Event Loop: la naturaleza asincrónica de Javascript

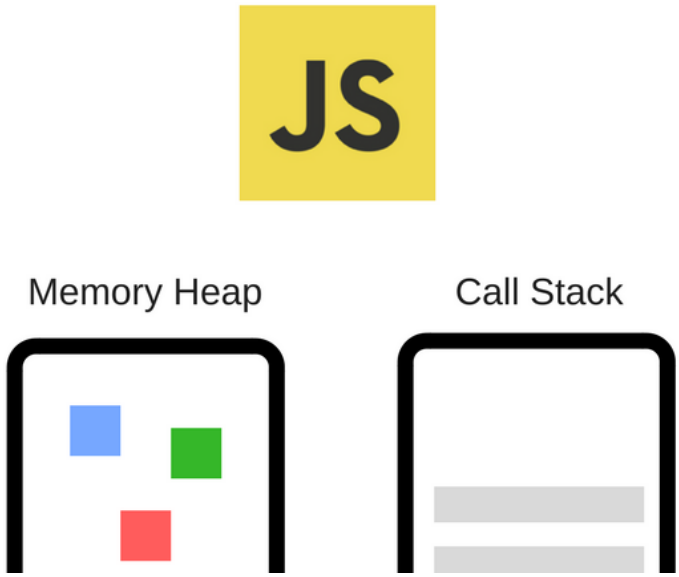
ubykuo Follow
Sep 20, 2017 · 6 min read

Decidí escribir este artículo ya que pienso que es importante comprender todos los aspectos del lenguaje para convertirse en un desarrollador profesional (o al menos uno bueno). Es algo que, lamentablemente, en javascript no es común.

Debido a la gran cantidad de librerías y herramientas presentes en este lenguaje, muchos programadores comienzan a desarrollar aplicaciones sin tener un entendimiento real de cómo es que las cosas funcionan “por debajo”.

En este caso voy a hablar de uno de estos aspectos, el cual es muy importante. El famoso event loop.

Antes que nada miremos un dibujo que representa el runtime de v8 (el runtime que usa chrome y node)



fuente: https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf

Como se puede ver en la imagen, el engine consiste de dos elementos principales

- Memory Heap: es donde se realiza la alocación de memoria
- Call Stack: es donde el runtime mantiene un track de las llamadas a las funciones

Solo hablaremos de la call stack, que es la que se relaciona con el Event Loop.

Call Stack

Para los que no sepan un stack (también llamado pila) es una estructura simple, similar a un arreglo en el que solo se puede agregar items al final (push) , y remover el último (pop).

El proceso que realiza el call stack es simple, cuando se está a punto de ejecutar una función, esta es añadida al stack. Si la función llama a su vez, a otra función, es agregada sobre la anterior. Si en algún momento de la ejecución hay un error, este se imprimirá en la consola con un mensaje y el estado del call stack al momento en que ocurrió.

Javascript es un lenguaje single threaded. Esto quiere decir que durante la ejecución de un script existe un solo thread que ejecuta el código. Por lo tanto solo se cuenta con un call stack

Veamos un ejemplo:

