted to share the swiss cheese wonder of sparse arrays!

(see what I did there? 🧀)

There is a TLDR if you really want to skip all my lovely words!

A closer look

A sparse array is an an array with "holes". Holes tend to look like `undefined` (when logging), but they're not really not defined. It's just that JavaScript doesn't have a value for a hole, which…one might argue is undefined, but alas, that value for undefined value is already in use!

More recently Chrome started using `empty` for holes…which makes sense, but spoils my fun…

A few fun ways to view sparse arrays:

```
new Array(1) // hole × 1
[ , ] // hole × 1
[ 1, ] // int(1), no holes, length: 1
[ 1, , ] // int(1) and hole × 1
```

Obviously that last example is super precious, you're left to spot the difference between the comma that causes the sparse array, and the comma that's treated as a trailing comma and is ignored entirely by JavaScript.

Then there's the fun confusion of trying to look at a hole…which I'm now starting to liken to a black hole in my mind (to be interpreted both ways.).

```
const a = [ undefined, , ];
console.log(a[0]) // undefined
console.log(a[1]) // undefined
```

These values are both undefined, but not because they're both undefined. WAT? `a[0]` points a set array element containing the value undefined. Whereas `a[1]` is not set to any value (it's a hole remember), but any value not defined is undefined. So, they kinda look the same.

There still be hope! A one line `prop in object` can tell us if we're looking at a hole:

```
const a = [ undefined, , ];
console.log(0 in a) // true
console.log(1 in a) // false
```

So we're all safe now right? we've escaped the clutches of the unknown undefined but not undefined hole. Nope, as The O.G. Daddy of JavaScript pointed out on Twitter, there's potential risk if there's any 3rd party scripts or libraries involved (which unless your code is 100% siloed, is likely at some level). A 3rd party could tamper with the array prototype and:

```
// elsewhere in boobooland
Array.prototype[1] = 'fool!'
// and in my code
const a = [ undefined, , ];
console.log(0 in a) // true
console.log(1 in a) // true … 😱
```

As with most things in JavaScript, the proper way of checking whether there's a hole or not requires using a method I barely remembered even existed:

```
// elsewhere in boobooland
Array.prototype[1] = 'fool!'
// and in my code
const a = [ undefined, , ];
console.log(a.hasOwnProperty(1)) // false
// 👊 have some of that boobooland
```

Since `hasOwnProperty` is from the days of ES3, using that method allows you to officially call yourself a Retro Scripter.

But why?

Performance is a really good reason these holes exist. Say you create a `new Array(10000000)` (10 million). There's not actually 10 million allocated values in that array, and the browser isn't storing anything.

Tripping over maps

The cherry on the top, is that I always trip up on sparse arrays combined with map, forEach and filter (though I'm sure it has some funsies effects on other callback-based methods too).

The callback passed to `forEach` will skip over holes. Makes sense. It seems right that a fully sparse array doesn't have any for-eaches:

```
const a = new Array(100);
let i = 0
a.forEach(() => i++);
console.log(i) // 0 - the callback is never cal
```

It is worth noting that the array is still walked, and the larger the array, the longer the `forEach` will take, even if it's sparse. Oddly, I ran some (contrived) tests that showed it took ~85ms to run `forEach` on a 10 million sparse array, but over 20 seconds for 100 million. Weird.

Then there's `map`. The `map` function, like `forEach` doesn't call on the hole, but it will always return the hole in your result.

```
const a = [ undefined, , true ];
const res = a.map(v => typeof v);
console.log(res) // [ "undefined", hole × 1, "b
```

This `map` with sparse arrays also explains why my goto method for generating an array of numbers from 1 to 9 (for instance) would never work:

```
new Array(10).map((_, i) => i + 1) // hole × 10
// not 1, 2, 3 … etc 😢
```

And then `filter` is just like, "meh, screw you all, no one wants sparse arrays", so it just removes them no matter what:

```
const a = [ undefined, , true ];
const res = a.filter(() => true);
console.log(res) // [ undefined, true ] - no ho
```

…which, I guess makes sense, since the result of a filter is every element whose callback returned a truthy value, and since the hole wasn't invited to the party (as we saw in `forEach`), then it's never even considered, so the `filter` result is never sparse.

Walking in sparseland

There's two ways to guarantee iterating over the elements that don't exist in a sparse array. The first is a "classic" loop (I'm using a `for`, but a `while` or `do/while` will work too):

```
const a = new Array(3);

for (let i = 0; i < a.length; i++) {
  console.log(i, a[i]) // logs 0…2 + undefined
}
```

In addition, there's the new ES6 array methods `values` and `keys`, both of which include sparse elements. This also means that array spread will translate holes into `undefined`. This is because under the hood, array spread is using the iterator protocol.

