你真的了解 JavaScript 异步么？

JavaScript 是一种单线程编程语言，这也就意味着在同一时刻它只能做一件事情。更明确地说，也就是 JavaScript 引擎在同一时刻单一线程内只能处理一个语句。

使用单线程语言写代码是一件很轻松愉悦的事情，因为你不需要担心因为并发而导致的问题，这也就意味着你不能执行长操作，例如：不阻塞主线程的网络访问。

可以想象一下浏览器从接口获取数据的过程，服务器需要花费一些时间去处理这个请求，那么此时基于单线程语言的情况，我们可以知道，浏览器的主线程正在被阻塞，并且导致页面无响应。然而实时并非如此，为什么呢，接着看。

这里就该 JavaScript 异步处理来解释解释了。使用异步的 JavaScript 例如：（callbacks，promises，async/await 等），就可以执行网络请求，而不需要阻塞主线程。

同步的 JavaScript 时如何工作的呢？

在我们潜心研究 JavaScript 异步之前，让我们先理解一下 JavaScript 同步代码是如何再引擎内部执行的。例如：

```
const second = () => {
  console.log('Hello there!');
}
const first = () => {
  console.log('Hi there!');
  second();
  console.log('The End');
}
first();
```

为了理解上面代码是如何再引擎内部执行的，我们必须理解一下执行上下问的概念以及调用栈。

执行上下文 context

执行上下文是个执行环境的一个抽象，在这个环境中 JavaScript 代码被执行和评估。任何 JavaScript 代码都在其内部的执行上下文环境上被执行。

函数代码执行在内部的函数执行上下文中，全局代码执行在内的全局上下问化境中。每一个函数都有它自己的执行上下文环境。

调用栈 Call Stack

调用栈顾名思义，就是一个栈，一种后进先出的数据结构。它常被用于存储代码执行期间创建的执行上下文环境。

JavaScript 是单线程编程语言，所以它只有一个栈。调用栈是这种后进先出（LIFO）的数据结构，这也就意为着栈内元素的删除或是添加只能是从栈顶开始。

让我们回到上面的代码片段部分，试着去理解一下这段代码是如何在 JavaScript 引擎内部执行的。

```
const second = () => {
  console.log('Hello there!');
}
const first = () => {
  console.log('Hi there!');
  second();
  console.log('The End');
}
first();
```

到底发生了什么？

当代码被执行的时候，全局执行上下文环境被创建（图中的 main()），并且被 push 到整个栈的栈顶。当遇到 first() 的时候又将它 push 到栈顶。

接下来，console.log('Hi there!') 被 push 到栈顶，当它执行完成后，它就会被出栈。之后这里又调用了 second()，于是函数 second() 被推入栈顶。

console.log('Hello there!') 被推入栈顶，执行完成后被出栈（pop）。这时 second() 函数页执行完成了，所以它也被出栈。

console.log('The End') 被推入栈中，当执行完成后，再次被出栈。接下来 first() 函数也执行完成被出栈。

这时程序完成了它的执行。所以全局执行上下文（main()）也被从栈顶移除。

异步的 JavaScript 时如何工作的呢？

现在我们对调用栈有了基本的认识，以及 JavaScript 同步代码是如何工作的。让我们回到 JavaScript 异步中。

什么是阻塞？

假设我们正在处理一张图片，或是一个网络请求，以同步的方式。例如：

```
const processImage = (image) => {
  /**
  * 正在对图片进行一下处理
  **/
  console.log('Image processed');
}
const networkRequest = (url) => {
  /**
  * 请求网络资源
  **/
  return someData;
}
const greeting = () => {
  console.log('Hello World');
}
processImage(logo.jpg);
networkRequest('www.somerandomurl.com');
greeting();
```

处理图片和发起网络请求都需要花费时间。于是当 processImage() 函数被调用的时候，它将要花费的时间取决于图片的大小。

当 processImage() 函数执行完成时，它会被从栈中移除。随后 networkRequest() 被调用，并且 push 到栈中。同样它也需要花费一些时间去完成执行操作。

最后，当 networkRequest() 函数完成后，greeting() 函数被调用，被推入栈中。因为它包含了 console.log 语句，二通常 console.log 执行非常快，所以 greeting() 函数会立即执行完成，并且被返回。