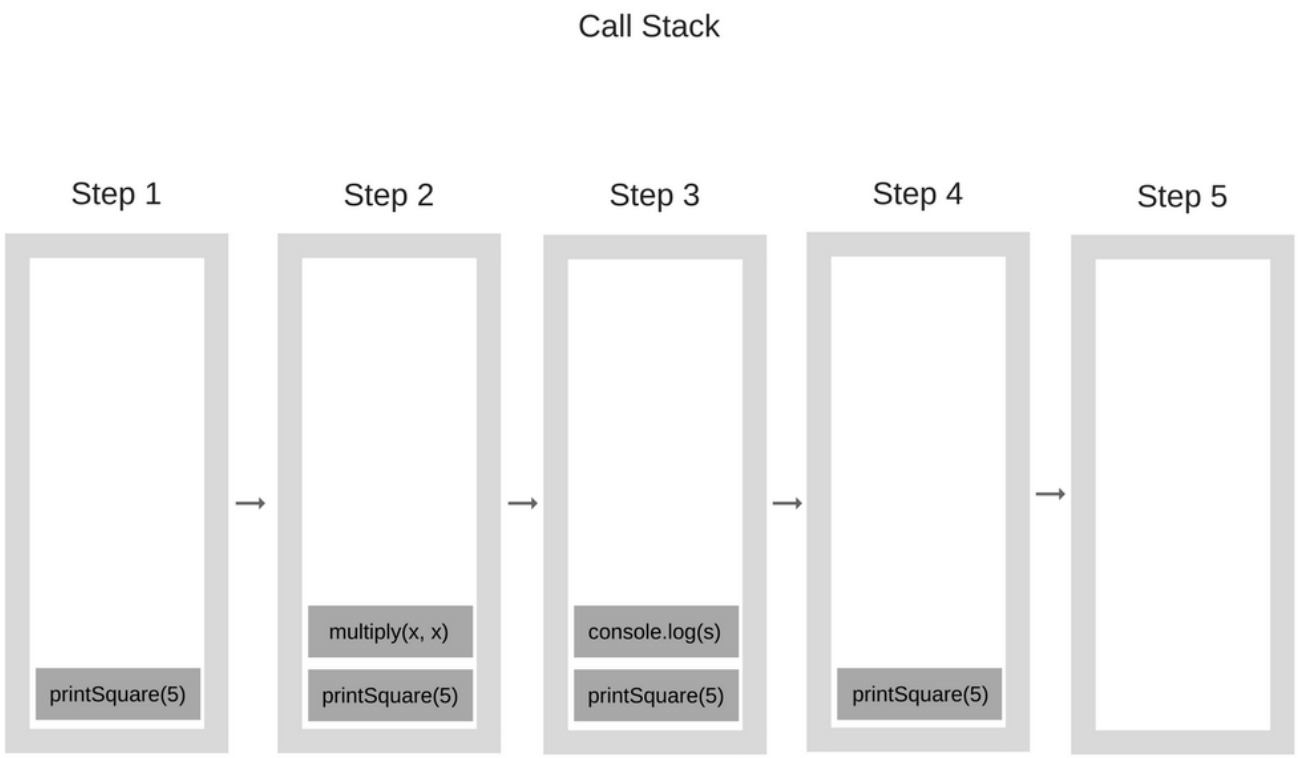
```
function multiply (x, y) {
  return x * y;
}

function printSquare (x) {
```

```
  var s = multiply(x, x);
  console.log(s);
}

printSquare(5);
```

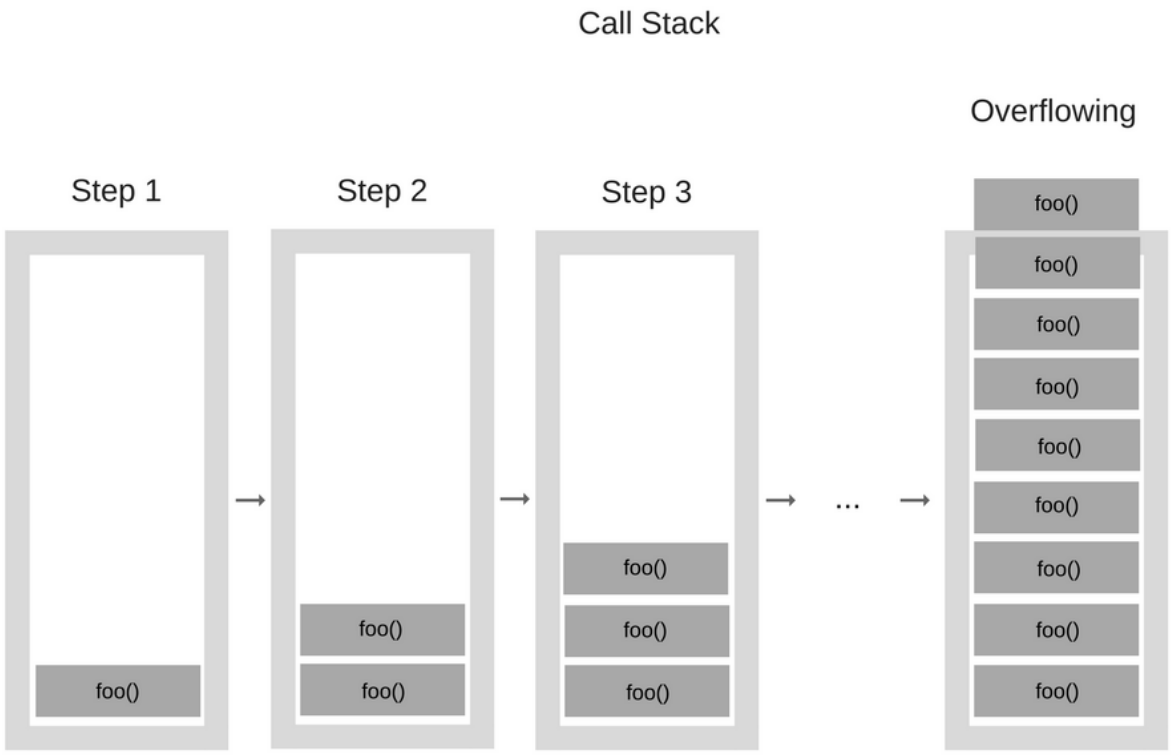
Los estados del call stack serían:



Y que pasa si tenemos una función de esta manera:

```
function foo() {
  foo();
}

foo();
```



Lo que sucedería es que en algún momento la cantidad de funciones llamadas excede el tamaño del stack , por lo que el navegador mostrará este error:

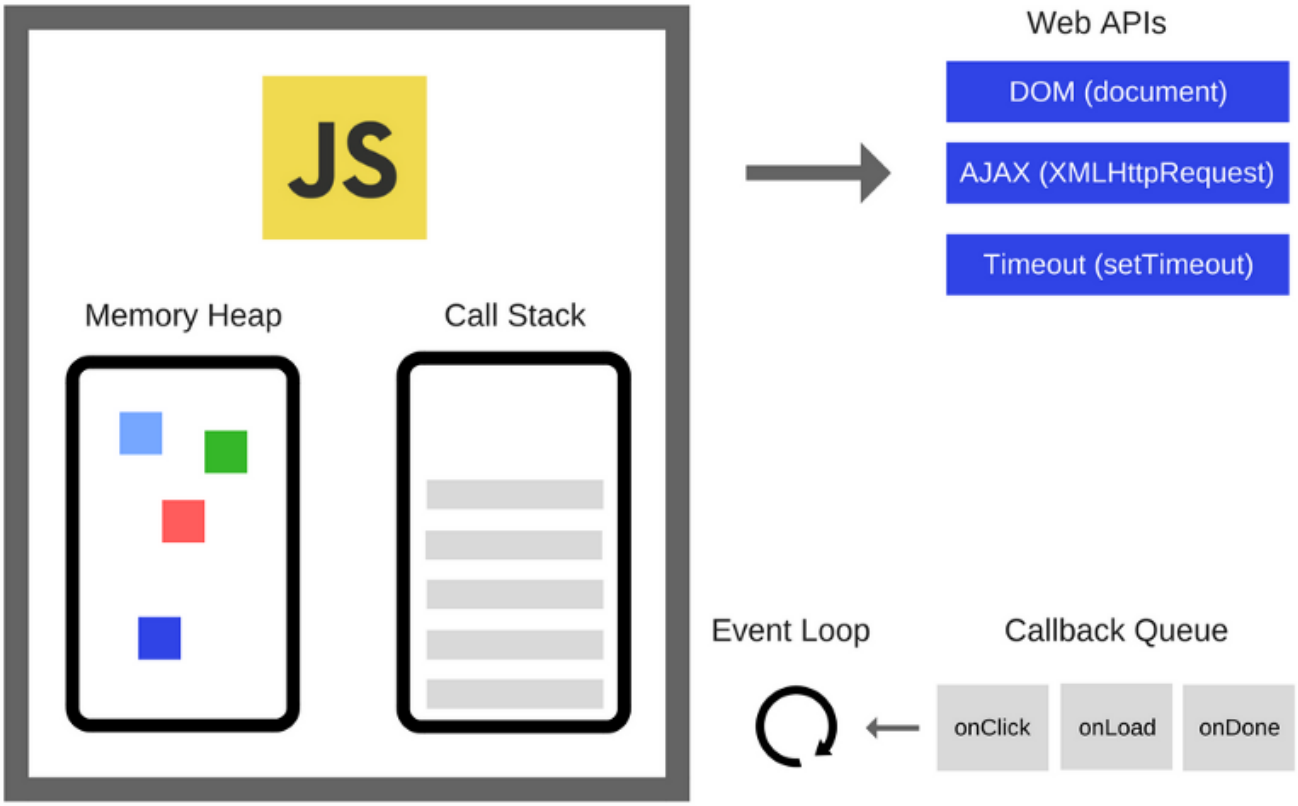
```
▶Uncaught RangeError: Maximum call stack size exceeded
```

Pero qué pasa si llamamos a un timeout o hacemos un request con AJAX a un servidor. Al ser un solo thread, hay un solo call stack y por lo tanto solo se puede ejecutar una cosa a la vez. Es decir el navegador debería congelarse, no podría hacer más nada, no podría renderizar, hasta que la llamada termine de ejecutarse. Sin embargo esto no es asi, javascript es asincrónico y no bloqueante. Esto es gracias al Event Loop.

Event Loop

Algo interesante acerca de javascript, o mejor dicho de los runtimes de javascript, es que no cuentan nativamente con cosas como setTimeout, DOM, o HTTP request. Estas son llamadas web apis, que el mismo navegador provee, pero no están dentro del runtime JS.

Por lo tanto este es el gráfico que muestra una visión más abarcativa de javascript. En este se puede ver el runtime, más las Web APIs y el callback queue del cual hablaremos más adelante.



Al haber un solo thread es importante no escribir codigo bloqueante para la UI no quede bloqueada.

