Vicky Lai Follow
I'm a nomadic developer who geeks out over 90s stuff and drinks too much coffee. My dream title is "Le...
May 30 · 5 min read

Understanding Array.prototype.reduce() and recursion using apple pie

Delicious, attention-retaining apple pie.

I was having trouble understanding `reduce()` and recursion in JavaScript, so I wrote this article to explain it to myself (hey, look, recursion!). These concepts share some similarities with making apple pies. I hope you find my examples both helpful and delicious.

Given an array with nested arrays:

```
var arr = [1, [2], [3, [[4]]]]
```

We want to produce this:

```
var flat = [1, 2, 3, 4]
```

Using for loops and if statements

If we know the maximum number of nested arrays we'll encounter (there are 4 in this example), we can use `for` loops to iterate through each array item, then `if` statements to check whether that item is in itself an array, and so on…

```
function flatten() {
    var flat = [];
    for (var i=0; i<arr.length; i++) {
    if (Array.isArray(arr[i])) {
        for (var ii=0; ii<arr[i].length; ii++) {
        if (Array.isArray(arr[i][ii])) {
            for (var iii=0; iii<arr[i][ii].length; iii++) {
            for (var iiii=0; iiii<arr[i][ii][iii].length; i
                if (Array.isArray(arr[i][ii][iii])) {
                flat.push(arr[i][ii][iii][iiii]);
                } else {
                flat.push(arr[i][ii][iii]);
                }
            }
            }
        } else {
            flat.push(arr[i][ii]);
```

…Which works, but is both hard to read and harder to understand. Besides, it only works if you know how many nested arrays to process, and can you imagine having to debug this mess?! (Gee, I think there's an extra `i` somewhere.)

Using reduce

JavaScript has a couple methods we can use to make our code more concise and easier to follow. One of these is `reduce()` and it looks like this:

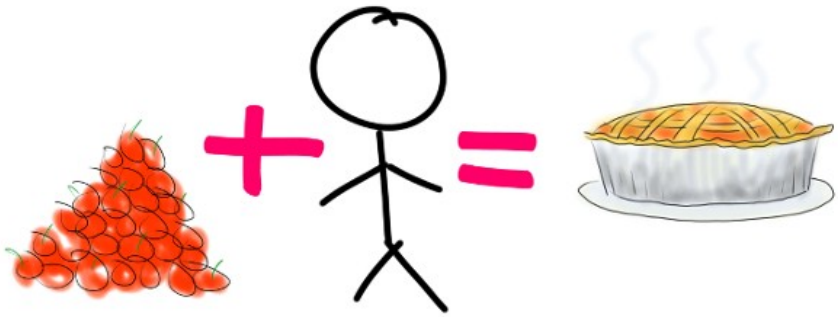
```
var flat = arr.reduce(function(done,curr){
    return done.concat(curr);
}, []);

```

It's a lot less code, but we haven't taken care of some of the nested arrays. Let's first walk through `reduce()` together and examine what it does to see how we'll correct this.

> *Array.prototype.reduce()*
>
> *The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value. (MDN)*

It's not quite as complicated as it seems. Let's think of `reduce()` as an out-of-work developer (AI took all the dev jobs) with an empty basket. We'll call him Adam. Adam's main function is now to take apples from a pile, shine them up, and put them one-by-one into the basket. This basket of shiny apples is destined to become delicious apple pies. It's a very important job.

