Home
Main Content

By Kyle Simpson on July 21, 2014

ES6 Generators: Complete Series

1. *The Basics Of ES6 Generators*
2. *Diving Deeper With ES6 Generators*
3. *Going Async With ES6 Generators*
4. *Getting Concurrent With ES6 Generators*

One of the most exciting new features coming in JavaScript ES6 is a new breed of function, called a **generator**. The name is a little strange, but the behavior may seem *a lot stranger* at first glance. This article aims to explain the basics of how they work, and build you up to understanding why they are so powerful for the future of JS.

Run-To-Completion

The first thing to observe as we talk about generators is how they differ from normal functions with respect to the "run to completion" expectation.

Whether you realized it or not, you've always been able to assume something fairly fundamental about your functions: once the function starts running, it will always run to completion before any other JS code can run.

Example:

```
setTimeout(function(){
    console.log("Hello World");
},1);
```

```
function foo() {
    // NOTE: don't ever do crazy long-running loops like this
    for (var i=0; i<=1E10; i++) {
        console.log(i);
    }
}

foo();
// 0..1E10
// "Hello World"
```

Here, the `for` loop will take a fairly long time to complete, well more than one millisecond, but our timer callback with the `console.log(..)` statement cannot interrupt the `foo()` function while it's running, so it gets stuck at the back of the line (on the event-loop) and it patiently waits its turn.

What if `foo()` could be interrupted, though? Wouldn't that cause havoc in our programs?

That's exactly the ~~nightmares~~ challenges of multi-threaded programming, but we are quite fortunate in JavaScript land to not have to worry about such things, because JS is always single-threaded (only one command/function executing at any given time).

Note: Web Workers are a mechanism where you can spin up a whole separate thread for a part of a JS program to run in, totally in parallel to your main JS program thread. The reason this doesn't introduce multi-threaded complications into our programs is that the two threads can only communicate with each other through normal async events, which always abide by the event-loop *one-at-a-time* behavior required by run-to-completion.

Run..Stop..Run

With ES6 generators, we have a different kind of function, which may be *paused* in the middle, one or many times, and resumed *later*, allowing other code to run during these paused periods.

If you've ever read anything about concurrency or threaded programming, you may have seen the term "cooperative", which basically indicates that a process (in our case, a function) itself chooses when it will allow an interruption, so that it can **cooperate** with other code. This concept is contrasted with "preemptive", which suggests that a process/function could be interrupted against its will.

ES6 generator functions are "cooperative" in their concurrency behavior. Inside the generator function body, you use the new `yield` keyword to pause the function from inside itself. Nothing can pause a generator from the outside; it pauses itself when it comes across a `yield`.

However, once a generator has `yield`-paused itself, it cannot resume on its own. An external control must be used to restart the generator. We'll explain how that happens in just a moment.

So, basically, a generator function can stop and be restarted, as many times as you choose. In fact, you can specify a generator function with an infinite loop (like the infamous `while (true) { .. }`) that essentially never finishes. While that's usually madness or a mistake in a normal JS program, with generator functions it's perfectly sane and sometimes exactly what you want to do!

Even more importantly, this stopping and starting is not *just* a control on the execution of the generator function, but it also enables 2-way message passing into and out of the generator, as it progresses. With normal functions, you get parameters at the beginning and a `return` value at the end. With generator functions, you send messages out with each `yield`, and you send messages back in with each restart.

Syntax Please!

Let's dig into the syntax of these new and exciting generator functions.

First, the new declaration syntax:

```
function *foo() {
    // ..
}
```

Notice the `*` there? That's new and a bit strange looking. To those from some other languages, it may look an awful lot like a function return-value pointer. But don't get confused! This is just a way to signal the special generator function type.

You've probably seen other articles/documentation which use `function* foo(){ }` instead of `function *foo(){ }` (difference in placement of the `*`). Both are valid, but I've recently decided that I think `function *foo() { }` is more accurate, so that's what I'm using here.

Now, let's talk about the contents of our generator functions. Generator functions are just normal JS functions in most respects. There's very little new syntax to learn *inside* the generator function.

