Most popular (last 30 days)

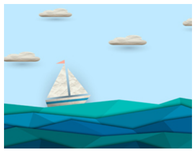
ECMAScript 2017: the final feature set
ECMAScript 6 modules: the final syntax
ES proposal: import() – dynamically importing ES modules
Making transpiled ES modules more spec-compliant
ES proposal: Shared memory and atomics
Classes in ECMAScript 6 (final semantics)
Communicating between Web Workers via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



Dr. Axel Rauschmayer

2012-06-22

Asynchronous programming and continuation-passing style in JavaScript

Labels: advancedjs, async, dev, javascript, jslang

In this blog post, we give a name to JavaScript's callback-based asynchronous programming style: *continuation-passing style* (CPS). We explain how CPS works and give tips for using it.

1. Asynchronous programming and callbacks

If you have ever done asynchronous programming in JavaScript, you probably noticed that everything works differently: Instead of returning a value, one passes it on to a callback. For example, a synchronous program looks as follows.

```
function loadAvatarImage(id) {
    var profile = loadProfile(id);
    return loadImage(profile.avatarUrl);
}
```

However, tasks such as loading a profile can take a while and are best handled asynchronously. Then one calls `loadProfile` with an additional argument, a callback. It returns immediately and one can go on to do different things. Once the profile has been loaded, the callback is invoked and receives the profile as an argument. Now you can perform the next step, loading the image. That leads to an asynchronous, callback-based programming style that looks as follows:

```
function loadAvatarImage(id, callback) {
    loadProfile(id, function (profile) {
        loadImage(profile.avatarUrl, callback);
    });
}
```

This asynchronous programming style is called *continuation-passing style* (CPS). The synchronous programming style is called *direct style*. The name CPS is due to the fact that you always pass a callback as the last argument to functions. That callback continues the execution of the function, is the next step to perform. It is thus often called a *continuation*, especially in functional programming. The problem with CPS is that it is contagious, an all-or-nothing proposition: `loadAvatarImage` uses CPS internally, but it can't hide that fact from the outside, it must be written in CPS, too. The same holds for everyone who invokes `loadAvatarImage`.

2. Converting to continuation-passing style

This section shows a few techniques that make it easier to translate code from direct style to continuation-passing style.

2.1. Sequences of function invocations

Very often, you will simply have a sequence of function or method invocations. In the previous example, you have seen that that leads to ugly nesting of functions. However, you can avoid that by using function declarations:

```
function loadAvatarImage(id, callback) {
    loadProfile(id, loadProfileAvatarImage);  // (*)
    function loadProfileAvatarImage(profile) {
            loadImage(profile.avatarUrl, callback);
        }
    }
```

JavaScript *hoists* the function `loadProfileAvatar` (moves it to the beginning of the function). Hence, it can be called at (*). We nested `loadProfileAvatarImage` inside `loadAvatarImage`, because it needed access to `callback`. You will see this kind of nesting whenever there is state to be shared between function invocations. An alternative is to use an Immediately-Invoked Function Expression (IIFE, [1]):

```
    var loadAvatarImage = function () {
        var cb;
        function loadAvatarImage(id, callback) {
            cb = callback;
            loadProfile(id, loadProfileAvatarImage);
        }
        function loadProfileAvatarImage(profile) {
            loadImage(profile.avatarUrl, cb);
        }
        return loadAvatarImage;
    }();
```

2.2. Iterating over an array

The following code contains a simple `for` loop.

```
    function logArray(arr) {
        for(var i=0; i < arr.length; i++) {
            console.log(arr[i]);
        }
        console.log("### Done");
    }
```

Let us convert it to CPS in two steps. The first step is to use recursion for iteration. That is a common technique in functional programming. The following code is still in direct style.

```
    function logArray(arr) {
        logArrayRec(0, arr);
        console.log("### Done");
    }
    function logArrayRec(index, arr) {
        if (index < arr.length) {
            console.log(arr[index]);
            logArrayRec(index+1, arr);
        }
        // else: done
    }
```

Now it is easier to convert the code to CPS. We do so by introducing a helper function `forEachCps`.

```
    function logArray(arr) {
        forEachCps(arr, function (elem, index, next) {  // (*)
            console.log(elem);
            next();
        }, function () {
            console.log("### Done");
        });
    }
    function forEachCps(arr, visitor, done) {  // (**)
        forEachCpsRec(0, arr, visitor, done)
    }
    function forEachCpsRec(index, arr, visitor, done) {
        if (index < arr.length) {
            visitor(arr[index], index, function () {
                forEachCpsRec(index+1, arr, visitor, done);
            });
        } else {
            done();
```