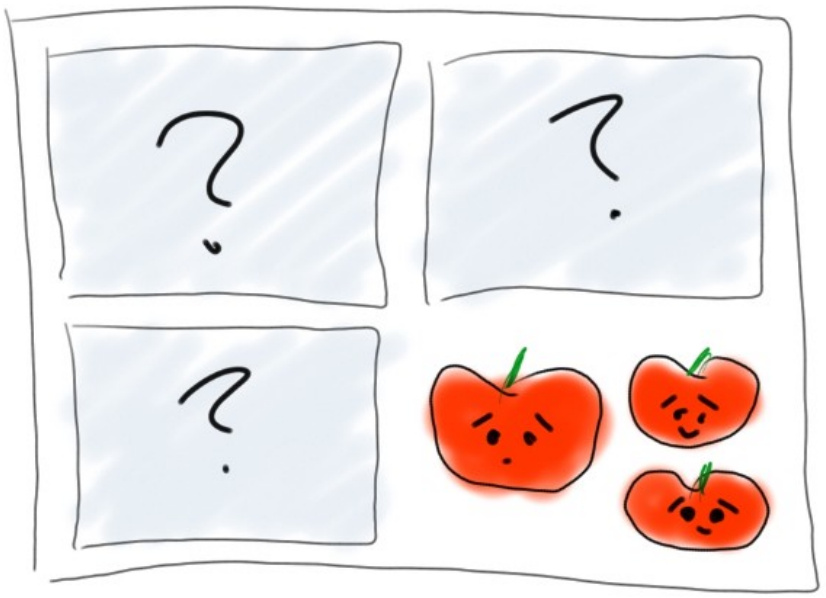


Apples plus human effort equals pie. Not to be confused with apple-human-pie, which is less appetizing.

In our above example, the pile of apples is our array, `arr`. Our basket is `done`, the accumulator. The initial value of `done` is an empty array, which we see as `[]` at the end of our reduce function. The apple that Adam is currently shining, you guessed it, is `curr`. Once Adam processes the current apple, he places it into the basket (`.concat()`). When there are no more apples in the pile, he returns the basket of polished apples to us, and then goes home to his cat.

Using reduce recursively to address nested arrays

So that's all well and good, and now we have a basket of polished apples. But we still have some nested arrays to deal with. Going back to our analogy, let's say that some of the apples in the pile are in boxes. Within each box there could be more apples, and/or more boxes containing smaller, cuter apples.



Adorable, slightly skewed apples just want to be loved/eaten.

Here's what we want our apple-processing-function/Adam to do:

1. If the pile of apples is a pile of apples, take an apple from the pile.
2. If the apple is an apple, polish it, put it in the basket.
3. If the apple is a box, open the box. If the box contains an apple, go to step 2.
4. If the box contains another box, open this box, and go to step 3.
5. When the pile is no more, give us the basket of shiny apples.
6. If the pile of apples is not a pile of apples, give back whatever it is.

A recursive reduce function that accomplishes this is:

```
function flatten(arr) {
  if (Array.isArray(arr)) {
  return arr.reduce(function(done,curr){
    return done.concat(flatten(curr));
    }, []);
  } else {
    return arr;
  }
```

Bear with me and I'll explain.

> *Recursion*
>
> *An act of a function calling itself. Recursion is used to solve problems that contain smaller sub-problems. A recursive function can receive two inputs: a base case (ends recursion) or a recursive case (continues recursion). (MDN)*

If you examine our code above, you'll see that `flatten()` appears twice. The first time it appears, it tells Adam what to do with the pile of apples. The second time, it tells him what to do with the thing he's currently holding, providing instructions in the case it's an apple, and in the case it's not an apple. The thing to note is that these instructions are a *repeat of the original instructions we started with* - and that's recursion.

We'll break it down line-by-line for clarity:

1. `function flatten(arr) {` - we name our overall function and specify that it will take an argument, `arr`.
2. `if (Array.isArray(arr)) {` - we examine the provided "arrgument" (I know, I'm very funny) to determine if it is an array.
3. `return arr.reduce(function(done,curr){` - if the previous line is true and the argument is an array, we want to reduce it. This is our recursive case. We'll apply the following function to each array item…
4. `return done.concat(flatten(curr));` - an unexpected plot twist appears! The function we want to apply is the very function we're in. Colloquially: take it from the top.
5. `}, []);` - we tell our reduce function to start with an empty accumulator (`done`), and wrap it up.
6. `} else {` - this resolves our if statement at line 2. If the provided argument isn't an array…
7. `return arr;` - return whatever the `arr` is. (Hopefully a cute apple.) This is our base case that breaks us out of recursion.

8. `}` - end the else statement.
9. `}` - end the overall function.

And we're done! We've gone from our 24 line, 4-layers-deep nested `for` loop solution to a much more concise, 9 line recursive reduce solution. Reduce and recursion can seem a little impenetrable at first, but they're valuable tools that will save you lots of future effort once you grasp them.

And don't worry about Adam, our out-of-work developer. He got so much press after being featured in this article that he opened up his very own AI-managed apple pie factory. He's very happy.

+1 for you if you saw that one coming.

. . .

Thanks for reading! You can find more articles explaining coding concepts with food on my blog.