Referencias, valores y administración de memoria

Ya hemos aludido al hecho de que las variables apuntan a ciertas referencias en la memoria, y también hemos cubierto brevemente cómo existen copias profundas y superficiales de los objetos. En este documento, tendremos más profundidad sobre cómo funciona realmente la asignación de memoria. Todos los conceptos que discutiremos en esta parte se dividen en un tema amplio conocido como "**administración de memoria**". La administración de la memoria, en términos simples, es cómo JavaScript asigna datos que creamos a la memoria.

Introducción

Para comprender la administración de la memoria, necesitamos comprender "montones (heaps)" y "pilas (stacks)". Estos son conceptos de memoria y están almacenados en la "memoria de acceso aleatorio" o **RAM** (Random Access Memory).

Las computadoras tienen una cantidad fija de RAM. Dado que JavaScript almacena datos en RAM, la cantidad de datos que usa JavaScript afecta la cantidad de RAM utilizada en tu computadora o servidor. Como tal, puede ser posible quedarse sin RAM si creas una aplicación JavaScript suficientemente complicada.

Entonces, si el heap y el stack se almacenan en RAM, y ambos se usan para almacenar datos de JavaScript, ¿cuál es la diferencia?

- **El stack** es un espacio de rasguño para el hilo actual de JavaScript. JavaScript es típicamente hilo-único, por lo que generalmente hay una pila por aplicación. La pila también tiene un tamaño limitado, por lo que los números en JavaScript solo pueden ser tan grandes.
- **El heap** es una tienda de memoria dinámica. Acceder a los datos desde el montón es más complicado, pero el montón no tiene un tamaño limitado. Como tal, el montón crecerá si es necesario.

JavaScript utiliza el montón para almacenar objetos y funciones. Para variables simples compuestas de números, cadenas y otros tipos primitivos, la pila se usa típicamente. La pila también almacena información sobre funciones que se llamarán.

Nota: JavaScript tiene una funcionalidad incorporada conocida como "recolección de basura" para evitar que tu aplicación se quede sin RAM. Este algoritmo utiliza una serie de

comprobaciones diferentes, como si una variable u objeto ya no se hace referencia en tu código, para borrarlo de la memoria. Esto va de alguna manera para evitar que te quedes sin memoria.

Pilas

La pila es un espacio de rastro para el hilo actual de JavaScript. Cada vez que apuntas a un tipo primitivo (primitivo, aquí, es decir, cualquier cosa que no sea un objeto) en JavaScript, se agrega a la parte superior de la pila. En el siguiente código, definimos variables. Los datos de tipo no objeto se agregan inmediatamente a la parte superior de la pila. Una representación de esto se puede ver en la figura 1.

```
const SOME_CONSTANT = 5
let myNumber = 10
let myVariable = "Some Text"
```

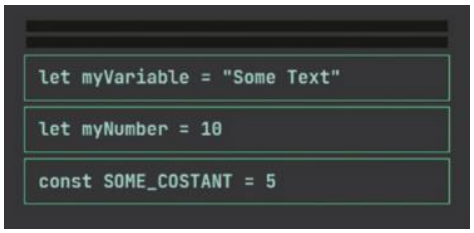

Figura 1: Cada nueva línea de código se agrega a la pila. Esto incluye funciones.

A veces, al ejecutar código o ciclos complejos, puedes ver la pila en acción. Si excedes el límite de la pila, obtendrás el error, RangeError: Maximum call stack size exceeded. Diferentes navegadores e implementaciones de JavaScript como Node.js tienen diferentes tamaños de pila, pero debes ejecutar mucho código simultáneamente para obtener este error. También puedes encontrar este error si accidentalmente ejecutas un ciclo infinito.

Si intentas reasignar una variable de tipo primitivo, también se agrega a la pila, incluso si las variables supuestamente apuntan al mismo valor. Considera el siguiente código, por ejemplo:

```
let myNumber = 5
let newNumber = myNumber
```

Aunque parece que ambas variables deberían apuntar al mismo valor subyacente (ese valor es 5), JavaScript agrega ambas referencias a la pila como entradas individuales con diferentes valores por completo.

