Seth Barden Follow
Budding Developer, bassist, dude
Dec 19, 2017 · 4 min read

Strange Ways to Flatten Arrays

There are times in my first week and half of JavaScript at the Flatiron School that I miss Ruby. It's as if Ruby was home and there was no place like it. But just because Ruby was my first programming language, does not mean that I can't keep clicking my heels in JavaScript!

Let's go! Let's grab an array!

```
let arr = [1,2,3,[4,5]]
```

Let’s add 3 to each number in the array! Map it out!

```
arr.map(function (num) {
 return num + 3
})

// [4, 5, 6, "4,53"]
```

That is just what I expected…wait. Is the that last element a funky string?!

Ok, ok, don’t panic. Let’s just flatten that array and add three to each element. JavaScript must have a handy .flatten method like good ol Ruby.

```
arr.flatten
// undefined
```

Oofs. There is no flatten method in JavaScript! This is going to get messy! But have no fear, we have several ways we can flatten an array!

```
[].concat.apply([], arr)
//[1, 2, 3, 4, 5]
```

This works! What is going on here?

Concat is a nice little method that merges two or more arrays by making a copy of the function receiver and merging the arguments into it. It will also merge a single value into an array that it is called upon. It is non destructive. Think ‘concatenate’. Below is the general syntax:

```
[1, 2, 3].concat([4, 5, 6])
// [1, 2, 3, 4, 5, 6]
```

Concat works great if your data all sits in the inside of the array. Nested arrays are a little different story since we are dealing with multiple layers.

.apply() helps us the rest of the way if an array has more a nested layer. .apply() takes the function that it is called on and takes its first argument as the receiver of that call. .apply()’s second argument is an array of arguments to be passed into the function upon which .apply() is called upon, in this case, concat(). Because .apply() is expecting an array full of arguments as its second argument, it effectively strips one

level of the the array passed in and sends it to concat to merge into a new array.

```
[].concat.apply([], arr)
//[1, 2, 3, 4, 5]
```

Unfortunately, this only works for level of nested arrays. If we are nested any more deeply than two layers, we can't get out from under concat.apply()

Another method! Reduce!

```
let arr = [1,2,3,[4,5]]
arr.reduce(function(a,b) {
  return a.concat(b)
  }, [])
// [1, 2, 3, 4, 5]
```

.reduce() takes a callback function with two arguments. The first argument 'a' is the accumulator. The second value is the current value.

The empty [] outside of the callback is the initial value. If the initial value is provided, than the accumulator is set equal to the initial value. If it is not provided then the the accumulator(a) will be equal to arr[0] (in this case) and the current value (b) will be equal to arr[1]. Then the function in the callback is executed on each element of the array. Above, the concat function is applied to the accumulator value([]) and to the current value(arr[0]). Long hand this call back function will operate like so:

```
[].concat(1) // First element in arr
[1].concat(2)
[1, 2].concat(3)
[1, 2, 3].concat([4, 5]) // The last element is an array
[1, 2, 3, 4, 5] // reduce has finished it's work with the
help of concat!
```

Alright, reduce! Good work! But reduce can't handled arrays, nested in arrays, nested in arrays either. This sounds like a custom job using our new friend concat!

```
arr = [[1],[2,[3,[4,[5,6], 7]], 8], 9, 10]


function flatten(array) {
  let result = []
  while (array.length) {
    let current = array.shift()
    if (Array.isArray(current)){
      array = current.concat(array)
    } else {
      result.push(current)
    }
  }
 return result
}
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here, we have a custom function called flatten that takes an array for an argument. Since 0 results in a false boolean, we set up a while conditional that will only evaluate to true if there are some sweet elements in our array. Next, we use the shift() method to test the first

element of the array. If the element is an array, we reset our array argument to current.concat(array). That takes our first element (which is an array) and *merges* the remaining elements (some of which are arrays and some not) into it, giving us a new array, but one level more shallow! If the first element is not an array, it will be pushed into the result array. This process will continue until array.length === 0, which means that all of our elements have been taken out of the original array and pushed into the result array. Finally, we return a nice, flat array, in order of breadth. Yay!

Take that Ruby!