Coroutine Event Loops in Javascript

ECMAScript 6 introduces a `yield` keyword for implementing generators and coroutines.

An intriguing use of coroutines is to implement event loops as an alternative to callback functions. This is particularly relevant to Javascript, where the use of callbacks is pervasive.

If you're in a hurry: try the interactive demo contrasting coroutines and callbacks.

The Basics

A coroutine is a function, marked by the `function*` syntax, which can suspend itself anywhere in its execution using the `yield` keyword.

It can then be resumed by *sending* it a value. The sent value will appear as the return value of `yield` and execution will resume until the next `yield`.

For example we can create a simple coroutine which will yield twice and print out the values that are sent to it after each yield:

```
function* test() {
    console.log('Hello!');
    var x = yield;
    console.log('First I got: ' + x);
    var y = yield;
    console.log('Then I got: ' + y);
}
```

To initiate an instance of a coroutine you just call the function, which returns a coroutine object:

```
var tester = test();
```

So far none of the function will have executed, so we must execute the function until it hits a `yield`, by calling:

```
tester.next(); // prints 'Hello!'
```

Now `tester` is paused at the first `yield`, and we can resume the coroutine by sending a value to the `yield` where it is paused:

```
tester.next('a cat'); // prints 'First I got: a cat'
```

Now `tester` is paused at the second `yield`, and we can send another value:

```
tester.next('a dog'); // prints 'Then I got: a dog'
```

That's about all there is to coroutines—they are functions which can suspend themselves using the `yield` keyword, and you can communicate with them by sending values, which will be returned as the value of `yield` and resume execution.

The Convenient `coroutine` Wrapper

Every time you use a coroutine you always do three things:

1. Call the function and store the resulting coroutine object.
2. Call `next()` on the coroutine object, to execute until the first `yield`.
3. After that all you can do is call `next(...)` to run the coroutine and send it values.

We can put all of this functionality into a wrapper function:

```
function coroutine(f) {
    var o = f(); // instantiate the coroutine
    o.next(); // execute until the first yield
    return function(x) {
        o.next(x);
    }
}
```

This sets up the coroutine and returns a function which we can call directly instead of calling `next` on the coroutine object.

Using this wrapper, our earlier example becomes:

```
var test = coroutine(function*() {
    console.log('Hello!');
    var x = yield;
    console.log('First I got: ' + x);
    var y = yield;
    console.log('Then I got: ' + y);
});
// prints 'Hello!'

test('a dog'); // prints 'First I got: a dog'
test('a cat'); // prints 'Then I got: a cat'
```

A Coroutine Loop

Our first example coroutine yields twice and then ends. Let's make a never-ending coroutine instead.

Here's a coroutine that never ends, and each time you resume it, it alternates between ticking and tocking:

```
var clock = coroutine(function*() {
    while (true) {
        yield;
        console.log('Tick!');
        yield;
        console.log('Tock!');
    }
});

clock(); // prints 'Tick!'
clock(); // prints 'Tock!'
clock(); // prints 'Tick!'
```

Note that in Javascript, calling `clock()` with no argument is the same as calling `clock(undefined)`, which in this case is fine since this coroutine doesn't even look at the values being sent to it.

Now let's make the clock tick/tock once every second. It's very easy:

```
setInterval(clock, 1000);
```

This resumes the coroutine once every 1000ms, and it will tick and tock for eternity.

A Coroutine Event Loop

In the same way that we used the `clock` function above as a timer callback, we can use coroutines as callbacks for events, and the event object will be sent to the running coroutine.

For example suppose we want to implement this little demo. Try dragging the box around:

This program needs to repeat the following steps:

1. wait for a `mousedown` on the box
2. process any `mousemove` events, and actually move the box
3. until we see a `mouseup` event

We can implement this as a coroutine that receives events from `yield`:

```
var loop = coroutine(function*() {
    while (true) { // wait for a mousedown
        var event = yield;
        if (event.type == 'mousedown') {
            while (true) { // process mousemoves until a mouseup
                var event = yield;
                if (event.type == 'mousemove') move(event);
                if (event.type == 'mouseup') break;
            }
        }
    }
});
```

Note that the `move` function is defined elsewhere, and just moves the div.

