Deleting elements from an JavaScript array and closing up the gaps

devblogs.microsoft.com/oldnewthing/20140818-00

August 18, 2014

Raymond Chen

Today's Little Program is an exercise that solves the following problem:

> Given a JavaScript array `a` and an unsorted array `indices` (possibly containing duplicates), calculate the result of deleting all of the elements from the original array as specified by the indices. For example, suppose `a = ["alice", "bob", "charles", "david", "eve"]` and `indices = [2, 4, 2, 0]`.
>
> | | |
> |---|---|
> | `a[0]` | `= "alice";` |
> | `a[1]` | `= "bob";` |
> | `a[2]` | `= "charles";` |
> | `a[3]` | `= "david";` |
> | `a[4]` | `= "eve";` |
> | `a.length` | `= 5;` |
>
> The indices specify that elements 2 (charles), 4 (eve), 2 (charles again, redundant), and 0 (alice) should be deleted from the array, leaving bob and david.

Now, if you had to delete only one element from the array, it is pretty simple:

```
a.splice(n, 1);
```

The trick with removing multiple elements is that deleting one element shifts the indices, which can throw off future calculations. One solution is to remove the highest-indexed element first; in other words, operate on the indices in reverse sorted order.

```
indices.sort().reverse().forEach(function(n) { a.splice(n, 1); });
```

This technique does still suffer from the problem that if there are duplicates in the indices, extraneous elements get deleted by mistake.

Another approach is to reinterpret the problem by focusing not on the deletion but on the survivors: Produce the array consisting of elements whose indices are not in the list of indices to be deleted.

```
a = a.filter(function(e, i) { return indices.indexOf(i) >= 0; });
```

The above approach works well if the list of indices to be deleted is short, but it gets quite expensive if the list is long.

My approach is to use the fact that JavaScript arrays can be sparse. This is a side effect of the fact that JavaScript array indices are actually object properties; the only thing that makes arrays different from generic objects in a language-theoretic sense is the magic `length` property:

- If a new property is added, and the property name is the stringification of a whole number, then the `length` is updated to the numeric value of the added property name, plus 1.
- If the `length` property is modified programmatically, the new value must be a whole number, and all properties which are the stringification of a whole number greater than or equal to the new `length` are deleted.

(See ECMA-262 sections 15.4, 15.4.5.1, and 15.4.5.2 for nitpicky details.)

The first step, then, is to delete all the indices that need to be deleted.

```
indices.forEach(function(n) { delete a[n]; });
```

When applied to our sample data, this leaves

`a[1]`	`= "bob";`
`a[3]`	`= "david";`
`a.length`	`= 5;`

which gets printed in a rather goofy way: `a = [, "bob", , "dave", ]`.

The next step is to close up the gaps. We take advantage of the fact that the `Array` enumeration methods operate on indices 0 through `length - 1` and that they *skip missing elements*. Therefore, I can simply apply a dummy filter:

```
a = a.filter(function() { return true; });
```

Exercise: What is the difference (aside from performance) between `a.push(1);` and `a = a.concat(1);`? How is this question relevant to today's exercise?

Raymond Chen

Follow