Flatten Completed

Zhenghao He
Engineering Manager, Robinhood

</> Medium 20mins 8.97k done

Implement a function `flatten` that returns a newly-created array with all sub-array elements concatenated recursively into a single level.

Examples

```
// Single-level arrays are unaffected.
flatten([1, 2, 3]); // [1, 2, 3]

// Inner arrays are flattened into a single level.
flatten([1, [2, 3]]); // [1, 2, 3]
flatten([
  [1, 2],
  [3, 4],
]); // [1, 2, 3, 4]

// Flattens recursively.
flatten([1, [2, [3, [4, [5]]]]]); // [1, 2, 3, 4, 5]
```

This is a common JavaScript interview question. It tests one's knowledge about checking for the array type, looping through an array, various native methods such as `Array.prototype.concat`, and recursion.

Clarification questions

1. What type of data does the array contain? Some approach only applies to certain data types.
2. How many levels of nesting can this array have? If there are thousands-of-levels of nesting, recursion might not be a good idea given its big upfront memory footprint.
3. Should we return a new array or we mutate the existing array?
4. Can we assume valid input, i.e. an array. Normally the answer is "yes", so you don't have to waste your time doing defensive programming.
5. Does the environment the code runs on has ES6+ support? The environment determines what methods/native APIs you have access to.

Solution

Solution

Approach 1: Iterative

First let's think through how we should check if a given value is an array or not. We can use `Array.isArray` or `instanceof Array` to achieve that. There are some nuances between these two. If you are interested you can check out this article by Jake Archibald.

Then, we don't want to mutate the original input array, which is a good practice in general, so we will make a copy of the input array and return a new array with all the items extracted from the input array.

Here is the solution. We loop through the array with a while loop and we take out an item from the array one at a time to check to see if that item is an array. If the item is not an array, we put it into the `res` array. If it is an array, we use a spread operator `...` to get all the items out of it and put them back to the array.

```
type ArrayValue = any | Array<ArrayValue>;

export default function flatten(value: Array<ArrayValue>): Array<any> {
  const res = [];
  const copy = value.slice();

  while (copy.length) {
    const item = copy.shift();
    if (Array.isArray(item)) {
      copy.unshift(...item);
    } else {
      res.push(item);
    }
  }

  return res;
}
```

Approach 2: Iteration using `Array.prototype.some`

A more concise approach, compared to the previous one, is to use `Array.prototype.some`.

```
type ArrayValue = any | Array<ArrayValue>;

export default function flatten(value: Array<ArrayValue>): Array<any> {
  while (value.some(Array.isArray)) {
    value = [].concat(...value);
```

```
  }

  return value;
}
```

Approach 3: Recursion using `Array.prototype.reduce`

A recursion approach fits well here given the recursive and nesting nature of this question. And it will simplify our code a lot.

JavaScript TypeScript

```
/**
 * @param {Array<*|Array>} value
 * @return {Array}
 */
export default function flatten(value) {
  return value.reduce(
    (acc, curr) => acc.concat(Array.isArray(curr) ? flatten(curr) : curr),
    [],
  );
}
```

Although a recursive approach always has the risk overflowing the call stack, as of this writing, in chrome, the number of recursive calls you can make is around 10 thousands and in Firefox it is 50 thousands, so this shouldn't be a problem in practice. However, when the cost of recursion becomes a concern, we can use a generator to lazily extract the flattened items from the array. We will see that solution later.

Approach 4: Flatten the array in-place

All the solutions we have seen so far are returning a new flattened array without mutating the original input array. Again, this is normally what you want.

However, the interviewer might ask you to implement an in-place solution that doesn't allocate extra memory. That is, a solution with a constant `O(1)` space complexity.

In this case, you will need to leverage array methods that *mutate*. There are 9 methods in total that mutate arrays: `pop`, `push`, `reverse`, `shift`, `sort`, `splice`, `unshift`, `copyWithin` and `fill`.

Here is one possible solution that uses `splice` to mutate the input array:

```
type ArrayValue = any | Array<ArrayValue>;

export default function flatten(value: Array<ArrayValue>): Array<any> {
  for (let i = 0; i < value.length; ) {
    if (Array.isArray(value[i])) {
      value.splice(i, 1, ...value[i]);
    } else {
      i++;
    }
  }

  return value;
}
```

Approach 5: Recursive approaching using `flatMap`

The `flatMap` function method returns a new array formed by applying a given callback function to each element of the array, and then flattening the result by one level. By calling it recursively, we can flatten the entire array until it is only one level deep.

```
type ArrayValue = any | Array<ArrayValue>;

export default function flatten(value: Array<ArrayValue>): Array<any> {
  return Array.isArray(value) ? value.flatMap((item) => flatten(item)) : value;
}
```