The main new toy we have to play with, as mentioned above, is the `yield` keyword. `yield ___` is called a "yield expression" (and not a statement) because when we restart the generator, we will send a value back in, and whatever we send in will be the computed result of that `yield ___` expression.

Example:

```
function *foo() {
    var x = 1 + (yield "foo");
    console.log(x);
}
```

The `yield "foo"` expression will send the `"foo"` string value out when pausing the generator function at that point, and whenever (if ever) the generator is restarted, whatever value is sent in will be the result of that expression, which will then get added to `1` and assigned to the `x` variable.

See the 2-way communication? You send the value `"foo"` out, pause yourself, and at some point *later* (could be immediately, could be a long time from now!), the generator will be restarted and will give you a value back. It's almost as if the `yield` keyword is sort of making a request for a value.

In any expression location, you *can* just use `yield` by itself in the expression/statement, and there's an assumed `undefined` value `yield` ed out. So:

```
// note: `foo(..)` here is NOT a generator!!
function foo(x) {
    console.log("x: " + x);
}

function *bar() {
    yield; // just pause
    foo( yield ); // pause waiting for a parameter to pass into `foo(..)`
}
```

Generator Iterator

"Generator Iterator". Quite a mouthful, huh?

Iterators are a special kind of behavior, a design pattern actually, where we step through an ordered set of values one at a time by calling `next()`. Imagine for example using an iterator on an array that has five values in it: `[1,2,3,4,5]`. The first `next()` call would return `1`, the second `next()` call would return `2`, and so on. After all values had been returned, `next()` would return `null` or `false` or otherwise signal to you that you've iterated over all the values in the data container.

The way we control generator functions from the outside is to construct and interact with a *generator iterator*. That sounds a lot more complicated than it really is. Consider this silly example:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
}
```

To step through the values of that `*foo()` generator function, we need an iterator to be constructed. How do we do that? Easy!

```
var it = foo();
```

Oh! So, calling the generator function in the normal way doesn't actually execute any of its contents.

That's a little strange to wrap your head around. You also may be tempted to wonder, why isn't it `var it = new foo()`. Shrugs. The whys behind the syntax are complicated and beyond our scope of discussion here.

So now, to start iterating on our generator function, we just do:

```
var message = it.next();
```

That will give us back our `1` from the `yield 1` statment, but that's not the only thing we get back.

```
console.log(message); // { value:1, done:false }
```

We actually get back an object from each `next()` call, which has a `value` property for the `yield` ed-out value, and `done` is a boolean that indicates if the generator function has fully completed or not.

Let's keep going with our iteration:

```
console.log( it.next() ); // { value:2, done:false }
console.log( it.next() ); // { value:3, done:false }
console.log( it.next() ); // { value:4, done:false }
console.log( it.next() ); // { value:5, done:false }
```

Interesting to note, `done` is still `false` when we get the value of `5` out. That's because *technically*, the generator function is not complete. We still have to call a final `next()` call, and if we send in a value, it has to be set as the result of that `yield 5` expression. Only **then** is the generator function complete.

So, now:

```
console.log( it.next() ); // { value:undefined, done:true }
```

So, the final result of our generator function was that we completed the function, but there was no result given (since we'd already exhausted all the `yield ___` statements).

You may wonder at this point, can I use `return` from a generator function, and if I do, does that value get sent out in the `value` property?

Yes...

```
function *foo() {
    yield 1;
    return 2;
}

var it = foo();

console.log( it.next() ); // { value:1, done:false }
console.log( it.next() ); // { value:2, done:true }
```

... and no.

It may not be a good idea to rely on the `return` value from generators, because when iterating generator functions with `for..of` loops (see below), the final `return` ed value would be thrown away.

For completeness sake, let's also take a look at sending messages both into and out of a generator function as we iterate it:

```
function *foo(x) {
    var y = 2 * (yield (x + 1));
    var z = yield (y / 3);
    return (x + y + z);
}

var it = foo( 5 );

// note: not sending anything into `next()` here
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

You can see that we can still pass in parameters (`x` in our example) with the initial `foo( 5 )` iterator-instantiation call, just like with normal functions, making `x` be value `5` .

The first `next(..)` call, we don't send in anything. Why? Because there's no `yield` expression to receive what we pass in.