Tweets by @rauschma

Axel Rauschm… @rauschma
Enjoying “Troll Hunters”:
– “Let’s call him ‘Gnome Chomsky’”
– “Juliet dies in this? Nooo!”
10h

Axel Rauschmayer Retweeted
Jonathan Creamer @jcreamer898
A nice little shout out to @rauschma… infoworld.com/article/316483 … #es2017 #async
JavaScr… ECMASc… infoworld…
14h

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

► January (19)

► 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)
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)

```
        }
    }
```

There are two interesting changes: The visitor at (*) gets its own continuation, `next`, which triggers the next step “inside” `forEachCpsRec`. That lets us make CPS calls in the visitor, e.g. to perform an asynchronous request. We also have to provide a continuation done at (**) to specify what happens after the loop is finished.

2.3. Mapping an array

If we slightly rewrite `forEachCps`, we get the CPS version of `Array.prototype.map`.

```
    function mapCps(arr, func, done) {
        mapCpsRec(0, [], arr, func, done)
    }
    function mapCpsRec(index, outArr, inArr, func, done) {
        if (index < inArr.length) {
            func(inArr[index], index, function (result) {
                mapCpsRec(index+1, outArr.concat(result),
                          inArr, func, done);
            });
        } else {
            done(outArr);
        }
    }
```

`mapCps` takes an array-like object and produces a new array by applying `func` to each element. The above version is non-destructive, it creates a new array for each recursion step. The following is a destructive variation:

```
    function mapCps(arrayLike, func, done) {
        var index = 0;
        var results = [];

        mapOne();

        function mapOne() {
            if (index < arrayLike.length) {
                func(arrayLike[index], index, function (result) {
                    results.push(result);
                    index++;
                    mapOne();
                });
            } else {
                done(results);
            }
        }
    }
```

`mapCps` is used as follows.

```
    function done(result) {
        console.log("RESULT: "+result);  // RESULT: ONE,TWO,THREE
    }
    mapCps(["one", "two", "three"],
        function (elem, i, callback) {
            callback(elem.toUpperCase());
        },
        done);
```

Variation: parallel map. The sequential version of `mapCps` is not as efficient as it could be. For example, if each mapping step involves a server request, it sends the first request, waits for the result, sends the second request, etc. Instead, it would be better to send all requests and then wait for the results. Extra care has to be taken to ensure that they are added to the output array in the proper order. The following code does all that.

```
    function parMapCps(arrayLike, func, done) {
        var resultCount = 0;
        var resultArray = new Array(arrayLike.length);
        for (var i=0; i < arrayLike.length; i++) {
            func(arrayLike[i], i, maybeDone.bind(null, i));  // (*)
```

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)
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)

```
        }
        function maybeDone(index, result) {
            resultArray[index] = result;
            resultCount++;
            if (resultCount === arrayLike.length) {
                done(resultArray);
            }
        }
    }
```

At (*), we must copy the current value of the loop variable `i`. If we don't copy, we will always get the current value of `i` in the continuation. For example, `arrayLike.length`, if the continuation is invoked after the loop has finished. The copying can also be done via an IIFE or by using `Array.prototype.forEach` instead of a `for` loop.

2.4. Iterating over a tree

Given the following direct style function that recurses over a tree of nested arrays.

```
    function visitTree(tree, visitor) {
        if (Array.isArray(tree)) {
            for(var i=0; i < tree.length; i++) {
                visitTree(tree[i], visitor);
            }
        } else {
            visitor(tree);
        }
    }
```

That function is used as follows:

```
    > visitTree([[1,2],[3,4], 5], function (x) { console.log(x) })
    1
    2
    3
    4
    5
```

If you want to allow `visitor` to make asynchronous requests, you have to rewrite `visitTree` in CPS:

```
    function visitTree(tree, visitor, done) {
        if (Array.isArray(tree)) {
            visitNodes(tree, 0, visitor, done);
        } else {
            visitor(tree, done);
        }
    }
    function visitNodes(nodes, index, visitor, done) {
        if (index < nodes.length) {
            visitTree(nodes[index], visitor, function () {
                visitNodes(nodes, index+1, visitor, done);
            });
        } else {
            done();
        }
    }
```