可以看到，我们必须去花费时间等待直到 networkRequest() 或是 processImage() 完成 才能执行接下来的任务。也就是说这些函数正在阻塞调用栈或者说主线程。所以我们不能执行任何的操作再上面提到的函数执行完成前，这并不是我们想要的。

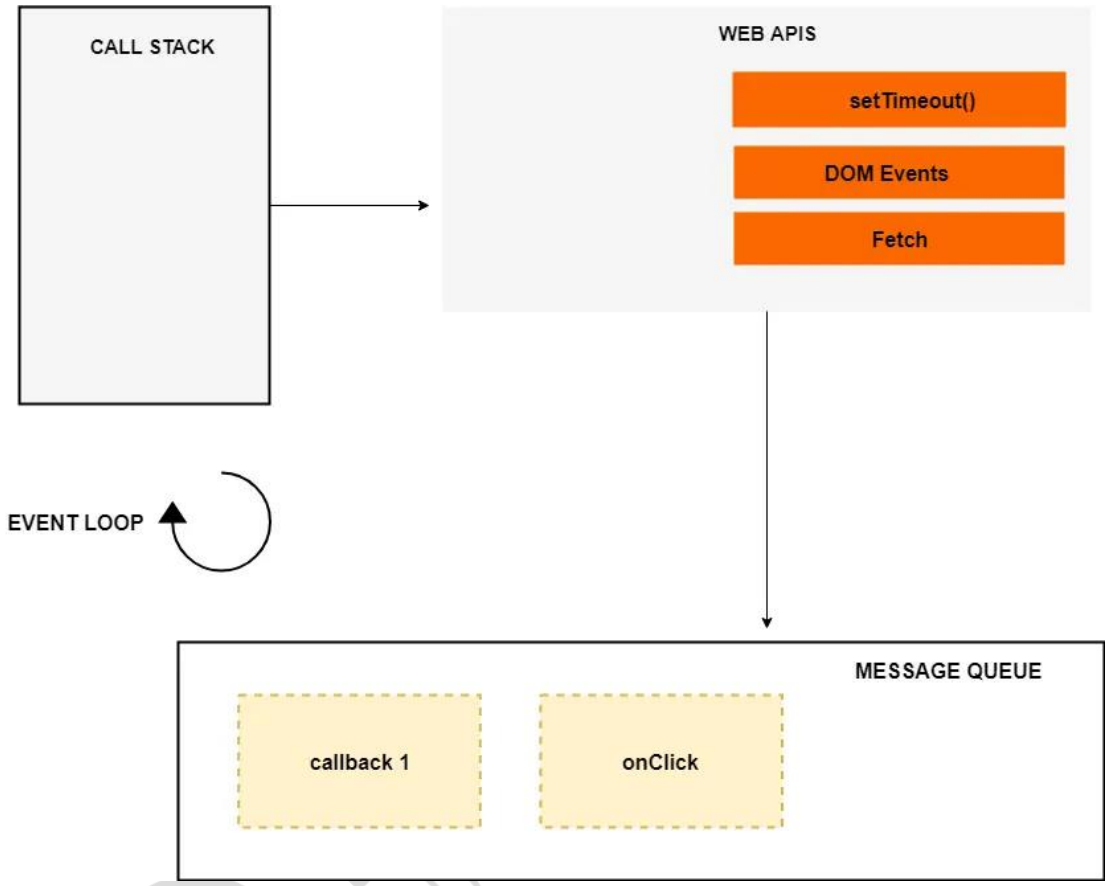
那么解决方案是什么呢？

最简单的方法是异步回调。使用 异步回调可以使我们的代码变成非阻塞代码。例如：

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
};
console.log('Hello World');
networkRequest();
```

这里有一个 setTimeout 方法用来模拟网路请求。请牢记 setTimeout 不是 JavaScript 引擎的一部分，它是我们熟知的 Web APIs (浏览器中) 还有 c/c++ APIs (nodeJs 中)。

为了理解这部分的代码是如何执行的，我们必须清楚一些概念，例如：事件循环机制，回调队列（又称之为：任务队列，或者消息队列）。



事件循环机制， **Web APIs 还有消息队列/任务队列** 他们都不是属于 JavaScript 引擎，它们是浏览器运行环境的一部分 或者 NodeJs 运行环境的一部分。在 NodeJs 中，这些 Web APIs 被替换成了 C/C++ APIs。

现在我们返回到上面代码中，它是如何以异步的方式执行的。

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
};
console.log('Hello World');
networkRequest();
console.log('The End');
```