But if we *did* pass in a value to that first `next(..)` call, nothing bad would happen. It would just be a tossed-away value. ES6 says for generator functions to ignore the unused value in this case. (**Note:** At the time of writing, nightlies of both Chrome and FF are fine, but other browsers may not yet be fully compliant and may incorrectly throw an error in this case).

The `yield (x + 1)` is what sends out value `6`. The second `next(12)` call sends `12` to that waiting `yield (x + 1)` expression, so `y` is set to `12 * 2`, value `24`. Then the subsequent `yield (y / 3)` (`yield (24 / 3)`) is what sends out the value `8`. The third `next(13)` call sends `13` to that waiting `yield (y / 3)` expression, making `z` set to `13`.

Finally, `return (x + y + z)` is `return (5 + 24 + 13)`, or `42` being returned out as the last `value`.

Re-read that a few times. It's weird for most, the first several times they see it.

`for..of`

ES6 also embraces this iterator pattern at the syntactic level, by providing direct support for running iterators to completion: the `for..of` loop.

Example:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
    return 6;
}

for (var v of foo()) {
    console.log( v );
}
// 1 2 3 4 5

console.log( v ); // still `5`, not `6` :(
```

As you can see, the iterator created by `foo()` is automatically captured by the `for..of` loop, and it's automatically iterated for you, one iteration for each value, until a `done:true` comes out. As long as `done` is `false`, it automatically extracts the `value` property and assigns it to your iteration variable (`v` in our case). Once `done` is `true`, the loop iteration stops (and does nothing with any final `value` returned, if any).

As noted above, you can see that the `for..of` loop ignores and throws away the `return 6` value. Also, since there's no exposed `next()` call, the `for..of` loop cannot be used in situations where you need to pass in values to the generator steps as we did above.

Summary

OK, so that's it for the basics of generators. Don't worry if it's a little mind-bending still. All of us have felt that way at first!

It's natural to wonder what this new exotic toy is going to do practically for your code. There's a **lot** more to them, though. We've just scratched the surface. So we have to dive deeper before we can discover just how powerful they can/will be.

After you've played around with the above code snippets (try Chrome nightly/canary or FF nightly, or node 0.11+ with the `--harmony` flag), the following questions may arise:

1. How does error handling work?
2. Can one generator call another generator?
3. How does async coding work with generators?

Those questions, and more, will be covered in subsequent articles here, so stay tuned!

Sooner or later you'll run across a regular expression. With their cryptic syntax, confusing documentation and massive learning curve, most developers settle for copying and pasting them from StackOverflow and hoping they work. But what if you could decode regular expressions and harness their power? In...

An Interview with Eric Meyer

Your early CSS books were instrumental in pushing my love for front end technologies. What was it about CSS that you fell in love with and drove you to write about it? At first blush, it was the simplicity of it as compared to the table-and-spacer...

Incredible Demos

Fancy Navigation with MooTools JavaScript

Navigation menus are traditionally boring, right? Most of the time the navigation menu consists of some imagery with a corresponding mouseover image. Where's the originality? I've created a fancy navigation menu that highlights navigation items and creates a chain effect. The XHTML Just some simple...

Drag & Drop Elements to the Trash with MooTools 1.2

Everyone loves dragging garbage files from their desktop into their trash can. There's a certain amount of irony in doing something on your computer that you also do in real life. It's also a quick way to get rid of things. That's...

Discussion

Bhanu Pratap Chaudhary

Nice Article. Cleared up `for..of loop` for me. I also wrote an article on JS generators which can easily qualify as part 2 of this article http://goo.gl/Ve2Krb. Would love to know your take on my article.

Kyle Simpson

Thanks Bhanu. Nice article you've written. As for this series here, this is part 1 in a 4-part series that will be published over the next few weeks here. I've already written all the posts, we're just spacing them out to give people a chance to digest. Don't worry, there's plenty of coverage of advanced topics in the rest of the series, including `yield*`, `throw(..)`, async generators (generators+promises), and even CSP-style co-routines.

Stay tuned! :)

Lino Quickels

Really enjoyed this article! Thanks a lot! :)

zhiyelee

> The second next(12) call sends 12 to the first yield (x + 1) expression. The third next(13) call sends 13 to the second yield (y / 3) expression.

Not very clear here.
The second next(12) resume the generator as if the previous yield expression(yield (x + 1)) return 12

Kyle Simpson

You've correctly interpreted it. I suppose the wording is a tad clumsy, but then again, so is the concept itself, at first. Thanks for the feedback!

Ralph Whitbeck

I just spent like a half an hour on this because I couldn't figure out how 8 is returned. Plus you told us to "Re-read that a few times." as I was hoping it would click. Thanks Zhiyelee for the comment it helped me out.

Chetan Dhembre

