Most popular (last 30 days)

ECMAScript 2017: the final feature set
ECMAScript 6 modules: the final syntax
ES proposal: import() – dynamically importing ES modules
Making transpiled ES modules more spec-compliant
ES proposal: Shared memory and atomics
Classes in ECMAScript 6 (final semantics)
Communicating between Web Workers via MessageChannel

Most popular (all time)

ECMAScript 6 modules: the final syntax
Classes in ECMAScript 6 (final semantics)
Iterating over arrays and objects in JavaScript
ECMAScript 6's new array methods
The final feature set of ECMAScript 2016 (ES7)
WebAssembly: a binary format for the web
Basic JavaScript for the impatient programmer
Google Dart to "ultimately ... replace JavaScript"
Six nifty ES6 tricks
Google's Polymer and the future of web UI frameworks

Blog archive

► 2017 (4)
► 2016 (38)

2015-03-04

No promises: asynchronous JavaScript with only generators

Labels: dev, esnext, javascript

Two ECMAScript 6 [1] features enable an intriguing new style of asynchronous JavaScript code: promises [2] and generators [3]. This blog post explains this new style and presents a way of using it without promises.

1. Overview

Normally, you make a function calls like this:

```
let result = func(···);
console.log(result);
```

It would be great if this style of invocation also worked for functions that perform tasks (such as downloading a file) asynchronously. For that to work, execution of the previous code would have to pause until `func()` returns with a result.

Before ECMAScript 6, you couldn't pause and resume the execution of code, but you could simulate it, by putting `console.log(result)` into a callback, a so-called *continuation* [4]. The continuation is triggered by `asyncFunc()`, once it is done:

```
asyncFunc('http://example.com', result => {
    console.log(result);
});
```

Promises [2] are basically a smarter way of managing callbacks:

```
asyncFunc('http://example.com')
.then(result => {
    console.log(result);
});
```

In ECMAScript 6, you can use generator functions [3], which can be paused and resumed. With a library such as Q, a generator-based solution looks almost like our ideal code:

```
Q.spawn(function* () {
    let result = yield asyncFunc('http://example.com');
    console.log(result);
});
```

Alas, `asyncFunc()` needs to be implemented using promises:

```
function asyncFunc(url) {
    return new Promise((resolve, reject) => {
        otherAsyncFunc(url,
            result => resolve(result));
    });
}
```

Dr. Axel Rauschmayer

▼ 2015 (65)

- ► December (7)
- ► November (6)
- ► October (13)
- ► September (5)
- ► August (6)
- ► July (3)
- ► June (3)
- ► April (4)
- ▼ March (4)
  - ES6 generators in depth
  - Using the ES6 transpiler Babel on Node.js
  - The destructuring algorithm in ECMAScript 6
  - No promises: asynchronous JavaScript with only gen...
- ► February (8)
- ► January (6)

► 2014 (55)

► 2013 (98)

► 2012 (177)

► 2011 (380)

► 2010 (174)

► 2009 (68)

► 2008 (46)

► 2007 (12)

► 2006 (1)

► 2005 (2)

Labels

dev (592)

javascript (400)

computers (316)

life (194)

jslang (179)

esnext (156)

apple (107)

webdev (95)

mobile (83)

scitech (50)

hack (49)

mac (47)

google (39)

java (37)

ios (33)

business (32)

video (32)

clientjs (31)

hci (27)

However, with a small library shown later, you can run the initial code like with `Q.spawn()`, but implement `asyncFunc()` like this:

```
function* asyncFunc(url) {
    const caller = yield; // (A)
    otherAsyncFunc(url,
        result => caller.success(result));
}
```

Line A is how the library provides `asyncFunc()` with callbacks. The advantage compared to the previous code is that this function is again a generator and can make other asynchronous calls via `yield`.

2. Code

I'll first show two examples, before I present the code of the library.

2.1. Example 1: `echo()`

`echo()` is an asynchronous function, implemented via a generator:

```
function* echo(text, delay = 0) {
    const caller = yield;
    setTimeout(() => caller.success(text), delay);
}
```

In the following code, `echo()` is used three time, sequentially:

```
run(function* echoes() {
    console.log(yield echo('this'));
    console.log(yield echo('is'));
    console.log(yield echo('a test'));
});
```

The parallel version of this code looks as follows.

```
run(function* parallelEchoes() {
    let startTime = Date.now();
    let texts = yield [
        echo('this', 1000),
        echo('is', 900),
        echo('a test', 800)
    ];
    console.log(texts); // ['this', 'is', 'a test']
    console.log('Time: '+(Date.now()-startTime));
});
```

As you can see, the library performs the asynchronous calls in parallel if you yield an array of generator invocations.