We also have the option of using `forEachCps`:

```
    function visitTree(tree, visitor, done) {
        if (Array.isArray(tree)) {
            forEachCps(
                tree,
                function (subTree, index, next) {
                    visitTree(subTree, visitor, next);
                },
                done);
        } else {
            visitor(tree, done);
        }
    }
```

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)
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)

2.5. Pitfall: Execution continues after passing a result

In direct style, returning a value terminates a function:

```
function abs(n) {
    if (n < 0) return -n;
    return n;  // (*)
}
```

Hence, (*) is not executed if n is less than zero. In contrast, returning a value in CPS at (**) does not terminate a function:

```
// Wrong!
function abs(n, success) {
    if (n < 0) success(-n);  // (**)
    success(n);
}
```

Hence, if `n < 0` then both `success(-n)` and `success(n)` are called. The fix is easy – write a complete `if` statement.

```
function abs(n, success) {
    if (n < 0) {
        success(-n);
    } else {
        success(n);
    }
}
```

It takes some getting used to that in CPS, the logical control flow continues via the continuation, but the physical control flow doesn't (yet).

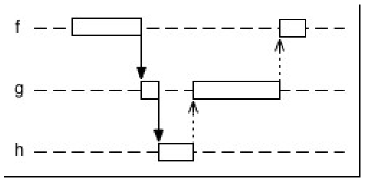
3. CPS and control flow

CPS reifies "the next step" – it "turns it into a thing" (the definition of "to reify") that you can work with. In direct style, a function is powerless over what happens after its invocation, in CPS, it has complete control. A so-called "inversion of control" happened. Let's take a closer look at the control flow in both styles.

Direct style. You call a function and it must return to you, it can't escape the nesting that happens with function calls. The following code contains two such calls: `f` calls `g` which calls `h`.

```
function f() {
    console.log(g());
}
function g() {
    return h();
}
function h() {
    return 123;
}
```

The control flow as a diagram:



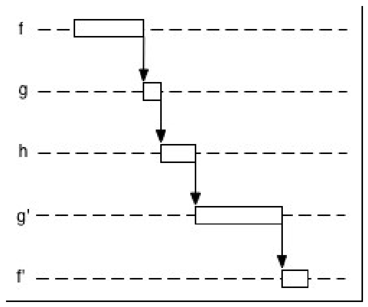
Continuation-passing style. A function determines where to go next. It can decide to continue "as ordered" or to do something completely different. The following code is the CPS version of the previous example.

```
function f() {
    g(function (result) {
        console.log(result);
    });
}
function g(success) {
    h(success);
```

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)
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)

```
    }
    function h(success) {
        success(123);
    }
```

Now, the control flow is completely different. `f` calls `g` calls `h` which then calls `g`'s continuation `g'` which calls `f'`. The control flow as a diagram:



3.1. Return

As a first illustration of how much power over the control flow a function has, let us look at the following code. We then rewrite it slightly and create a helper function that performs the equivalent of a `return` for the caller(!)

```
    function searchArray(arr, searchFor, success, failure) {
        forEachCps(arr, function (elem, index, next) {
            if (compare(elem, searchFor)) {
                success(elem);  // (*)
            } else {
                next();
            }
        }, failure);
    }
    function compare(elem, searchFor) {
        return (elem.localeCompare(searchFor) === 0);
    }
```

CPS allows us to immediately exit the loop at (*). In `Array.prototype.forEach`, we cannot do that, we have to wait until the loop is finished. If we rewrite `compare` in CPS then it automatically returns for us from the loop.

```
    function searchArray(arr, searchFor, success, failure) {
        forEachCps(arr, function (elem, index, next) {
            compareCps(elem, searchFor, success, next);
        }, failure);
    }
    function compareCps(elem, searchFor, success, failure) {
        if (elem.localeCompare(searchFor) === 0) {
            success(elem);
        } else {
            failure();
        }
    }
```

That is quite astonishing. To achieve a similar effect in direct style, we would need exceptions.