```
function* foo(x) {
    var y = 2 * (yield (x + 1));
    var z = yield (y / 3);
    return (x + y + z);
}

var it = foo( 5 );

// note: not sending anything into `next()` here
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

in above example you mention

"The first `next(..)` call, we don't send in anything. Why? Because there's no yield expression to receive what we pass in."

I am confuses by this statement. What is relation between number of `next()` function i can call on generator function and number `yield` generator function have in its definition?

Kyle Simpson

Great question!

1. Not all generators have to be fully exhausted, so there's no requirement that `next(..)` calls "match" `yield` statements 1-for-1.

2. But, if you are going to exhaust a generator (that can be exhausted — no infinite loop) you will need one more `next(..)` call than the number of `yield` expressions. In the example you cited, there's 2 `yield` s and 3 `next(..)` calls.

Chetan Dhembre

Thank for such quick reply... what is use to first `next()` call (actually i do not understand by "exhaust generator" and it's relation to first `next()` call).. as next two `next()` call actually related to `yield` in function definition.

Kyle Simpson

@Chetan:

The generator starts out paused when you call `foo(5)` (that is, nothing has run inside it yet). Why? Because you need `foo(5)` to give you the iterator, but if any of the contents also ran, whenever the first `yield __` expression was sent out, you'd lose that value (because you only got the iterator `it` in the assignment).

So, the first `next()` `unpauses the initially paused generator, and starts it going. It's the "extra next() call" in a sense, where you end up with one more` `next()` than `yield` . The subsequent `next()` calls all line up 1-for-1 with the currently paused `yield` , respectively.

Garrett Dawson

This is also confusing for me. I don't understand why the first next() doesn't give back the first yield expression, when it did in the first iterator example.

Kyle Simpson

@Garrett

It **does** give back the first `yield` expression, exactly as it did in the first iterator

example.

See this line:

```
console.log( it.next() );       // { value:6, done:false }
```

See how `value:6` came out? That's because the first `yield` expression was `yield (x + 1)`, which was `yield (5 + 1)`, which was `yield 6`. :)

zhiyelee

Before, there were two functions: `next` and `send`.
* `next` can not receive any arguments, and is used to resume and iterate a generator object.
* `send` can receive one argument which will be treated as the result of the last `yield` expression. `send` can not be used with a `newborn` generator, in other word we can only call `send` after we have called one or more `next`

Later, the `send` function was eliminated and replaced by an argument to `next`.

Hope that helps.

nightire

What do you mean by "before"? Is there another version of generators?

What about the future? Is the current spec stable?

However, nice explaination, now I think 2 methods would be better to understand.

Moshe Kolodny

As noted above, you can see that the `for..of` loop ignores and throws away the return 6 value. Also, since there's no exposed next() call, the for..of loop cannot be used in situations where you need to pass in values to the generator steps as we did above.

You can always change it to `return yield 6`

Kyle Simpson

`return yield 6` is likely going to end up as `yield 6; return undefined;` which is the same thing as `yield 6; return;` which is the same thing as `yield 6;`.

Of course you can change the final `return` to `yield`, but my whole point is to not rely on `return` IF you're going to consume it with `for..of`. :)

Dougal Campbell

Okay, I think I'm starting to wrap my brain around it. You can't think of `yield` like a function. With a regular function, we think "pass a value in, then get a value out". With yield, this is sort of reversed: "get a value out, then pass a value in". It might also help if you think of that first, empty `next()` call as the trigger to assign the argument values to the function? (I'm sure that's not correct in practice, but it's helping me to think of it that way)

So the first time you call `next()`, you unpause the generator function, and it parses up until it hits the first `yield`, at which point it yields the value of the `x+1` expression and stops, *waiting for an incoming value*.