Lo que eso significa en la práctica es que para cualquier valor primitivo o no objeto en JavaScript, siempre se hace una copia profunda. Las variables no apuntarán al mismo valor subyacente en la memoria, sino que aparecerán como nuevas copias de datos, cada una actuando independientemente entre sí, como se muestra en la figura 2.

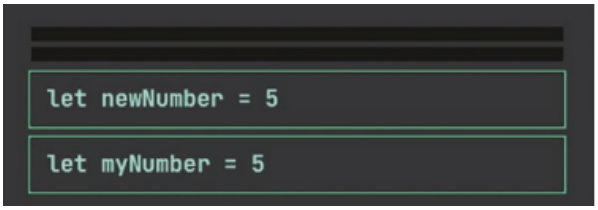

Figura 2: Cada nueva variable de tipo no objeto aparecerá como una nueva referencia y valor en la pila.

El ciclo del evento

Nota: En esta sección entraremos brevemente en cómo funcionan las APIs para mostrarte cómo funciona el ciclo de eventos. Si bien no cubriremos qué son las APIs y cómo usarlas en detalle aquí, entraremos en esto en futuros cursos. Trabaja por tu cuenta si estás interesado en aprender más sobre APIs.

El código JavaScript solo tiene una pila en los navegadores, por lo que comúnmente se describe como un solo subproceso. Sin embargo, JavaScript puede actuar como si tuviera múltiples hilos a la vez en el front end a través de algo conocido como APIs web.

Utilizamos algo llamado API o interfaces de programación de aplicaciones (Application Programming Interfaces) para externalizar parte de nuestro cálculo en otro lugar y esperar la respuesta. Por lo general, las APIs existen en un servidor. Por ejemplo, podrías utilizar una API para recuperar una lista de artículos en tu sitio web. Entonces podríamos usar estos datos en nuestro código.

“API” suena complicado, pero es solo una URL que podemos alcanzar, que eventualmente nos enviará una respuesta desde dentro de nuestro código. Por ejemplo, podemos alcanzar la URL https://some-website.org/api/articles para recuperar artículos del sitio web. Luego recibiríamos una respuesta de la API con todos los artículos. En un tema futuro, profundizaremos en cómo funcionan las APIs , depende del avance del curso.

Cuando ejecutamos una API, el servidor hará algún cálculo en nuestro nombre, pero nuestro código JavaScript continuará ejecutándose. Dado que el código API y JavaScript se procesan al mismo tiempo, esto nos da una forma de crear múltiples hilos.

Los navegadores tienen algunas APIs incorporadas que se pueden llamar directamente desde tu código. Estos se conocen como API web, y generalmente ofrecen una interfaz entre el código en tu navegador y el sistema operativo en sí. Un ejemplo de una API web es la función global, SetTimeOut, que nos permite ejecutar código después de un cierto número de segundos:

```
let myNumber = 5
setTimeout(function() {
    console.log("Hello World")
}, 1000)
```

SetTimeout acepta dos argumentos: una función de devolución de llamada y un tiempo en milisegundos, que son procesados por la API web. La mayoría de las APIs web aceptan una función de devolución de llamada, que se ejecuta en la pila de la API web. Esto significa que SetTimeout se elimina inmediatamente de la pila principal y solo vuelve a la pila principal una vez que se haya procesado.

Esto significa que las APIs web se ejecutan en paralelo a la pila principal de JavaScript. Cuando se termina la devolución de llamada, el resultado se agrega nuevamente a la pila principal, y esto está mediado por algo llamado evento loop, que decide cuándo agregar tareas al hilo principal. Puedes ver una ilustración de cómo funciona esto en la figura 3:

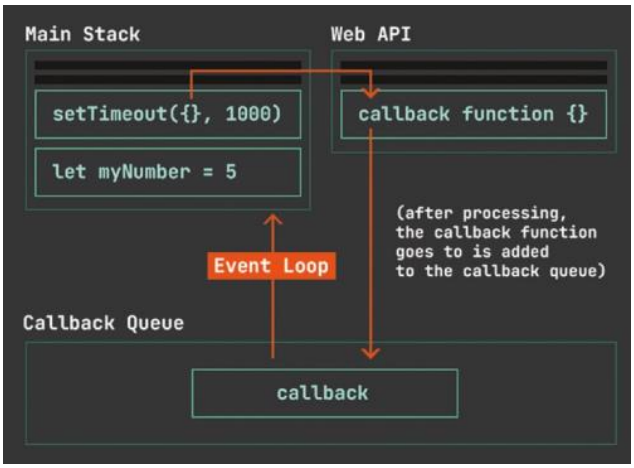

Figura 3: Las APIs web se pueden usar para crear concurrencia en JavaScript. Las APIs web usan funciones de devolución de llamada, que son procesadas por una pila de API web, separadas de otras JavaScript. Cuando se completa el procesamiento, la devolución de llamada se agrega a la cola de devolución de llamada, que actúa al igual que la pila, ya que las nuevas devoluciones de llamada se agregan a la parte superior y el procesamiento comienza desde la parte inferior. El ciclo de eventos luego media la devolución de llamada que se agrega a la pila principal, donde se puede procesar como cualquier otro elemento de la pila.

Nota: Las funciones de API web que tardan mucho tiempo en ejecutarse, como SetTimeout, se agregan al ciclo de eventos como “macrotasks”. Algunas funciones de API web más rápidas se agregan como “microtasks”. Las microtasks tienen prioridad cuando se agregan de nuevo a la pila principal. ¡Si una API web genera una macro o microtask depende de cuánto tiempo lleva ejecutarse!

El heap

Cuando revisamos el tema de los objetos, cubrimos cómo podemos copiar objetos haciendo copias “profundas” y “superficiales” de ellos. La razón por la que tenemos copias profundas y superficiales en JavaScript es por cómo se almacenan los objetos en el heap.

Si bien los tipos no objeto se almacenan solo en la pila, los objetos se arrojan al heap, que es una forma más dinámica de memoria sin límite. Esto significa que los objetos grandes nunca excederán el límite de la pila.

Considera el siguiente objeto. Primero, definimos un nuevo objeto, y luego establecemos otra variable para apuntarlo:

```
let userOne = { name: "John Schmidt" }
let userTwo = userOne
```

Mientras que en ejemplos anteriores usando la pila, este tipo de código conduciría a dos nuevas entradas, este código hace algo ligeramente diferente. Aquí, el objeto se almacena en el heap, y la pila solo se refiere a la referencia del heap. El código anterior produciría algo un parecido a lo que se muestra en la figura 4.

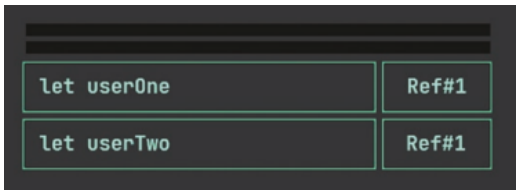

Figura 4: Los objetos se almacenan en el heap, y la pila se refiere a la referencia de ese objeto. Los códigos de referencia utilizados aquí son ilustrativos.

Dado que las variables ahora se refieren al mismo objeto, actualizar uno puede tener un efecto en el otro, lo que puede causar confusión al actualizar los arreglos que crees que son copias profundas, pero en realidad son copias superficiales.

Nota sobre los tipos de objetos: Dado que las funciones y los arreglos también son de tipo "objeto", ¡ellos también se almacenan en el heap!

Objeto y igualdad de referencia

La complejidad final que JavaScript introduce mediante el uso de este método para almacenar objetos y no objetos es hacer con la igualdad. JavaScript en realidad tiene muchas dificultades para comparar los valores de dos objetos diferentes, y básicamente se reduce a pilas y heaps. Para entender por qué, considera el siguiente código:

```
let myNumber = 5
let newNumber = 5
let newObject = { name: "John Schmidt" }
let cloneObject = { name: "John Schmidt" }
let additionalObject = newObject
```

Acabamos de definir un heap de variables, y en la pila, se parecería un poco a lo que se muestra en la figura 5.

