Basic Functional Programming With Async/Await

by Valeri Karpov
@code_barbarian (http://www.twitter.com/code_barbarian)
April 20, 2017

Async/await (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) makes it easy to integrate asynchronous behavior with imperative constructs like for loops, if statements, and try/catch blocks (http://thecodebarbarian.com/common-async-await-design-patterns-in-node.js.html). Unfortunately, it doesn't do the same for functional constructs like `forEach` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach), `map` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map), `reduce` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce), and `filter` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). Using these constructs with async functions leads to behavior that can seem downright baffling. In this article, I'll show you some common gotchas for async functions with JavaScript's built-in functional array methods and how to work around them.

Note: the below code is only tested on Node v7.6.0. Furthermore, the below code is only intended to be a thought experiment and a didactic example. I wouldn't recommend using it in production.

Motivation and `forEach`

In synchronous land, `forEach()` executes a function for each element of the array in order. For example, the below script is guaranteed to print 0-9:

```
function print(n) {
  console.log(n);
}

function test() {
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].forEach(print);
}

test();
```

Unfortunately, things get more subtle with async functions. The below script will print 0-9 in reverse order!

```
async function print(n) {
  // Wait 1 second before printing 0, 0.9 seconds before printing 1, etc.
  await new Promise(resolve => setTimeout(() => resolve(), 1000 - n * 100));
  // Will usually print 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 but order is not strictly
  // guaranteed.
  console.log(n);
}

async function test() {
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].forEach(print);
}

test();
```

Even though both functions are async, Node.js won't wait for the first `print()` call to finish before executing the next one! Can you just add an `await`?

```
async function test() {
  // SyntaxError: Unexpected identifier
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].forEach(n => { await print(n); });
}
```

Nope, can't let you do that Starfox (http://knowyourmeme.com/memes/i-can-t-let-you-do-that-starfox), that's a `SyntaxError`, because `await` must **always** be in an `async` function. At this point, you can just give up and use the non-standard `Promise.series()` function (https://www.npmjs.com/package/promise-series). But, if you remember that `async` functions are just functions that return promises, you can use promise chaining and `.reduce()` your way into getting an in-order `forEach()`.

```
async function print(n) {
  await new Promise(resolve => setTimeout(() => resolve(), 1000 - n * 100));
  console.log(n);
}

async function test() {
  // This is where the magic happens. Each `print()` call returns a promise,
  // so calling `then()` chains them together in order and prints 0-9 in order.
  await [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].
    reduce((promise, n) => promise.then(() => print(n)), Promise.resolve());
}

test();
```

You can also wrap this functionality into a handy `forEachAsync()` function:

```
async function print(n) {
  await new Promise(resolve => setTimeout(() => resolve(), 1000 - n * 100));
  console.log(n);
}

Array.prototype.forEachAsync = function(fn) {
  return this.reduce((promise, n) => promise.then(() => fn(n)), Promise.resolve());
};

async function test() {
  await [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].forEachAsync(print);
}

test();
```

Chaining with `map()` and `filter()`

One big advantage of JavaScript's functional array constructs is chaining. Suppose you're given an array of ids and you want to pull the documents that correspond to those ids, filter out the ones that are already in another database, and save all of those. You can do this without any functional primitives, but it will involve a lot of intermediate values.

```
const { MongoClient } = require('mongodb');

async function copy(ids, db1, db2) {
  // Find all docs from db1
  const fromDb1 = await db1.collection('Test').find({ _id: { $in: ids } }).sort({ _id: 1 }).toArray();
  // And db2
  const fromDb2 = await db2.collection('Test').find({ _id: { $in: ids } }).sort({ _id: 1 }).toArray();

  // Find all docs from db1 that aren't in db2
  const toInsert = [];
  for (const doc of fromDb1) {
    if (!fromDb2.find(_doc => _doc._id === doc._id)) {
      toInsert.push(doc);
      console.log('Insert', doc);
    }
  }
  // And insert all of them
  await db2.collection('Test').insertMany(toInsert);
}

async function test() {
  const db1 = await MongoClient.connect('mongodb://localhost:27017/db1');
  const db2 = await MongoClient.connect('mongodb://localhost:27017/db2');
  await db1.dropDatabase();
  await db2.dropDatabase();

  const docs = [
    { _id: 1 },
    { _id: 2 },
    { _id: 3 },
    { _id: 4 }
  ];

  await db1.collection('Test').insertMany(docs);
  // Only insert docs with _id 2 and 4 into db2
  await db2.collection('Test').insertMany(docs.filter(doc => doc._id % 2 === 0));

  await copy(docs.map(doc => doc._id), db1, db2);
}

test();
```

Functional would make this a lot cleaner - you'd just need `ids.map().filter().forEach()`, but each of those `map()`, `filter()`, and `each()` need to handle async functions. We already have `forEachAsync()`, implementing `mapAsync()` and `filterAsync()` is similar:

```
Array.prototype.mapAsync = function(fn) {
  return Promise.all(this.map(fn));
};

Array.prototype.filterAsync = function(fn) {
  return this.mapAsync(fn).then(_arr => this.filter((v, i) => !!_arr[i]));
};
```