The second time you call `next()`, the incoming value 12 is substituted for the *first* waiting `yield`, and y is calculated as `2 * 12`, or 24.

Execution then continues until it hits the `yield` on the next line (the second one), at which point we yield the value of `y / 3`, and the second `next()` receives the value 8. Execution pauses again, with this yield waiting for an incoming value.

Then we get the third `next()` call, which passes the value 13 in to the (second) waiting `yield`, and z is now assigned that value.

Execution continues. There are no more `yield`s, so the third `next()` receives the value of the `return` statement, x + y + z, which at this point is 5 + 24 + 13, or 42.

Kyle Simpson

You've got it exactly correct! :)

Kyle Simpson

@Dougal

Actually, let me build on/clarify something you've asserted here.

If you're thinking about the code inside the generator (the code with `yield`), then the proper way to think is "yield a value out, get a value back" — and you don't care or think about how that "get a value back" actually happens — (kinda like functions, but not really).

But if you're thinking about the iterator interface, it IS reversed as you asserted: you get a value out of the previous `next(..)`, you send a value back in with the next `next(..)`.

IOW, to call a `next(..)` call, you need to give it an an "answer" to the "question" the previous `next(..)`s return value "asked", and what you'll get back is a new return value

"question". You'll need to answer THAT new question with the next `next(..)`.

Yay! :)

RajYRaman

Thanks for the great explanation Doug. I had trouble understanding this part of the post, until I read your comment.

Anonymous

> The second time you call next(), the incoming value 12 is substituted for the *first* waiting yield, and y is calculated as 2 * 12, or 24.

I'm confused here... I thought 12 was passed to 2 * (x+1) thus resulting in 2 * (12 +1) which is 26 not 24.

Kyle Simpson

Ahh... I can see how that's confusing. Here's what's happening:

`(x + 1)` is whatever `x` is at that moment, which is `5` from the `foo(5)` call. So, the `5 + 1` value is `6`, and that value is `yield`ed out, thus you see the `{ value:6, done:false }`.

Now, when we call `it.next( 12 )`, `12` goes back in as the final result of the whole `yield ____` expression. So, it's `12`, not `12 + 1`.

If the code had said `(yield x) + 1` then you'd be correct, but order-of-operations wise, `yield (x + 1)` does the `x + 1` before the `yield`, not after it.

Make sense now?

Gregory Orton

@Kyle Simple (and others) – I have a question / want to state my understanding:

Would a useful analogy / paradigm be to think of the

```
yield _____
```

expression to almost be a hybrid / mash-up of both a variable identifier and an expression?

The left side is the variable identifier for the message that goes back in, and the right side is the message that goes out?

e.g. (I'm using vertical bars here to split up the yield expression

```
let y  = yield | (x+2)
//  'next' call before this yield expression sends out (x+2) (assuming X was set)
// the next 'next' call means anything left of | gets evaluated and assigned to y, an
```

Does this make sense?

Kyle Simpson

Gregory-

Yep, that's a pretty good way of modeling the behavior in a mental model. Good job!

Dougal Campbell

Consider this:

```
function* foo() {
  var a = yield "Hi there!";
  console.log("a = ", a);

  var b = a * 3 + (yield a+ 5);
  console.log("b = ", b);

  return b - 7;
```

There are 2 `yield` s in the function, so we know we'll have to call `next` 3 times (the first one just to kick things off).

```
var it = foo(); // get our iterator
console.log( "First next:", it.next() ); // don't pass anything in the first call

// First next: Object {value: "Hi there!", done: false} // from the first yield

console.log( "Second next:", it.next(4) );

// a = 4
// Second next: Object {value: 9, done: false} // from 'yield a + 5'

console.log( "Third next:", it.next(1) );
// b = 13  // a * 3 + 1 -> 4 * 3 + 1 -> 12 + 1
// Third next: Object {value: 6, done: true} // from return b - 7 -> 13 - 7
```

Does that help any?

David Walsh

Loads of great discussion here — glad to see people learning!

