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)

2014-09-23

ECMAScript 6 promises (1/2): foundations

Labels: async, dev, esnext, javascript, jslang, promises

This blog post is outdated. Please read chapter "Asynchronous programming (background)" in "Exploring ES6".

This blog post explains foundations of asynchronous programming in JavaScript. It is first in a series of two posts and prepares you for part two, which covers promises and the ECMAScript 6 promise API.

1. The JavaScript call stack

When a function `f` calls a function `g`, `g` needs to know where to return to (inside `f`) after it is done. This information is usually managed with a stack, the *call stack*. Let's look at an example.

```
function h(z) {
    // Print stack trace
    console.log(new Error().stack); // (A)
}
function g(y) {
    h(y + 1); // (B)
}
function f(x) {
    g(x + 1); // (C)
}
f(3); // (D)
return; // (E)
```

Initially, when the program above is started, the call stack is empty. After the function call `f(3)` in line (D), the stack has one entry:

- Location in global scope

After the function call `g(x + 1)` in line (C), the stack has two entries:

- Location in `f`
- Location in global scope

After the function call `h(y + 1)` in line (B), the stack has three entries:

- Location in `g`
- Location in `f`
- Location in global scope

The stack trace printed in line (A) shows you what the call stack looks like:

```
Error
    at h (stack_trace.js:2:17)
    at g (stack_trace.js:6:5)
```

Dr. Axel Rauschmayer

```
    at f (stack_trace.js:9:5)
    at <global> (stack_trace.js:11:1)
```

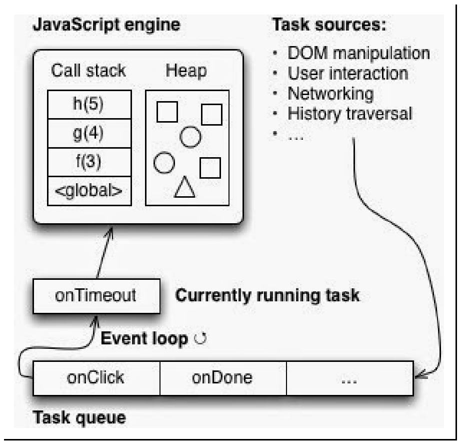
Next, each of the functions terminates and each time the top entry is removed from the stack. After function `f` is done, we are back in global scope and the call stack is empty. In line (E) we return and the stack is empty, which means that the program terminates.

2. The browser event loop

Simplifyingly, each browser tab runs (in) a single process: the *event loop*. This loop executes browser-related things (so-called *tasks*) that it is fed via a *task queue*. Examples of tasks are:

1. Parsing HTML
2. Executing JavaScript code in script elements
3. Reacting to user input (mouse clicks, key presses, etc.)
4. Processing the result of an asynchronous network request

Items 2–4 are tasks that run JavaScript code, via the engine built into the browser. They terminate when the code terminates. Then the next task from the queue can be executed. The following diagram (inspired by a slide by Philip Roberts [1]) gives an overview of how all these mechanisms are connected.



The event loop is surrounded by other processes running in parallel to it (timers, input handling, etc.). These processes communicate with it by adding tasks to its queue.

2.1. Timers

Browsers have timers. `setTimeout()` creates a timer, waits until it fires and then adds a task to the queue. It has the signature:

```
setTimeout(callback, ms)
```

After `ms` milliseconds, `callback` is added to the task queue. It is important to note that `ms` only specifies when the callback is *added*, not when it actually executed. That may happen much later, especially if the event loop is blocked (as demonstrated later in this post).

`setTimeout()` with `ms` set to zero is a commonly used work-around to add something to the task queue right away. However, some browsers do not allow `ms` to be below a minimum (4 ms in Firefox); they set it *to* that minimum if it is.

2.2. Displaying DOM changes

For most DOM changes (especially those involving a re-layout), the display isn't updated right away. "Layout happens off a refresh tick every 16ms" (@bz_moz) and must be given a chance to run via the event loop.

There are ways to coordinate frequent DOM updates with the browser, to avoid clashing with its layout rhythm. Consult the documentation on `requestAnimationFrame()` for details.