This means that if you intend to copy an array, you should be very wary of spread and possible stick to slice.

```
const a = [ 1, , ];
const b = [ ...a ];
const c = a.slice();

expect(a).toEqual(b); // false
expect(a).toEqual(c); // true
```

Here's a "clever" (read: not very good or even remotely clever) emoji sequence to help you in the future: 🔪🙅🍞👍

Sometimes sparse is okay

As I mentioned earlier, a sparse array is going to outperform to generation time when compared to using `Array.from({ length })`. Here's some very rough timings I got from Chrome Canary:

```
const length = 1000000; // 1 million 🤪
new Array(length); // ~5ms
Array.from({ length }) // ~150ms
```

The reason `Array.from` is so much slower, is that it's populated with real `undefined` values, and not holes. For typical daily use cases, I suspect that it won't really matter which you use. If you were working with something that handles a lot of data, perhaps audio samples, then a sparse array is clearly (I think) the way to go.

In summary aka TLDR

A trailing comma in array syntax is simply ignored.

```
[ 1, 2, 3, ] // no hole at the end, just a regu
```

Empty values between commas create holes and thus sparse arrays - these are known as: holes, empty or an elision (apparently)

```
[ 1, , 2 ] // hole at index(1) aka empty
```

Detecting a hole is done using `array.hasOwnProperty(index)`

```
[ 1, , 2 ].hasOwnProperty(1) // false: index(1)
```

Iterating methods, such as `map`, `forEach`, `every`, etc won't call your callback for the hole

```
const a = new Array(1000);
let ctr = 0;
a.forEach(() => ctr++);
console.log(ctr); // 0 - the callback was never
```

`map` will return a new array including the holes

```
[ 1, , 2 ].map(x => x * x) // [ 1, <empty>, 4 ]
```

`filter` will return a new array excluding the holes

```
[ 1, , 2 ].filter(x => true) // [ 1, 2 ]
```

`keys` and `values` return iterator functions that do iterate over hole (ie. includes them in a `for key of array.keys()`)

```
const a = [ 'a', , 'b' ];
for (let [index, value] of a.entries()) {
  console.log(index, value);
}
/* logs:
- 0 'a'
- 1 undefined
- 2 'b'
*/
```

Array spread `[...array]` will transform holes into `undefined` (which will also increase memory and affect performance)

```
[...[ 'a', , 'b' ]] // ['a', undefined, 'b']
```

Large sparse array creation is fast - much faster than `Array.from`.

```
const length = 10000000; // 10 million
new Array(length); // quick
Array.from({ length }) // less quick
```

In practise though, for me, it just helps to remember these pitfalls (a pun, yes), as they rarely cause me any real trouble day-to-day.

Posted 26-Jun 2018 under code.

Comments

Comments Community 1 Login

Recommend 9 Share Sort by Newest

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

thekaleb • 13 days ago

The concept of sparse arrays may make creating large arrays quick, but it slows down array methods like map. The Lodash library ignores sparseness and is faster than the native methods because of that design decision.

Reply • Share ›

Carlos Roberto Gomes Junior • 15 days ago

Great explanation!

Reply • Share ›

Serhii Palash • 15 days ago

const hasUndefined = [1, , 3, , 4,].some((item, index) => {
console.log('index ' + index)

return item === undefined
})
// index 0
// index 2
// index 4

console.log(hasUndefined) // false

----

const findResult = [1, , 3, , 4,].find((item, index) => {
console.log('index ' + index)

return item === undefined

})
// index 0
// index 1

console.log(findResult) // undefined

Seems like that the "find" method is the only one that goes over the holes.

Reply • Share ›

Evgeny Naidenyshev • 16 days ago

Does sparse arrays allocates same memory as regular?
E.g.
a = new Array(1000)
a[999] = 1
How much memory it will be allocated?

Reply • Share ›

Dannii Willis • 19 days ago

Manually changing the length property is another way to introduce holes isn't it?

1 • Reply • Share ›

Invalid Username • 22 days ago

I enjoyed this!

Reply • Share ›

Andy H • a month ago

It feels inconsistent that map and filter shouldn't have the same output in your example.

Reply • Share ›

rem Mod Andy H • a month ago

Not sure I follow… map and filter would definitely have different outputs.

1 • Reply • Share ›

ALSO ON **REMY SHARP'S B:LOG**

To gist and back again

3 comments • 7 months ago

jasonkarns — Yep, jq is awesome!

Opening Windows again (part 1)

15 comments • a year ago

csixty4 — There's no good built-in way to enter special characters. You can open up

When helpful turns into super bad security!

2 comments • 8 hours ago

My extinguished kindle

1 comment • 2 months ago

Archive

- All years
- 2018
- 2017
- 2016
- 2015
- 2014
- 2013
- 2012
- 2011
- 2010
- 2009
- 2008
- 2007
- 2006

Links

- About Remy
- Work with Remy
- Subscribe
- Ethos (WIP)
- House Rules
- The Attic
- RSS feed
- Search
- Random post
- On GitHub
- On Twitter
- Presentations

Remy Sharp

Follow @rem 49.2K followers

I'm a JavaScript developer working professionally on the web since 1999. I run my own consultancy, build products, run training, speak at conferences and curate the UK's best JavaScript conference.

License

this blog is available as open source on GitHub.