However, now we get to the problem of chaining. How do you chain together `mapAsync()` and `filterAsync()`? You could use `.then()` but that would not be quite as neat. Instead, you can create an `AsyncArray` class that represents a Promise that will eventually return an array, and attach `mapAsync`, `filterAsync`, and `forEachAsync` as class methods:

```
class AsyncArray {
  constructor(promise) {
    this.$promise = promise || Promise.resolve();
  }

  then(resolve, reject) {
    return new AsyncArray(this.$promise.then(resolve, reject));
  }

  catch(reject) {
    return this.then(null, reject);
  }

  mapAsync(fn) {
    return this.then(arr => Promise.all(arr.map(fn)));
  }

  filterAsync(fn) {
    return new AsyncArray(Promise.all([this, this.mapAsync(fn)]).then(([arr, _arr]) => arr.filter((v, i) => !!_arr[i])));
  }

  forEachAsync(fn) {
    return this.then(arr => arr.reduce((promise, n) => promise.then(() => fn(n)), Promise.resolve()));
  }
}
```

With this `AsyncArray` class, you chain `mapAsync()`, `filterAsync()`, and `forEachAsync()` since each of these helper methods returns an `AsyncArray`. Here's how it looks with the previous MongoDB example:

```
async function copy(ids, db1, db2) {
  new AsyncArray(Promise.resolve(ids)).
    mapAsync(function(_id) {
      return db1.collection('Test').findOne({ _id });
    }).
    filterAsync(async function(doc) {
      const _doc = await db2.collection('Test').findOne({ _id: doc._id });
      return !_doc;
    }).
    forEachAsync(async function(doc) {
      console.log('Insert', doc);
      await db2.collection('Test').insertOne(doc);
    }).
    catch(error => console.error(error));
}

async function test() {
  const db1 = await MongoClient.connect('mongodb://localhost:27017/db1');
  const db2 = await MongoClient.connect('mongodb://localhost:27017/db2');
  await db1.dropDatabase();
  await db2.dropDatabase();

  const docs = [
    { _id: 1 },
    { _id: 2 },
    { _id: 3 },
    { _id: 4 }
  ];

  await db1.collection('Test').insertMany(docs);
  // Only insert docs with _id 2 and 4 into db2
  await db2.collection('Test').insertMany(docs.filter(doc => doc._id % 2 === 0));

  await copy(docs.map(doc => doc._id), db1, db2);
}

test();
```

Wrapping Up with `reduce()`

Now that you have `mapAsync()`, `filterAsync()`, and `forEachAsync()`, why not go all the way and implement `reduceAsync()` too?

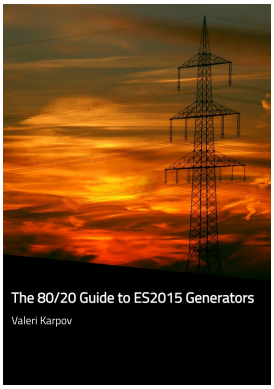
```
  reduceAsync(fn, initial) {
    return Promise.resolve(initial).then(cur => {
      return this.forEachAsync(async function(v, i) {
        cur = await fn(cur, v, i);
      }).then(() => cur);
    });
  }
```

Here's how to use `reduceAsync()`:

```
async function test() {
  const db = await MongoClient.connect('mongodb://localhost:27017/test');
  await db.dropDatabase();

  const docs = [
    { _id: 1, name: 'Axl' },
    { _id: 2, name: 'Slash' },
    { _id: 3, name: 'Duff' },
    { _id: 4, name: 'Izzy' },
    { _id: 5, name: 'Adler' }
  ];

  await db.collection('People').insertMany(docs);

  const ids = docs.map(doc => doc._id);

  const nameToId = await new AsyncArray(Promise.resolve(ids)).
    reduceAsync(async function (cur, _id) {
      const doc = await db.collection('People').findOne({ _id });
      cur[doc.name] = doc._id;
      return cur;
    }, {});
  console.log(nameToId);
}

test();
```

Overall, using async functions with `map()`, `filter()`, `reduce()`, and `forEach()` is possible, but requires custom functions and sophisticated promise chaining. I'm sure somebody will come out with a library that makes working with promise arrays seamless, and I look forward to seeing it. Functional programming primitives make synchronous array operations clean and elegant, and remove a lot of wasteful intermediary values via chaining. Adding helpers that can operate on promises that resolve to arrays opens up some exciting possibilities.

Async/await is powerful, but if you're stuck using Node.js 4.x or 6.x because of LTS (especially since Node.js 8 is delayed (https://github.com/nodejs/CTC/issues/99)), you can still use similar functional programming patterns using ES6 generators and co (http://npmjs.org/package/co). If you're looking for a much deeper dive into co, including how to write your own co replacement from scratch, check out my ebook, The 80/20 Guide to ES2015 Generators (http://es2015generators.com/)



(http://es2015generators.com)

Found a typo or error? Open up a pull request! This post is available as markdown on Github (https://github.com/vkarpov15/thecodebarbarian.com/blob/master/lib/posts/20170420_async_fp.md)