3.2. try-catch

With CPS, you can even implement exception handling in the language – there is no need for a special language construct. In the following example, we implement a function `printDiv` in CPS. It calls the CPS function `div` which might throw an exception. Hence, it wraps that call in `tryIt`, our implementation of try-catch.

```
    function printDiv(a, b, success, failure) {
        tryIt(
            function (succ, fail) {  // try
                div(a, b, function (result) {  // might throw
                    console.log(result);
                    succ();
                }, fail);
```

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)
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)

```
            },
            function (errorMsg, succ, fail) {  // catch
                handleError(succ, fail);  // might throw again
            },
            success,
            failure
        );
    }
```

To make exception handling work, every function needs two continuations; one for successful termination and one if a failure happens. The function `try` implements the try-catch statement. Its first argument is the try block which has its own local versions of the success and failure continuation. The second argument is the catch block which again has local continuations. The last two arguments are the continuations that apply to the function as a whole. The CPS division `div` throws an exception if the divisor is zero.

```
    function div(dividend, divisor, success, failure) {
        if (divisor === 0) {
            throwIt("Division by zero", success, failure);
        } else {
            success(dividend / divisor);
        }
    }
```

And now the implementation of exception handling.

```
    function tryIt(tryBlock, catchBlock, success, failure) {
        tryBlock(
            success,
            function (errorMsg) {
                catchBlock(errorMsg, success, failure);
            });
    }
    function throwIt(errorMsg, success, failure) {
        failure(errorMsg);
    }
```

Note that the continuations of the catch block are statically determined, they are not passed to it when the failure continuation is invoked. They are the same continuations that the complete `tryIt` function has.

3.3. Generator

Generators are an ECMAScript.next feature that you can already try out in the current Firefox version [2]. A generator is an object that wraps a function. Every time one invokes the method `next()` on the object, the execution of the function continues. Every time the function performs `yield` *`value`*, the execution is paused and `next()` returns with *`value`*. The following generator produces an infinite sequence of numbers 0, 1, 2, ...

```
    function* countUp() {
        for(let i=0;; i++) {
            yield i;
        }
    }
```

Note the infinite loop in the function wrapped by the generator object. That infinite loop is continued every time `next()` is called and paused every time `yield` is used. Example interaction:

```
    > let g = countUp();
    > g.next()
    0
    > g.next()
    1
```

If we implement generators via CPS, the distinction between the generator function and the generator object becomes more obvious. First, let’s write the generator as a function `countUpCps`.

```
    function countUpCps() {
        var i=0;
        function nextStep(yieldIt) {
            yieldIt(i++, nextStep);
        }
        return new Generator(nextStep);
    }
```

`countUpCps` wraps a generator object around a generator function that is written in CPS. It is used as follows:

```
    var g = countUpCps();
    g.next(function (result) {
        console.log(result);
        g.next(function (result) {
            console.log(result);
            // etc.
        });
    });
```

The constructor for generator objects can be implemented as follows.

```
    function Generator(genFunc) {
        this._genFunc = genFunc;
    }
    Generator.prototype.next = function (success) {
        this._genFunc(function (result, nextGenFunc) {
            this._genFunc = nextGenFunc;
            success(result);
        });
    };
```

Note how we store the current continuation of the generator function inside the object. We don't pass it on.

4. CPS and the stack

Another interesting aspect of CPS is that it makes the stack obsolete, because you always continue and never return. That means that if your program is completely in CPS then you only need mechanisms for jumping to a function and for creating an environment (to hold parameters and local variables). But no stack. In other words, a function call in CPS is very similar to a `goto` statement. Let's look an example that illustrates that. The following is a function with a `for` loop:

```
    function f(n) {
        var i=0;
        for(; i < n; i++) {
            if (isFinished(i)) {
                break;
            }
        }
        console.log("Stopped at "+i);
    }
```

The same program implemented via `goto` looks like this:

```
    function f(n) {
        var i=0;
    L0: if (i >= n) goto L1;
        if (isFinished(i)) goto L1;
        i++;
        goto L0;
    L1: console.log("Stopped at "+i);
    }
```

The CPS version (ignoring `isFinished`) is not much different:

```
    function f(n) {
        var i=0;
        L0();
        function L0() {
            if (i >= n) {
                L1();
            } else if (isFinished(i)) {
                L1();
            } else {
                i++;
                L0();
            }
        }
        function L1() {
            console.log("Stopped at "+i);
        }
    }
```

4.1. Tail calls

Let’s take one more look at the following direct style function that uses recursion to iterate over an array:

```
    function logArrayRec(index, arr) {
        if (index < arr.length) {
            console.log(arr[index]);
            logArrayRec(index+1, arr);  // (*)
        }
        // else: done
    }
```

In current JavaScript, the stack will grow for each additional array element. However, if we look closer, we realize that it is unnecessary to preserve the stack while making the self-recursive call at (*). It is the last call in the function, so there is nothing to return to. Instead, one could remove the data of the current function from the stack before making the call and the stack would not grow. If a function call comes last in a function it is called a *tail call*. Most functional programming languages make the aforementioned optimization. Which is why looping via recursion is as efficient in those languages as an iterative construct (e.g. a `for` loop). All true CPS function calls are tail calls and can be optimized. A fact that we hinted at when we mentioned that they are similar to `goto` statements.

