A Drip of JavaScript

Partial Application with Function#bind

Originally published in the A Drip of JavaScript newsletter.

In last week's drip, we covered using `bind` to create bound functions. But `bind` is also a very convenient way of implementing partial application. First, let's take a look at exactly what partial application is.

According to Wikipedia, partial application "refers to the process of fixing a number of arguments to a function, producing another function of smaller arity." Arity refers to the number of arguments that a function takes.

Here's an example:

```
function multiply (x, y) {
    return x * y;
}

function double (num) {
    return multiply(2, num);
}
```

That's a very rudimentary form of partial application, where inside `double` we permanently fix `multiply`'s first argument as `2`, producing a new function `double`.

But explicitly declaring the body of the new function isn't actually necessary if you use `bind`. Here's how we could rewrite that:

```
function multiply (x, y) {
    return x * y;
}

var double = multiply.bind(null, 2);
```

As we covered last time, the first argument to `bind` sets it's internal `this` value. Since our function doesn't depend on `this`, we just pass in `null`. But any subsequent arguments that we supply will be used to permanently fix the arguments of the function we are binding. In this case, we are only fixing one argument, but we can potentially fix as many as we want.

```
function greet (salutation, person, delivery) {
    var message = '"'+ salutation + ', ' + person + ',"' ' +
                  delivery + ' the greeter.';

    console.log(message);
}

var hail = greet.bind(null, "Hail");

// Outputs: '"Hail, Lord Elrond," said the greeter.'
hail("Lord Elrond", "said");

var begone = greet.bind(null, "Begone", "Wormtongue", "commanded");

// Outputs: '"Begone, Wormtongue," commanded the greeter.'
begone();
```

One thing you may have noticed is that `bind` always fixes the arguments from left to right. This means that it isn't suitable for all forms of partial application, but it does cover the most common ones.

You may be wondering if partial application is actually all that useful in real code. To answer, let's revisit an example from our discussion of dispatch tables:

```
var commandTable = {
    north:    function() { movePlayer("north"); },
    east:     function() { movePlayer("east");  },
    south:    function() { movePlayer("south"); },
    west:     function() { movePlayer("west");  },
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}
```

In this example we are taking user input from a text adventure game, and then calling the appropriate command from `commandTable`. As you can see, we're using the same rudimentary form of partial application that we saw in our first example.

Using `bind` we can clean this up a bit.

```
var commandTable = {
    north:    movePlayer.bind(null, "north"),
    east:     movePlayer.bind(null, "east"),
    south:    movePlayer.bind(null, "south"),
    west:     movePlayer.bind(null, "west"),
    look:     describeLocation,
    backpack: showBackpack
};

function processUserInput(command) {
    commandTable[command]();
}
```

As I mentioned last time, `bind` isn't available in IE8 and older, so you may want to use a polyfill or library to achieve the same functionality. If you only want to implement partial application and don't need to create bound functions, you might want to consider the `partial` methods in Underscore and Lo-Dash.

Thanks for reading!

Josh Clanton