This code takes about 1000 milliseconds.

2.2. Example 2: `httpGet()`

The following code demonstrates how you can implement a function that gets a file via `XMLHttpRequest`:

```
function* httpGet(url) {
    const caller = yield;

    var request = new XMLHttpRequest();
    request.onreadystatechange = function () {
        if (this.status === 200) {
            caller.success(this.response);
        } else {
            // Something went wrong (404 etc.)
            caller.failure(new Error(this.statusText));
        }
    }
    request.onerror = function () {
```

Tweets by @rauschma

Axel Rauschm… @rauschma

Enjoying "Troll Hunters":
– "Let's call him 'Gnome Chomsky'"
– "Juliet dies in this? Nooo!"

9h

Axel Rauschmayer Retweeted

Jonathan Creamer @jcreamer898

A nice little shout out to @rauschma... infoworld.com/article/316483 … #es2017 #async

JavaScr…
ECMASc…
infoworld…

13h

Axel Rauschmayer @rauschma

Not a physics book! twitter.com/ManningBooks/s …

02 Feb

Axel Rauschmayer Retweeted

Jordan Harband @ljharb

Making our React components forbid extra props has caught SO many bugs. I highly recommend it.

npmjs.com/airbnb-prop-ty…

npm: air…
Custom …
npmjs.com

02 Feb

Axel Rauschmayer Retweeted

Seth Petry-Johnson @spetryjohnson

Open strong. Be bold. Tell a story. Do something to get me interested. Opening w/ the "obligatory 'about me' slide" puts me to sleep :)

02 Feb

Google

entertainment (26)
nodejs (26)
society (26)
browser (25)
firefox (25)
html5 (24)
ipad (24)
movie (23)
psychology (22)
2ality (18)
tv (18)
android (17)
social (17)
chrome (16)
fun (16)
jsmodules (16)
tablet (16)
humor (15)
politics (15)
web (15)
cloud (14)
hardware (14)
microsoft (14)
software engineering (13)
blogging (12)
gaming (12)
eclipse (11)
gwt (11)
numbers (11)
programming languages (11)
app store (10)
media (10)
nature (10)
security (10)
semantic web (10)
software (10)
twitter (10)
webos (10)
12quirks (9)
education (9)
jstools (9)
photo (9)
webcomponents (9)
windows 8 (9)
async (8)
idea (8)
iphone (8)
itunes (8)
scifi-fantasy (8)
app (7)
babel (7)
bookmarklet (7)
chromeos (7)

```
        caller.failure(new Error(
            'XMLHttpRequest Error: '+this.statusText));
    };
    request.open('GET', url);
    request.send();
}
```

Let’s use `httpGet()` sequentially:

```
run(function* downloads() {
    let text1 = yield httpGet('https://localhost:8000/file1.html');
    let text2 = yield httpGet('https://localhost:8000/file2.html');
    console.log(text1, text2);
});
```

Using `httpGet()` in parallel looks like this:

```
run(function* parallelDownloads() {
    let [text1,text2] = yield [
        httpGet('https://localhost:8000/file1.html'),
        httpGet('https://localhost:8000/file2.html')
    ];
    console.log(text1, text2);
});
```

2.3. The library

The library profits from the fact that calling a generator function does not execute its body, but returns a generator object.

```
/**
 * Run the generator object `genObj`,
 * report results via the callbacks in `callbacks`.
 */
function runGenObj(genObj, callbacks = undefined) {
    handleOneNext();

    /**
     * Handle one invocation of `next()`:
     * If there was a `prevResult`, it becomes the parameter.
     * What `next()` returns is what we have to run next.
     * The `success` callback triggers another round,
     * with the result assigned to `prevResult`.
     */
    function handleOneNext(prevResult = null) {
        try {
            let yielded = genObj.next(prevResult); // may throw
            if (yielded.done) {
                if (yielded.value !== undefined) {
                    // Something was explicitly returned:
                    // Report the value as a result to the caller
                    callbacks.success(yielded.value);
                }
            } else {
                setTimeout(runYieldedValue, 0, yielded.value);
            }
        }
        // Catch unforeseen errors in genObj
        catch (error) {
            if (callbacks) {
                callbacks.failure(error);
            } else {
                throw error;
            }
        }
    }
    function runYieldedValue(yieldedValue) {
        if (yieldedValue === undefined) {
```