2.3. Run-to-completion semantics

▶ 2015 (65)
▼ 2014 (55)
▶ December (4)
▶ November (1)
▶ October (2)
▼ September (4)
ECMAScript 6 promises (1/2): foundations
ECMAScript 6 modules: the final syntax
Standardizing (a flavor of) Markdown
Preventing function signature mismatch in ES5 and …
▶ August (3)
▶ July (7)
▶ June (7)
▶ May (8)
▶ April (6)
▶ March (1)
▶ February (5)
▶ January (7)
▶ 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)
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)

JavaScript has so-called run-to-completion semantics: The current task is always finished before the next task is executed. That means that each task has complete control over all current state and doesn't have to worry about concurrent modification.

Let's look at an example:

```
setTimeout(function () { // (A)
    console.log('Second');
}, 0);
console.log('First'); // (B)
```

The function starting in line (A) is added to the task queue immediately, but only executed after the current piece of code is done (in particular line (B)!). That means that this code's output will always be:

```
First
Second
```

2.4. Blocking the event loop

As we have seen, each tab (in some browers, the complete browser) is managed by a single process – both the user interface and all other computations. That means that you can freeze the user interface by performing a long-running computation in that process. The following code demonstrates that. You can try it out online.

```
<a id="block" href="">Block for 5 seconds</a>
<p>
<button>Simple button</button>
<script>
    ...
    function onClick(event) {
        event.preventDefault();

        console.log('Blocking...');
        sleep(5000);
        console.log('Done');
    }
    function sleep(milliseconds) {
        var start = Date.now();
        while ((Date.now() - start) < milliseconds);
    }
</script>
```

Whenever the link at the beginning is clicked, the function `onClick()` is triggered. It uses the – synchronous – `sleep()` function to block the event loop for five seconds. During those seconds, the user interface doesn't work. For example, you can't click the "Simple button".

2.5. Avoiding blocking

You avoid blocking the event loop in two ways:

First, you don't perform long-running computations in the main process, you move them to a different process. This can be achieved via the Worker API.

Second, you don't (synchronously) wait for the results of a long-running computation (your own algorithm in a Worker process, a network request, etc.), you carry on with the event loop and let the computation notify you when it is finished. In fact, you usually don't even have a choice in browsers and have to do things this way. For example, there is no built-in way to sleep synchronously (like the previously implemented `sleep()`). Instead, `setTimeout()` lets you sleep asynchronously.

The next section explains techniques for waiting asynchronously for results.

3. Receiving results asynchronously

Two common patterns for receiving results asynchronously are: events and callbacks.

3.1. Asynchronous results via events

In this pattern for asynchronously receiving results, you create an object for each request and register event handlers with it: one for a successful computation, another

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)
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)

one for handling errors. The following code shows how that works with the `XMLHttpRequest` API:

```
var req = new XMLHttpRequest();
req.open('GET', url);

req.onload = function() {
    if (req.status == 200) {
        processData(req.response);
    } else {
        console.log('ERROR', req.statusText);
    }
};

req.onerror = function() {
    console.log('Network Error');
};

req.send(); // Add request to task queue
```

Note that the last line doesn't actually perform the request, it adds it to the task queue. Therefore, you could also call that method right after `open()`, before setting up `onload` and `onerror`. Things would work the same, due to JavaScript's run-to-completion semantics.

If you are used to multi-threaded programming languages, IndexedDB requests look like they might be prone to race conditions. However, run to completion makes the following code safe in JavaScript:

```
var openRequest = indexedDB.open('test', 1);

openRequest.onsuccess = function(event) {
    console.log('Success!');
    var db = event.target.result;
};

openRequest.onerror = function(error) {
    console.log(error);
};
```

`open()` does not immediately open the database, it adds a task to the queue, which is executed after the current task is finished. That is why you can (and in fact must) register event handlers *after* calling `open()`.

3.2. Asynchronous results via callbacks

If you handle asynchronous results via callbacks, you pass callback functions as trailing parameters to asynchronous function or method calls.

The following is an example in Node.js. We read the contents of a text file via an asynchronous call to `fs.readFile()`:

```
// Node.js
fs.readFile('myfile.txt', { encoding: 'utf8' },
    function (error, text) { // (A)
        if (error) {
            // ...
        }
        console.log(text);
    });
```

If `readFile()` is successful, the callback in line (A) receives a result via the parameter `text`. If it isn't, the callback gets an error (often an instance of `Error` or a sub-constructor) via its first parameter.

The same code in classic functional programming style would look like this:

```
// Functional
readFileFunctional('myfile.txt', { encoding: 'utf8' },
```

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)
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)

```
    function (text) { // success
        console.log(text);
    },
    function (error) { // failure
        // ...
    });
```

3.3. Continuation-passing style

The programming style of using callbacks (especially in the functional manner shown previously) is also called *continuation-passing style* (CPS), because the next step (the *continuation*) is explicitly passed as a parameter. This gives an invoked function more control over what happens next and when.