Chetan Dhembre

IMO `console.log` in example explaining what is going on will help a lot

Miron

Thanks for the explanation. At first, it was very difficult to get behind the generator iterator but after i have read it a few times i figured it out :)

Thank you very much!

Dougal Campbell

After it started to seep in a little, I was trying to think of good uses for generators. The first things I thought of were state machines, an in particular, a state machine as part of a task manager. I guess that instinct was pretty good, because a little googling turned up this article about various uses of generators in Python: http://excess.org/article/2013/02/itergen2/

Kyle Simpson

@Dougal-

Check this preview out: http://jsbin.com/luron/1/edit?js,console :)

Zoom

Thanks for the great introduction, generators rock! I'd love to see an example of how generators can be used as a state-machine solution. How would transitioning between yields work? It remains unclear at this point.

The current FSM options for JS seem overly complex or else they are not well supported. It would be great to see this design pattern solved elegantly by generators.

Kyle Simpson

@Zoom-

Here's a preview of exactly that, which I'll address in detail in part 4 of the series:

http://jsbin.com/luron/1/edit?js,console

Haoqun Jiang

>In any expression location, you can just use yield by itself in the expression/statement, and there's an assumed undefined value yielded out.

That statement seems not correct. Neither Node.js 0.11 or Firefox 31 can run the following sample code.

And the wiki page of ES6 Draft on generator (http://wiki.ecmascript.org/doku.php?id=harmony:generators) only says that 'return; is equivalent to return (void 0);' I don't think the same rule applies to the `yield` keyword.

Kyle Simpson

> That statement seems not correct.

It is. :)

All of these are valid (just double-checked with @awbjs):

```
yield;
x = yield;
foo( yield );
x = [ yield, yield ]
```

> Neither Node.js 0.11 or Firefox 31 can run

Chrome accepts them all. Firefox and node (since it's using an older v8 still) haven't caught up.

> And the wiki page of ES6 Draft on generator

That's pretty old. The most authoritative source (besides a TC39 member himself, like @awbjs) is the working draft for ES6: http://people.mozilla.org/~jorendorff/es6-draft.html

Haoqun Jiang

Thank you for the explanation. :)
I've just installed Chrome Canary and tried it out. All these expression are accepted.
Seems Chrome Stable haven't caught up either, which misled and drew me to the wrong conclusion.

shawpnendu

Please write more on generator...

Kyle Simpson

@shawpnendu-

Did you see part 2? http://davidwalsh.name/es6-generators-dive

Parts 3 and 4 will be out shortly (already written, just spacing out their publication). :)

idearat

Nice article! Helped get me started on these finally :).

It's a minor style nit but since the resulting Generator function's name will not include the '*' I think the function* foo syntax could be interpreted as being a bit more "accurate" in terms of what is ultimately created (e.g. a special function whose name is 'foo').

Kyle Simpson

This is under some debate right now. Some links to peruse:

gist.github.com/getify/710fda43c96d681bf2fb

http://www.2ality.com/2014/08/formatting-generator-asterisk.html

(the whole thread and various sub-threads)
https://twitter.com/getify/status/493588893573214208

I still believe `function *foo(){}` and `*foo()` and `yield *foo()` all make more sense. But there's definitely no consensus.

nightire

You didn't call a generator with `*` in your article, why you use

```
*foo()
```

here?

jacob

hey Kyle, possible to add some `console.log(it.next());` example outputs in the *Syntax Please!* section?

Rob Simpson

I had a very hard time understanding `*foo` but I think I might have a better grasp on it. I wrote a blog post where I try to step into it each time and what the context is before suspended and then the execution context that is returned after it resumed. Let me know if you read this whether I have my logic correct. http://robesimpson.me/2014/11/22/my-first-attempt-at-es6-generator/

Alberto Cole

Can you give some real life examples where generators are a better solution that what we have now in ES5?