english (7)
es proposal (7)
fringe (7)
html (7)
jsint (7)
jsshell (7)
thunderbolt (7)
webapp (7)
advancedjs (6)
blogger (6)
crowdsourcing (6)
latex (6)
lion (6)
promises (6)
ted (6)
book (5)
environment (5)
gadget (5)
googleio (5)
intel (5)
jsarrays (5)
jshistory (5)
layout (5)
light peak (5)
michael j. fox (5)
music (5)
pdf (5)
polymer (5)
shell (5)
tc39 (5)
template literals (5)
underscorejs (5)
vlc (5)
__proto__ (4)
coffeescript (4)
concurrency (4)
dart (4)
facebook (4)
gimp (4)
googleplus (4)
health (4)
howto (4)
hp (4)
javafx (4)
kindle (4)
leopard (4)
macbook (4)
motorola (4)
münchen (4)
occupy (4)
pl fundamentals (4)
presenting (4)
publishing (4)
series (4)

```
                // If code yields `undefined`, it wants callbacks
                handleOneNext(callbacks);
            } else if (Array.isArray(yieldedValue)) {
                runInParallel(yieldedValue);
            } else {
                // Yielded value is a generator object
                runGenObj(yieldedValue, {
                    success(result) {
                        handleOneNext(result);
                    },
                    failure(err) {
                        genObj.throw(err);
                    },
                });
            }
        }

        function runInParallel(genObjs) {
            let resultArray = new Array(genObjs.length);
            let resultCountdown = genObjs.length;
            for (let [i,genObj] of genObjs.entries()) {
                runGenObj(genObj, {
                    success(result) {
                        resultArray[i] = result;
                        resultCountdown--;
                        if (resultCountdown <= 0) {
                            handleOneNext(resultArray);
                        }
                    },
                    failure(err) {
                        genObj.throw(err);
                    },
                });
            }
        }
    }

    function run(genFunc) {
        runGenObj(genFunc());
    }
```

Note that you only need use `caller = yield` and `caller.success(···)` in asynchronous functions that use callbacks. If an asynchronous function only calls other asynchronous functions (via `yield`) then you can simply explicitly `return` a value.

One important feature is missing: support for calling async functions implemented via promises. It would be easy to add, though – by adding another case to `runYieldedValue()`.

3. Conclusion: asynchronous JavaScript via coroutines

Couroutines [5] are a single-threaded version of multi-tasking: Each coroutine is a thread, but all coroutines run in a single thread and they explicitly relinquish control via `yield`. Due to the explicit yielding, this kind of multi-tasking is also called *cooperative* (versus the usual *preemptive multi-tasking*).

Generators are *shallow* co-routines [6]: their execution state is only preserved *within* the generator function: It doesn't extend further backwards than that and recursively called functions can't yield.

The code for asynchronous JavaScript without promises that you have seen in this blog post is purely a proof of concept. It is completely unoptimized and may have other flaws preventing it from being used in practice.

But coroutines seem like the right mental model when thinking about asynchronous computation in JavaScript. They could be an interesting avenue to explore for ECMAScript 2016 (ES7) or later. As we have seen, not much would need to be added to generators to make this work:

- `caller = yield` is a kludge.

textbook (4)
web design (4)
amazon (3)
asmjs (3)
back to the future (3)
bitwise_ops (3)
css (3)
es2016 (3)
flattr (3)
fluentconf (3)
food (3)
foreign languages (3)
house (3)
icloud (3)
info mgmt (3)
jsfuture (3)
jsstyle (3)
linux (3)
mozilla (3)
python (3)
regexp (3)
samsung (3)
tizen (3)
traffic (3)
typedjs (3)
unix (3)
adobe (2)
angry birds (2)
angularjs (2)
astronomy (2)
audio (2)
comic (2)
design (2)
dom (2)
ecommerce (2)
eval (2)
exploring es6 (2)
facebook flow (2)
facets (2)
flash (2)
free (2)
futurama (2)
guide (2)
history (2)
hyena (2)
internet explorer (2)
iteration (2)
journalism (2)
jquery (2)
jsengine (2)
jslib (2)
law (2)
lightning (2)
markdown (2)

- Similarly, having to report results and errors via callbacks is unfortunate. It'd be nice if `return` and `throw` could always be used, but they don't work inside callbacks.