4.2. Trampolining

CPS is a popular intermediate format for compilers of function languages, because many control flow constructs can be elegantly expressed and tail call optimization is easy. When compiling to a language without optimized tail calls, one can avoid stack growth via a technique called *trampolining*. The idea is to not make the final continuation call inside the function, but to exit and to return the continuation to a *trampoline*. That trampoline is simply a loop that invokes the returned continuations. Hence, there are no nested function calls and the stack won’t grow. For example, the previous CPS code can be rewritten as follows to support a trampoline.

```
    function f(n) {
        var i=0;
        return [L0];
        function L0() {
            if (i >= n) {
                return [L1];
            } else if (isFinished(i)) {
                return [L1];
            } else {
                i++;
                return [L0];
            }
        }
        function L1() {
            console.log("Stopped at "+i);
        }
    }
```

In CPS, every function call is always a tail call. We transform each call

```
    func(arg1, arg2, arg3);
```

into a `return` statement

```
    return [func, [arg1, arg2, arg3]];
```

The trampoline picks up the returned arrays and performs the corresponding function call.

```
function trampoline(result) {
    while(Array.isArray(result)) {
        var func = result[0];
        var args = (result.length >= 2 ? result[1] : []);
        result = func.apply(null, args);
    }
}
```

We now invoke `f` like this:

```
trampoline(f(14));
```

4.3. The event queue and trampolining

In browsers and Node.js trampolining is performed via the event queue. If you have many CPS calls in a row (without waiting for an asynchronous result via the event queue), you can push the continuation to that queue and avoid a stack overflow. Hence, instead of

```
continuation(result);
```

you write

```
setTimeout(function () { continuation(result) }, 0);
```

Node.js even has the special function `process.nextTick()` for this purpose:

```
process.nextTick(function () { continuation(result) });
```

5. Conclusion

JavaScript's style of asynchronous programming is efficient, because you don't have to create new processes. And it is easy to understand. But it can also quickly become unwieldy. It therefore helps to have more background on it, which this blog post provided by explaining continuation-passing style. It also showed some techniques that make CPS more bearable, but there are more out there. Those will be the topic of an upcoming blog post (quick teaser: promises).

6. Related reading

[1] JavaScript variable scoping and its pitfalls

[2] Trying out ECMAScript.next's for...of loop in Firefox 13

8 Comments The 2ality blog 1 Login

Recommend Share Sort by Best

Join the discussion…

Marcin Sas-Szymański • 3 years ago

Probably the best explanation of cps on the internet.

2 • Reply • Share ›

pvillela Marcin Sas-Szymański • a day ago

Yes, thank you Axel for the excellent explanation of CPS. While promises and async-await are the way to do asynchronous coding in JavaScript nowadays, it is important to understand the CPS foundation upon which all asynchronous programming in JavaScript is based.

• Reply • Share ›

Erik Schoel • 3 years ago

Have you ever seen this one? http://www.cs.umd.edu/projects... Could work, will give it a try at least.

1 • Reply • Share ›

schlamar • 4 years ago

Nested callbacks are really ugly (section 2.1) but your solution is even more. Please use Promises (aka deferred) or (even better) coroutines. See this presentation: http://slideshare.net/domenicd...

1 • Reply • Share ›

Axel Rauschmayer Mod schlamar • 4 years ago

Yes, Promises are almost always preferable, as mentioned in the last paragraph of this post. This blog post is about explaining the foundations of asynchronous programming.

1 • Reply • Share ›

Alex McPherson • 5 years ago

I just gave a talk at jQuery conf 2012 about Deferreds that you might be interested in as well (I discussed the jQuery implementation, but there are dependency-less ports available all over as well)

https://github.com/alexmcphers...

1 • Reply • Share ›

Ivan Kurnosov • 5 years ago

Probably it worth mentioning about $.Deferred from jQuery, which makes doing async stuff much easier

1 • Reply • Share ›

it-ony • 5 years ago

Have a look at https://github.com/it-ony/flow...

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)