Regards :)

Kyle Simpson

Alberto-

Essentially, every single snippet of "real life" code that you've ever written which was async and had two or more steps to it. You'll need to read the rest of the posts in this series to get a an idea of what I mean by that. But as a brief glimpse:

```
function ajax(url,cb) {
   // do some ajax request
   var content = ..
   cb(content);
}

ajax("http://some.url.1", function(text1) {
   ajax("http://some.url.2/?v=" + text1,function(text2) {
      console.log(text2);
   });
});
```

That sucks. Generators:

```
function ajax(url) {
   return new Promise(function(resolve){
      // do so some ajax request
      var content = ..
      resolve(content);
   });
}

run(function*(){

   var text1 = yield ajax("http://some.url.1");
   var text2 = yield ajax("http://some.url.2/?v=" + text1);

   console.log(text2);
});
```

Hopefully that glimpse plus the rest of the blog posts here will answer why generators will revolutionize "real world" async code all over the JS world.

Simon

I think you are confusing this with the await keyword?

cdenis

```
function *foo() {
  console.log(yield);
  console.log(yield);
  console.log(yield);
  console.log(yield);
  console.log(yield);
  console.log(yield);
}

const it = foo();
let i = 0;
while (!it.next(i * 10).done) {
  i++;
  console.log(i);
}
```

this dumb test helped me figure out the relation between next and yield call and return values, thanks a ton for the very well written article.

jikleshoog

I think a more inspired keyword would have been something like `initGen()` or `startGen()` instead of a no-arg `next()` in JS. Sending `next()` is almost like a hack to me.

jikleshoog

You mentioned you prefer to add the asterisk (*) to the function name rather than the function keyword, but for anonymous functions, you have no choice, but to place it around function keyword. Which makes your preference inconsistent a bit :)

Thanks for the great intro!

andyyou

Nice article, I've got a lot from this and I just a little bit curious that I want to know why isn't use `new` keyword for create generator iterator. Could you give me some keyword for search or post.

Thanks your article again :)

btw Coz I really like your article so I translate ur article to Traditional Chinese [here] (http://andyyou.logdown.com/posts/276655-es6-generators-teaching)

Jay Doubleyou

Thanks for the great article.

```
function *foo(x) {
    var y = 2 * (yield (x + 1));
    var z = yield (y / 3);
    return (x + y + z);
}

var it = foo( 5 );

// note: not sending anything into <code>next()</code> here
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

dardenfall

Thanks for the article. This verbiage is what I've come up with to help me grok and regrok this concept:

'yield' can be read as "Return what follows and stop; when .next is called, evaluate to whatever is passed in as an argument to next."

Alex

Hi. how i can change init example script with hello world message with generators?

wmehanna

Is there a way to use generator function with yied inside a class in Es2015 ?

I'm trying calling a generator function inside a class from the constructor, it runs but nothing happens (my console.log are not printing).

Here's an example of my code.

```
"use strict";

var parse = require("co-body");

var monk = require("monk");
var wrap = require("co-monk");
```

```
var db = monk("localhost/test");
var users  = wrap(db.get("users"));

class User {

    constructor(id) {

            var id = user;
            this.findOne*(id);
            console.log("findOne");

    }


     *findOne(id){
        console.log("1");
        var userData = yield users.findOne(id);
        console.log("2");
        this._user = userData;
        console.log("3");
        return this;
    }


}

var _user = new User(1111);
console.log(_user);

export default User;
```

mizuki

I think it would be less confusing if you chose different values to pass the `next()` calls. Passing in `12` which matches with `2 * (x + 1)` from the first call makes it a bit tricky on a first glance.

I guess that's just the pitfall of human pattern matching kicking in too early, but I also remember our math teacher showing us how in these 'learn by example' situations it is best to avoid such coincidences.

Ovi-Wan Kenobi

```
var it = foo( 5 );
```

it does not execute any line of code inside the iterator. first `next()` is actually executing the iterator up to the first yield...

confusing.

also first next throws away the value passed in.

confusing.

the rest is OK simple :)
thanks for explaining this. but without comments the article (for me at least) would not cleared all things up. comments and answer to comments made light in my head :)

Avinash Thakur

@kyle you don't know generators :)

KB

Excellent article. Demystified yield and generations.
Furthermore, your discussion with Doug in comments section added to the clarity.

Thank you.

Name | Email | Website

Wrap your code in `<pre class="{language}"></pre>` *tags, link to a GitHub gist, JSFiddle fiddle, or CodePen pen to embed!*

Continue this conversation via email

Post Comment! Use Code Editor