3.1. What about streams?

When it comes to asynchronous computation, there are two fundamentally different needs:

1. The results of a single computation: One popular way of performing those are promises.
2. A series of results: Asynchronous Generators [7] have been proposed for ECMAScript 2016 for this use case.

For #1, coroutines are an interesting alternative. For #2, David Nolen has suggested [8] that CSP (Communicating Sequential Processes) work well. For binary data, WHATWG is working on Streams [9].

3.2. Current practical solutions

All current practical solutions are based on Promises:

- Q is a promise library and polyfill that includes the aforementioned `Q.spawn()`, which is based on promises.
- co brings just the `spawn()` functionality and relies on an external Promise implementation. It is therefore a good fit for environments such as Babel that already have Promises.
- Babel has a first implementation of async functions (as proposed for ECMAScript 2016). Under the hood, they are translated to code that is similar to `spawn()` and based on Promises. However, if you use this feature, you are leaving standard territory and your code won't be portable to other ES6 environments. Async functions may still change considerably before they are standardized.

4. Further reading

1. "Exploring ES6: Upgrade to the next version of JavaScript", book by Axel
2. ECMAScript 6 promises (2/2): the API
3. Iterators and generators in ECMAScript 6
4. Asynchronous programming and continuation-passing style in JavaScript
5. "Coroutine" on Wikipedia
6. "Why coroutines won't work on the web" by David Herman
7. "Async Generator Proposal" by Jafar Husain
8. "ES6 Generators Deliver Go Style Concurrency" by David Nolen
9. "Streams: Living Standard", edited by Domenic Denicola and Takeshi Yoshino

22 Comments   The 2ality blog   1 Login

Recommend 4   Share   Sort by Best

Join the discussion…

trusktr • a year ago

You might love to know about `async-csp`, which I think is amazing, and uses async/await instead of generators: https://www.npmjs.com/async-cs...

3 • Reply • Share ›

Serge Zarouski • 2 years ago

If anyone is interested I wrote an article about how co is working internally - http://webuniverse.io/asynchro....

2 • Reply • Share ›

Alex Mills • a month ago

did you publish this library?

math (2)
meego (2)
month (2)
nike (2)
nokia (2)
npm (2)
programming (2)
raffle (2)
repl (2)
servo (2)
sponsor (2)
steve jobs (2)
travel (2)
typescript (2)
usb (2)
winphone (2)
wwdc (2)
airbender (1)
amdefine (1)
aol (1)
app urls (1)
architecture (1)
atscript (1)
basic income (1)
biology (1)
blink (1)
bluetooth (1)
canada (1)
clip (1)
coding (1)
community (1)
cross-platform (1)
deutsch (1)
diaspora (1)
distributed-social-network (1)
dsl (1)
dvd (1)
dzone (1)
emacs (1)
emberjs (1)
energy (1)
esnext news (1)
esprop (1)
example (1)
facetator (1)
feedback (1)
firefly (1)
firefoxos (1)
fritzbox (1)
german (1)
git (1)
guest (1)
guice (1)
h.264 (1)

• Reply • Share ›

Axel Rauschmayer Mod ➦ Alex Mills • a month ago

I thought about doing so, but Promises are getting more optimized all the time. Thus, in the long run, there is no point.

• Reply • Share ›

Alex Mills ➦ Axel Rauschmayer • a month ago

@Axel Rauschmayer one thing you *could* do. Make this work with Observables. Observables are a nice and powerful alternative to promises, and it would be nice if they could be paired with generators as well.

• Reply • Share ›

Alex Mills ➦ Axel Rauschmayer • a month ago

I would definitely experiment with your idea here if you published a usable version! One thing is that I still don't quite understand what "const caller = yield" is doing. I am guessing it ends up storing a reference to a callback.

• Reply • Share ›

Alex Mills ➦ Axel Rauschmayer • a month ago

yeah, the downside of your idea here appears to be that all functions/methods involved need to be generators. With promises+generators, that is not the case. Right? One thing I do like about your idea here, is the ease of doing parallel operations. Not sure if it's quite as easy to do parallel opts with promises+generators.

