Concurrency Event loop

4th Jun, 2020 12:00 PM

Albert Sebastian

Week # 7 | Day # 4

Deep dive into JavaScript and Asynchronous Programming

The Event loop and Concurrency Model:

Before we actually go over asynchronous programming lets go over some core JavaScript concepts first.

Source: MDN web docs

Stack

When you call a function in JavaScript all the instructions within that function get loaded into a Stack. Each function has a logical **frame** within the call Stack. Javascript then executes the instructions in each frame by popping them from the stack.

Example From MDN:

```
function foo(b) {
  var a = 10;
  return a + b + 11;
}

function bar(x) {
  var y = 3;
  return foo(x * y);
}

console.log(bar(7)); //returns 42
```

"When calling bar, a first frame is created containing bar's arguments and local variables. When bar calls foo, a second frame is created and pushed on top of the first one containing foo's arguments and local variables. When foo returns, the top frame element is popped out of the stack (leaving only bar's call frame). When bar returns, the stack is empty."

Heap

On the other hand, objects in JavaScript are stored in what is called the **Heap**. This is an unstructured portion of memory which object data lives.

Task/Message Queue

JavaScript also uses a data structure known as a Queue to store tasks/messages.

Tasks/messages are added to the queue anytime an event occurs and there is a listener attached to that event.

Functions like `setTimeout` and `setInterval` which accept callbacks also add tasks/messages to the queue.

The Event loop

JavaScript has one thread called which runs a loop that looks for new messages/tasks on the message queue and pushes them onto the call stack to be executed.

Think of it sort of like this:

```
while (queue.waitForMessage()) {
"video":
["/lecture/536/material/65658fde58ab3c2b6e5132a39fae7cb9/concurrency.mp4"
]  queue.processNextMessage();
}
```

However, the event loop gives priority to the frames currently on the call stack pushes a new message from the Queue onto the stack after all the frames in the stack have been executed and the call stack is empty.

Run to completion:

It is also important to note that each message on the queue is completely processed before another other message is processed.

Great Talk on Event Loop

Call Stack and Queue Visualizer

Synchronous Programming

Most programs you write often tend to be **synchronous**. This means that actions in code happen one at a time.

Take the function `sumUntil` that is given below. Any code that needs to run after line 2 can only run after `sumUntil` has finished all its work and returned some value.

```
let a = 10; //1
console.log(sumUntil(a), new Date().getTime()); //2
console.log("The function is done!", new Date().getTime());

function sumUntil(y){
    let arr = [];
    for(let i = 0; i < y; i++){
        for(let j = 0; j < y; j ++){
            arr.push([i, j])
        }
    }
    return arr;
}
```

This means that while waiting for that function to return its value the rest of your program is still waiting and not able to any useful work in the mean time.

This can be a huge problem with tasks that take a very long time or an **indeterminate** amount of time.

For example HTTP requests could take any amount of time to receive a response from the server. There is even a chance you won't even get a response!

However, you don't want your entire WebPage or Javascript program waiting for that request to complete. You still want users to be able to use your Application and perform I/O (input/output) actions!

Moreover, if you entire application must wait for this request it will freeze or hang, this is known as blocking and is really undesirable.

Threads:

A solution to this problem that is popular in many languages is using multiple **threads**.

Imagine thread as small part of your overall program that runs in parallel with your main program. For example a specific function could be run on a separate thread.

This means that while your main program is always ready for user inputs and other work, you threads can do other work like HTTP requests in the background.

Most browsers implement this using Web Workers API. NodeJS also has a implementation with service workers.

However, multi-threaded applications can get really complex to understand and implement as a beginner programmer.

Asynchronous Programming

Source : An Introduction to Asynchronous Programming in Python

https://medium.com/velotio-perspectives/an-introduction-to-asynchronous-programming-in-python-af0189a88bbb

The alternative to using threads that is heavily used in JavaScript is Asynchronous programming.

In asynchronous programming, the main thread, in our case the event loop finishes all its primary work without being blocked.

In synchronous programming is that easy task that is started must be completed before another task can start.

On the other hand, in asynchronous programming, many tasks can start before other tasks are completed and when each task is done, it notifies the main thread. The above diagram illustrates this well.

Asynchronous Programming in JavaScript:

We have already done lots of asynchronous programming in JavaScript using methods that accept callbacks.

Since callbacks are pushed to the callback queue rather than the call stack, they will only begin execution after all the frames in the callstack have completed execution.

Example of Synchronous vs Asynchronous:

Synchronous:

```
let j = 0;
while(j < 3){
    console.log(j);
    j++;
}

let i = 0;
while(i < 3){
    console.log(i);
    i++;
}

console.log("Welcome to Masai.");
```

Asynchronous:

```
setTimeout(function() {
    let j = 0;
    while(j < 3){
        console.log(i);
        j++;
    }
}, 0)


let i = 0;
while(i < 3){
    console.log(i);
    i++;
}

console.log("Welcome to Masai.");
```

4th Jun, 2020 12:26 PM