We can be slightly more concise by noticing that event objects are always 'truthy', so we can change the while loops to:

```
while (event = yield) { ... }
```

This form of infinite loop is the heart of a coroutine event loop. We end up with:

```
var loop = coroutine(function*() {
    var event;
    while (event = yield) { // wait for a mousedown
        if (event.type == 'mousedown') {
            while (event = yield) { // process mousemoves until a mouseup
                if (event.type == 'mousemove') move(event);
                if (event.type == 'mouseup') break;
            }
        }
    }
});
```

Coroutines as State Machines

Let's return to the `clock` example earlier. We can implement it using coroutines as above, or we can of course implement it without coroutines, using a function and a variable:

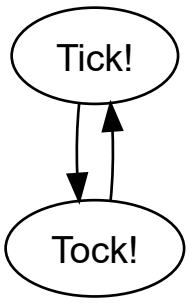
```
var clock = coroutine(function*() {
    while (true) {
        yield;
        console.log('Tick!');
        yield;
        console.log('Tock!');
    }
});
```

clock as a coroutine

```
var ticking = true;
var clock = function() {
    if (ticking)
        console.log('Tick!');
    else
        console.log('Tock!');
    ticking = !ticking;
}
```

clock as a function and a variable

Both implementations behave the same, and implement the same simple 2-state machine, as pictured.



the clock state machine

The interesting difference between the two implementations is that the coroutine version involves no `ticking` variable, and never explicitly stores any state.

Without using coroutines you *must* use a variable to store the state of whether the next call should tick or tock.

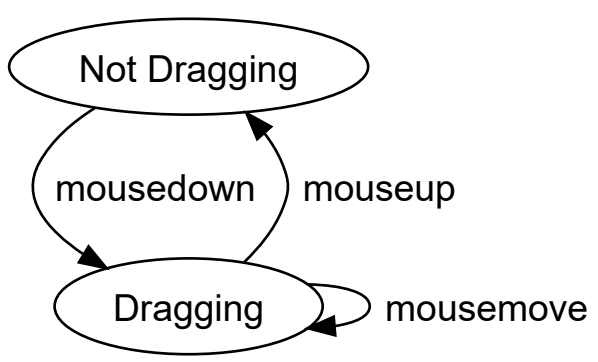
This variable can only be avoided because **coroutines add an entirely new form of state** to the language, namely the state of *where* the coroutine is currently suspended.

One benefit of this 'implicit' coroutine state is that it is guaranteed to be consistent. This differs from the variable state, which could easily become invalid if we do something like `ticking="maybe"`.

As a slightly more complex example, we can also implement the 2-state machine from the dragging demo using standard callback functions.

Once again the coroutine implementation uses no explicit state, while the callback version must.

Also note that the single event loop is equivalent to three callback functions.



the dragging state machine

Drag the box, and observe how the two different implementations behave:

```
var loop = coroutine(function*() {
  var e;
  while (e = yield) {
    if (e.type == 'mousedown') {
      while (e = yield) {
        if (e.type == 'mousemove')
          move(e);
        if (e.type == 'mouseup')
          break;
      }
    }
    // ignore mousemoves
  }
});

$('#box').mousedown(loop);
$(window).mousemove(loop)
  .mouseup(loop);
```

draggable box as a coroutine event loop

```
var dragging = false;
function mousedown(e) {
  dragging = true;
}
function mousemove(e) {
  if (dragging)
    move(e);
  else {
    // ignore mousemoves
  }
}
function mouseup(e) {
  dragging = false;
}

$('#box').mousedown(mousedown);
$(window).mousemove(mousemove)
  .mouseup(mouseup);
```

draggable box as callbacks and a variable

The coroutine implementation above is just a simulation, so that it works in all browsers, some of which still don't support ECMAScript 6.

But if you're using a recent version of Firefox or Chrome, you can try the real live coroutine implementation!

These examples of coroutine event loops are no better than their standard callback function equivalents, but complex state machines could be more elegantly implemented as coroutines, perhaps for example something involving Ajax requests or something in node.js, where almost everything is a callback.

This writeup was inspired by the section "Coroutines and Event Dispatching" starting on page 53 of David Beazley's mind-blowing presentation *A Curious Course on Coroutines and Concurrency*.

There was a good discussion of this writeup on Hacker News.

Harold Cooper, December 2012

view the source on github