• Reply • Share ›

Joshua Gough • 2 years ago

Thanks for the very interesting article.

• Reply • Share ›

dark_ruby • 2 years ago

is it possible to write generator functions also using arrow notation? something like *=> ?

• Reply • Share ›

Tom Hodbod ➦ dark_ruby • 2 years ago

nope

1 • Reply • Share ›

trusktr ➦ dark_ruby • a year ago

Not yet, but it's been discussed on at https://esdiscuss.org/topic/ge..., where some people suggested things like *=>.

• Reply • Share ›

niloy_mondal • 2 years ago

Great post.

• Reply • Share ›

bradleymeck • 2 years ago

I wrote something similar (generator-runner), you might want to mention why locking may be important; so `caller` cannot be called multiple times in the future to step the generator inappropriately. Also would be nice to see a blurb about using generator.return for abort / try{}finally{}.

• Reply • Share ›

Guest • 2 years ago

Thanks for this very interesting post, as always !

home entertainment (1)
hosting (1)
htc (1)
ical (1)
jsdom (1)
jsmyth (1)
library (1)
location (1)
marketing (1)
mars (1)
meta-data (1)
middle east (1)
mpaa (1)
msl (1)
mssurface (1)
netflix (1)
nsa (1)
obama (1)
openoffice (1)
opinion (1)
oracle (1)
organizing (1)
philosophy (1)
pixar (1)
pnacl (1)
prism (1)
privacy (1)
proxies (1)
puzzle (1)
raspberry pi (1)
read (1)
rodney (1)
rust (1)
safari (1)
sponsoring (1)
star trek (1)
static generation (1)
talk (1)
technique (1)
theora (1)
thunderbird (1)
typography (1)
unicode (1)
v8 (1)
voice control (1)
webassembly (1)
webkit (1)
webm (1)
webpack (1)
yahoo (1)

You were talking about the "how" of using generators without promises and abit less about the "why" of doing just that.
I was using Promises(Bluebird) with Co.js for some pretty complicated server side async code - and everything worked smoothly.
The async code had a double advantage:
1) normal js syntax and language constructs - like for-loops, try-catch, semi colons instead of "then" or callbacks (thanks to co.js)
2) traditional functional programming functions (map, fold and friends) in the async world (thanks to Bluebird).

Also, since co.js 4.0 and up returns a promise from the co function - you can basically use yield recuresevly inside inner functions - just warp in another co call. That's very important for functional async programming.

So I was wondering - Do you think there are any important/vital drawbacks in the generatores + promises approach? Thanks.

Reply • Share ›

Florent • 2 years ago
What is the point of defaulting arguments (here callbacks) to undefined ?

function runGenObj(genObj, callbacks = undefined) {

Florent
Reply • Share ›

Axel Rauschmayer Mod → Florent • 2 years ago
It's a visual clue that the parameter is optional.
1 Reply • Share ›

Alex Mills → Axel Rauschmayer • a month ago
yeah that's what I figured, maybe default it to null instead
Reply • Share ›

Randy Creasi → Florent • 2 years ago
Good question. The only possible rationale I can imagine is that it's a form of documentation, implying "this argument is optional and it's ok if it's undefined."

EDIT: According to this, it also changes the .length property of the function:
http://tc39wiki.calculist.org/...
1 Reply • Share ›

Serkan Sipahi • 2 years ago
have you tested your code? I have tested it on traceur(https://google.github.io/trace...) , babel(https://babeljs.io/repl/#?expe...) and with iojs(for iojs i have make a rewrite because it doesnt support let, default parameters, Destructuring Assignment and arrow functions). Your example doesnt work.
Reply • Share ›

Axel Rauschmayer Mod → Serkan Sipahi • 2 years ago
The blog had a bug due to a last minute edit. Fixed now.
Reply • Share ›

~flow • 2 years ago
Very interesting! You may want to have a look at https://github.com/loveencount... which is a small library to simplify setting up 'synchronous' functions that use `yield` to call asynchronous functions. Examples can be found at https://github.com/loveencount....
Reply • Share ›

Rafał Krupiński • 2 years ago

I keep wondering why does prevResult default to null instead of undefined. This null could be basically anything since this value cannot be accessed. Oh, and it breaks Traceur :/

• Reply • Share ›

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)