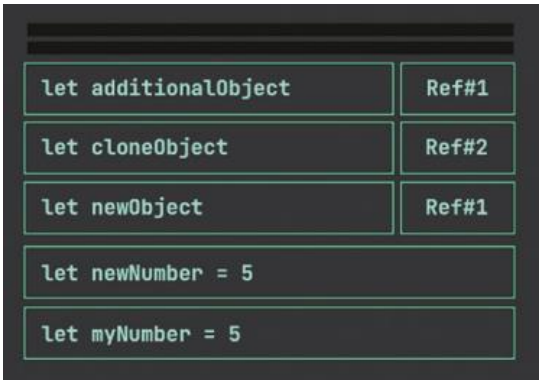

Figura 5: Mientras que los no objetos se realizan en la pila de manera normal, un nuevo objeto crea una nueva referencia. Aunque tanto cloneObject como newObject tienen el mismo "valor" subyacente, sus referencias aún difieren.

Entonces, newNumber y myNumber tienen el mismo valor y nos darán un valor "verdadero" si intentamos probar su igualdad con el signo de triple iguales:

```
let myNumber = 5
let newNumber = 5
console.log(myNumber === newNumber) // TRUE
```

Del mismo modo, additionalObject y newObject se refieren a la misma referencia y también probarán true para igualdad:

```
let newObject = { name: "John Schmidt" }
let additionalObject = newObject
console.log(newObject === additionalObject) // TRUE
```

La complejidad aparece cuando comparamos cloneObject y newObject. Aunque cloneObject tiene el mismo valor subyacente que newobject, JavaScript comparará la igualdad de las dos referencias, no el valor subyacente en sí mismo:

```
let newObject = { name: "John Schmidt" }
let cloneObject = { name: "John Schmidt" }
console.log(newObject === cloneObject) // False
```

Este es un punto de confusión común para la mayoría de las personas que aprenden JavaScript porque es contradictorio. Cuando entiendes heaps y pilas, comienza a tener sentido. newObject y cloneObject no son iguales, ya que ambos tienen diferentes referencias en la pila.

Comparación de la igualdad de objetos

Entonces, ¿cómo se compara la igualdad de objetos para los objetos con diferentes referencias en JavaScript? Bueno, no es fácil, y no hay una forma incorporada de hacerlo. En cambio, tienes las siguientes opciones:

1. Puedes escribir tu propia función para comparar cada llave y valor en un objeto individualmente o encontrar un buen ejemplo en línea.
2. En Node.js, que no cubriremos con mucho detalle en estas notas sino más adelante, puedeS usar la función incorporada 'assert.deepStrictEqual(obj1, obj2)' para comparar dos objetos.
3. Finalmente, aunque no se recomienda, también puedes convertir ambos objetos en cadenas usando JSON.stringify() para comparar sus valores. JSON.stringify convertirá cualquier objeto en una cadena, que se almacenará en la pila. Entonces, si ambas cadenas son las mismas, puedes asumir la igualdad. Esto no se recomienda ya que este método JSON también elimina algunos tipos de datos de objetos, como funciones. Por lo tanto, no puedes estar seguro de que siempre pruebes la igualdad, lo que lo hace bastante poco confiable.

Si estás utilizando Node.js, tu mejor opción es la opción 2. De lo contrario, se recomienda más que escribas tu propia función (o encuentres una en línea), como se describe en la opción 1.

Resumen

En este documento, hemos cubierto muchos conceptos básicos sobre cómo funciona la administración de la memoria. Hemos visto cómo funcionan las pilas y los heaps en JavaScript. También hemos revisado brevemente las APIs web y cómo simulan el comportamiento multiproceso en JavaScript. Hemos cubierto brevemente las APIs y las APIs web, aunque a lo mejor se revisarán posteriormente.

Temas como la forma en que JavaScript maneja la igualdad de objetos puede parecer confuso y contradictorio hasta que entendemos cómo los objetos se almacenan y comparan de manera diferente a otros tipos de datos. Como tal, una buena comprensión de la administración de la memoria puede ayudar a explicar algunas de las peculiaridades que experimentamos al desarrollar aplicaciones en JavaScript.