Generators

by Forbes Lindesay

Introduction

One of the most exciting features introduced in ES6 is generators. Their primary use case is in representing lazy (possibly infinite) sequences. For example, the following function returns the first n positive integers.

```
function count(n){
  var res = []
  for (var x = 0; x < n; x++) {
    res.push(x)
  }
  return res
}

for (var x of count(5)) {
  console.log(x)
}
```

We can write a very similar function using generators, that returns all the positive integers.

```
function* count(){
  for (var x = 0; true; x++) {
    yield x
  }
}

for (var x of count()) {
  console.log(x)
}
```

What's actually going on here, is that the `count` function is being lazily evaluated, so it pauses at each `yield` and waits until another value is asked for. This means that the for/of loop will execute forever, continually getting the next integer in an infinite list.

The reason this is so exciting, is that we can exploit the ability to pause a function in order to help us write asynchronous code. Specifically, this will allow us to do asynchronous things inside our existing control flow structures, such as loops, conditionals and try/catch blocks.

What generators do **not** do is give us a way of representing the result of an asynchronous operation. For that, we need a promise.

The goal of this article is to teach you to be able to write code like:

```
var login = async(function* (username, password, session) {
  var user = yield getUser(username);
  var hash = yield crypto.hashAsync(password + user.salt);
  if (user.hash !== hash) {
    throw new Error('Incorrect password');
  }
  session.setUser(user);
});
```

Here the code reads just like it would if it were synchronous, but is in fact doing asynchronous work at each of the `yield` keywords. The result of calling the `login` function would be a promise.

How it works - Fulfilling

As you saw in the introduction, we can pause to wait for a promise using the `yield` keyword. What we need now is a way to get fine control over that generator function so as to have it start again once the promise completes. Fortunately, it's possible to step through a generator function via the `.next` method.

```
function* demo() {
  var res = yield 10;
  assert(res === 32);
  return 42;
}

var d = demo();
var resA = d.next();
// => {value: 10, done: false}
var resB = d.next(32);
// => {value: 42, done: true}
//if we call d.next() again it throws an error
```

What's happening here is that we call `d.next()` once to get it to the `yield`, and then when we call `d.next()` a second time, we give it a value that is the result of the `yield` expression. The function can then move on to the `return` statement to return a final result.

We can use this, by calling `.next(result)` to signal that a promise has been fulfilled with result.

How it works - Rejecting

We also need a way to represent a promise that's been yielded being rejected. We can use the `.throw(error)` method on the generator to do this.

```
var sentinel = new Error('foo');
function* demo() {
  try {
    yield 10;
  } catch (ex) {
    assert(ex === sentinel);
  }
}

var d = demo();
d.next();
// => {value: 10, done: false}
d.throw(sentinel);
```

Like before, we call `d.next()` to get to the first `yield` keyword. We can then signal rejection using `d.throw(error)`, which causes the generator to act as though `yield` throw an error. In our example, this will trigger the `catch` block.

How it works - Putting it all together

Putting all of this together, we just have to keep manually moving the generator forwards with the results of any promises it has yielded. We can do that using a simple function like this one:

```
function async(makeGenerator){
  return function () {
    var generator = makeGenerator.apply(this, arguments);

    function handle(result){
      // result => { done: [Boolean], value: [Object] }
      if (result.done) return Promise.resolve(result.value);

      return Promise.resolve(result.value).then(function (res){
        return handle(generator.next(res));
      }, function (err){
        return handle(generator.throw(err));
      });
    }

    try {
      return handle(generator.next());
    } catch (ex) {
      return Promise.reject(ex);
    }
  }
}
```

Note how we use `Promise.resolve` to ensure we are always dealing with well behaved promises and we use `Promise.reject` along with a try/catch block to ensure that synchronous errors are always converted into asynchronous errors.

Further Reading

- MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*) - The mozilla developer network has great documentation on generators.

- MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) - The mozilla developer network has great documentation on promises.
- regenerator (http://facebook.github.io/regenerator/) - A project by Facebook to add generator support to older environments by cross-compiling the code.
- gnode (https://github.com/TooTallNate/gnode) - A command line applicaton to use regenerator to support generators in older versions of node.js
- then-yield (https://github.com/then/yield) - A library that provides functions for writing code that uses promises with generators.
- Task.js (http://taskjs.org/) - An alternative library for using promises with generators.
- YouTube (https://www.youtube.com/watch?v=qbKWsbJ76-s) - A video of my JSConf.eu talk that discusses many of the same things as appear in this article.

← patterns (/patterns/)

implementing → (/implementing/)

Developed by @ForbesLindesay (http://www.forbeslindesay.co.uk)
Can you make this better? Please fork it on GitHub (https://github.com/ForbesLindesay/promisejs.org)