14th February 2020

Currying

Currying is an advanced technique of working with functions. It's used not only in JavaScript, but in other languages as well.

Currying is a transformation of functions that translates a function from callable as `f(a, b, c)` into callable as `f(a)(b)(c)`.

Currying doesn't call a function. It just transforms it.

Let's see an example first, to better understand what we're talking about, and then practical applications.

We'll create a helper function `curry(f)` that performs currying for a two-argument `f`. In other words, `curry(f)` for two-argument `f(a, b)` translates it into a function that runs as `f(a)(b)`:

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// usage
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

As you can see, the implementation is straightforward: it's just two wrappers.

- The result of `curry(func)` is a wrapper `function(a)`.
- When it is called like `sum(1)`, the argument is saved in the Lexical Environment, and a new wrapper is returned `function(b)`.
- Then this wrapper is called with `2` as an argument, and it passes the call to the original `sum`.

More advanced implementations of currying, such as _.curry from lodash library, return a wrapper that allows a function to be called both normally and partially:

```
function sum(a, b) {
  return a + b;
}
```

```
4
5 let curriedSum = _.curry(sum); // using _.curry from lodash library
6
7 alert( curriedSum(1, 2) ); // 3, still callable normally
8 alert( curriedSum(1)(2) ); // 3, called partially
```

Currying? What for?

To understand the benefits we need a worthy real-life example.

For instance, we have the logging function `log(date, importance, message)` that formats and outputs the information. In real projects such functions have many useful features like sending logs over the network, here we'll just use `alert` :

```
1 function log(date, importance, message) {
2   alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}
3 }
```

Let's curry it!

```
1 log = _.curry(log);
```

After that `log` works normally:

```
1 log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

…But also works in the curried form:

```
1 log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Now we can easily make a convenience function for current logs:

```
1 // logNow will be the partial of log with fixed first argument
2 let logNow = log(new Date());
3
4 // use it
5 logNow("INFO", "message"); // [HH:mm] INFO message
```

Now `logNow` is `log` with fixed first argument, in other words "partially applied function" or "partial" for short.

We can go further and make a convenience function for current debug logs:

```
1 let debugNow = logNow("DEBUG");
2
3 debugNow("message"); // [HH:mm] DEBUG message
```

So:

1. We didn't lose anything after currying: `log` is still callable normally.
2. We can easily generate partial functions such as for today's logs.

Advanced curry implementation

In case you'd like to get in to the details, here's the "advanced" curry implementation for multi-argument functions that we could use above.

It's pretty short:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

Usage examples:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying
```

The new `curry` may look complicated, but it's actually easy to understand.

The result of `curry(func)` call is the wrapper `curried` that looks like this:

```
// func is the function to transform
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function pass(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

When we run it, there are two `if` execution branches:

1. Call now: if passed `args` count is the same as the original function has in its definition (`func.length`) or longer, then just pass the call to it.
2. Get a partial: otherwise, `func` is not called yet. Instead, another wrapper `pass` is returned, that will re-apply `curried` providing previous arguments together with the new ones. Then on a new call, again, we'll get either a new partial (if not enough arguments) or, finally, the result.

For instance, let's see what happens in the case of `sum(a, b, c)` . Three arguments, so `sum.length = 3` .

For the call `curried(1)(2)(3)` :

1. The first call `curried(1)` remembers `1` in its Lexical Environment, and returns a wrapper `pass` .
2. The wrapper `pass` is called with `(2)` : it takes previous args (`1`), concatenates them with what it got `(2)` and calls `curried(1, 2)` with them together. As the argument count is still less than 3, `curry` returns `pass` .
3. The wrapper `pass` is called again with `(3)` , for the next call `pass(3)` takes previous args (`1` , `2`) and adds `3` to them, making the call `curried(1, 2, 3)` – there are `3` arguments at last, they are given to the original function.

If that's still not obvious, just trace the calls sequence in your mind or on paper.

Fixed-length functions only

The currying requires the function to have a fixed number of arguments.

A function that uses rest parameters, such as `f(...args)` , can't be curried this way.

A little more than currying

By definition, currying should convert `sum(a, b, c)` into `sum(a)(b)(c)` .

But most implementations of currying in JavaScript are advanced, as described: they also keep the function callable in the multi-argument variant.

Summary

Currying is a transform that makes `f(a,b,c)` callable as `f(a)(b)(c)` . JavaScript implementations usually both keep the function callable normally and return the partial if the arguments count is not enough.

Currying allows us to easily get partials. As we've seen in the logging example, after currying the three argument universal function `log(date, importance, message)` gives us partials when called with one argument (like `log(date)`) or two arguments (like `log(date, importance)`).

Previous lesson Next lesson

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)