Pero ¿Cómo hacemos para escribir código no bloqueante?

La solución son callbacks asincronicas. Para esto combinamos el uso de callbacks (funciones que pasamos como parámetros a otras funciones) con las WEB API's.

Por ejemplo:

```
console.log("hola");
```

```
setTimeout(function timeoutCallback() {

console.log("mundo");

}, 500);

console.log("Ubykuo, everytime, everywhere");

/*
 * Resultados:
 * => hola
 * => Ubykuo, everytime, everywhere
 * => mundo
 */
```

Como pueden ver la ejecución no se queda bloqueada en *setTimeout()* ya que imprime la instrucción que le sigue primero) ¿Pero entonces cómo es que posible que esto sea así si solo existe un solo thread? ¿ Cómo es que la ejecución continua y al mismo tiempo el setTimeout hace la cuenta regresiva para ejecutar la función pasada como callback?

Esto es porque, como mencione anteriormente, el setTimeout NO es parte del runtime. Sino que es provista por el navegador como WEB APIs (o en el caso de Node por c++ apis). Los cuales SI se ejecutan en un thread distinto.

¿Como se maneja esto con una única call stack?

Existe otra estructura donde se guardan las funciones que deben ser ejecutadas luego de cierto evento (timeout, click, mouse move), en el caso del código de ejemplo de arriba se guarda que, cuando el timeout termine se debe ejecutar la función timeoutCallback(). Tener en cuenta que cuando sucede el evento, esta estructura no es la que la ejecuta y tampoco las agrega al call stack ya que sino podría pasar que la función se ejecutará en medio de otro código. Lo que hace es enviarla a la Callback Queue.

Lo que hace el event loop es fijarse el call stack, y si está vacío (es decir no hay nada ejecutandose) envía la primera función que esté en la callback queue al call stack y comienza a ejecutarse.

Luego de terminar la cuenta regresiva del setTimeout() (que no es ejecutada en el runtime de javascript), timeoutCallback() será enviada a la callback queue. El event loop

chequeara el Call Stack, si este está vacío enviará timeoutCallback() al call stack para su ejecución.

El flujo en imágenes de todo este trabalenguas seria:

Gif creado con http://latentflip.com/loupe. Excelente herramienta para entender los conceptos que hemos visto

De esta manera se logra que el código sea no bloqueante, en vez de un setTimeout podría ser una llamada a un servidor, en donde habría que esperar que se procese nuestra solicitud y nos envíe una respuesta , el cual sería tiempo ocioso si no contáramos con callbacks asincronicas, de modo que el runtime pueda seguir con otro código. Una vez que la respuesta haya llegado del servidor y Call Stack esté vacío, se podrá procesar la respuesta (mediante la función pasada como callback) y hacer algo con ella , por ejemplo mostrarla al usuario.

¿Por que si bloqueamos el call stack la ui ya no responde más?

Esto se debe a que el navegador intenta realizar un proceso de renderizado cada cierto tiempo. Pero este no puede realizarse si hay código en el stack. El proceso de renderizado es similar a una callback asincrónica , ya que debe esperar a que el stack está vacío, es como una función más en la Callback Queue (aunque con cierta prioridad). Por lo que sí hay codigo bloqueante , el proceso de renderizado tardará más en realizarse y el usuario no podrá hacer nada, no podrá seleccionar texto, no podra ingresar texto, no podrá apretar un botón.

¿Que pasaria si a un usuario que interactuando con nuestra página le sucediera esto?

Lo más probable es que cierre el navegador y nunca más vuelva a entrar a nuestra página. No es algo que queremos que suceda.

. . .

Yapa

Algunos “hacks” interesantes para entender Javascript.

Una forma ingeniosa de ejecutar una función asincrónicamente es usar setTimeout (funcionAEjecutar, 0). Si bien el timeout es 0 , al llamar a una Web Api esta es enviada a la callback queue y será ejecutada cuando el stack esté vacío.

Otra hack interesante es para evitar el error de overflow del call stack. Una forma de permitir una cantidad absurda de llamadas recursivas, es la de envolver la función en un setTimeout. Esto permitirá que las funciones vayan al Callback Queue, evitando que se apilen en el Call Stack. Claramente esta no es una buena práctica , pero es un buen ejemplo del comportamiento de Javascript.

Autor

Lucas Botteri, Backend Programmer @ ubykuo.com.

Fuentes:

https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf
https://www.youtube.com/watch?v=8aGhZQkoFbQ

Recomiendo MUCHO ver este vídeo de Philip Roberts, del cual me base para escribir este post.

http://latentflip.com/loupe Excelente herramienta, tambien de Philip Roberts , el cual use para crear el gif.

JavaScript Performance Nodejs Async Events