The following code illustrates CPS:

```
console.log('A');
identity('B', function step2(result2) {
    console.log(result2);
    identity('C', function step3(result3) {
       console.log(result3);
    });
    console.log('D');
});
console.log('E');

// Output: A E B D C

function identity(input, callback) {
    setTimeout(function () {
        callback(input);
    }, 0);
}
```

For each step, the control flow of the program continues inside the callback. This leads to nested functions, which are sometimes referred to as *callback hell*. However, you can often avoid nesting, because JavaScript's function declarations are *hoisted* (their definitions are evaluated at the beginning of their scope). That means that you can call ahead and invoke functions defined later in the program. The following code uses hoisting to flatten the previous example.

```
console.log('A');
identity('B', step2);
function step2(result2) {
    // The program continues here
    console.log(result2);
    identity('C', step3);
    console.log('D');
}
function step3(result3) {
   console.log(result3);
}
console.log('E');
```

[3] contains more information on CPS.

3.4. Composing code in CPS

In normal JavaScript style, you compose pieces of code via:

1. Putting them one after another. This is blindingly obvious, but it's good to remind ourselves that concatenating code in normal style is sequential composition.
2. Array methods such as `map()`, `filter()` and `forEach()`
3. Loops such as `for` and `while`

The library Async.js provides combinators to let you do similar things in CPS, with Node.js-style callbacks. It is used in the following example to load the contents of three files, whose names are stored in an array.

```
    var async = require('async');

    var fileNames = [ 'foo.txt', 'bar.txt', 'baz.txt' ];
    async.map(fileNames,
        function (fileName, callback) {
            fs.readFile(fileName, { encoding: 'utf8' }, callback);
        },
        // Process the result
        function (error, textArray) {
            if (error) {
                console.log(error);
                return;
            }
            console.log('TEXTS:\n' + textArray.join('\n----\n'));
        });
```

3.5. Pros and cons of callbacks

Using callbacks results in a radically different programming style, CPS. The main advantage of CPS is that its basic mechanisms are easy to understand. However, it has disadvantages:

- Error handling becomes more complicated: There are now two ways in which errors are reported – via callbacks and via exceptions. You have to be careful to combine both properly.
- Less elegant signatures: In synchronous functions, there is a clear separation of concerns between input (parameters) and output (function result). In asynchronous functions that use callbacks, these concerns are mixed: the function result doesn't matter and some parameters are used for input, others for output.
- Composition is more complicated: Because the concern "output" shows up in the parameters, it is more complicated to compose code via combinators.

Callbacks in Node.js style have three disadvantages (compared to those in a functional style):

- The `if` statement for error handling adds verbosity.
- Reusing error handlers is harder.
- Providing a default error handler is also harder. A default error handler is useful if you make a function call and don't want to write your own handler. It could also be used by a function if a caller doesn't specify a handler.

4. Looking ahead

The second part of this series covers promises and the ECMAScript 6 promise API. Promises are more complicated under the hood than callbacks. In exchange, they bring several significant advantages.

5. Further reading

1. "Help, I'm stuck in an event-loop" by Philip Roberts (video).
2. "Event loops" in the HTML Specification.
3. Asynchronous programming and continuation-passing style in JavaScript

6. Reviewers

I'd like to thank the following people for reviewing this post.

- Andrea Giammarchi
- Philip Roberts

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)
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)
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)

6 Comments **The 2ality blog** 1 **Login**

Recommend 2 **Share** **Sort by Best**

Join the discussion…

cendrizzi • 2 years ago
Wonderful post. Thank you.
1 • Reply • Share ›

Šime Vidas • 2 years ago
Explaining that diagram won't be an easy feat :)
1 • Reply • Share ›

Axel Rauschmayer Mod Šime Vidas • 2 years ago
Yes, there are lots of arrows. However, I'll be building it up in small steps, which helps with understanding the end result.
• Reply • Share ›

CYB • 2 years ago
Nice post, thank you :) Looking forward to reading part 2!
• Reply • Share ›

feenan • 2 years ago
nice post,thank you!
• Reply • Share ›

Bangfu Tao feenan • 7 months ago
Good technical explanation!
• Reply • Share ›

Subscribe **Add Disqus to your site Add Disqus Add** **Privacy**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)