Event Loop
当上面例子中的代码被加载到浏览器时，console.log('Hello World') 最先被 push 到栈中，和之前一样在完成后出栈。接下来是 networkRequest() 的调用，同样它被 push 到栈顶。

接下来 setTimeout() 函数被执行，所以它也会被 push 到栈中。
setTimeout() 函数接受了两和参数，1）callbaclk 和 2)毫秒数

setTimeout() 方法在 Web APIs 环境中开启了一个两秒钟的定时器。这时 setTimeout() 已经完成，随即被出栈。之后 console.log('The End') 被 push 到栈中，在它执行完后出栈 pop。

与此同时定时器已经到期，这时 callback 被 push 到消息队列。但是回调并不是立即执行，这里就是事件循环机制发送作用的地方。

事件循环 Event Loop

事件循环机制的作用就是查看调用栈，并且判断调用栈是否为空。如果调用栈为空，它就会去查看消息队列中是否存被挂起的回调正在等待被执行。

在这个例子中，消息队列只包含一个回调，于是当检测到调用栈为空的时候，就将这个回调 push 都执行栈中。

在 console.log('Async Code') 被 push 到栈顶之后，执行然后出栈。这时回调已经从栈中移除，并且程序执行结束。

DOM Events

DOM 事件的回调也会被放置在消息队列中处理，例如键盘事件或是鼠标事件等。例子：

```
document.querySelector('.btn').addEventListener('click',(event) => {
  console.log('Button Clicked');
});
```

在 DOM 事件中，位于 Web APIs 环境中的事件监听器等待着一个确定的事件发生比如：click ，于是当事件被触发生的时候这个回调函数就会被放置到消息队列中，等待调用栈为空的时候被执行。

当事件循环机制检查到调用栈为空的时候，这个事件的回调就会被 push 到栈中被执行，

我们已经学习到了异步以及 DOM 事件的回调函数是如何被执行。就是都被保存在消息队列中等待栈空时被执行。

ES6 中的微任务队列

ES6 引入微任务队列的概念，在 JavaScript 中微任务队列最常见的应用就是 Promise 。微任务队列和消息队列（宏任务队列）之间最大的不同之处就在于

微任务队列用于更高的优先级，这也就意味着 Promise 中的回调会被添加至微任务队列中，并且会在消息队列中的回调之前被添加到执行栈中。

例如：

```
console.log('Script start');
setTimeout(() => {
  console.log('setTimeout');
}, 0);
new Promise((resolve, reject) => {
    resolve('Promise resolved');
  }).then(res => console.log(res))
    .catch(err => console.log(err));
console.log('Script End');

// 输出结果
/*
        Script start
  Script End
  Promise resolved
  setTimeout
*/
```

我们看到 Promise 执行在 setTimeout 函数之前，因为 Promise 的响应被存储在内部的微任务队列中，它相对于消息队列来说拥有更高的优先级。

我们在看看另外一个例子：

```
console.log('Script start');
setTimeout(() => {
  console.log('setTimeout 1');
}, 0);
setTimeout(() => {
  console.log('setTimeout 2');
}, 0);
new Promise((resolve, reject) => {
    resolve('Promise 1 resolved');
  }).then(res => console.log(res))
    .catch(err => console.log(err));
new Promise((resolve, reject) => {
    resolve('Promise 2 resolved');
  }).then(res => console.log(res))
    .catch(err => console.log(err));
console.log('Script End');
```

```
// 输出结果
/*
        Script start
  Script End
  Promise 1 resolved
  Promise 2 resolved
  setTimeout 1
  setTimeout 2

*/
```

上面例子中，两个 Promise 都执行在 setTimeout 中的回调之前。因为事件循环机制优先处理微任务队列中的任务。

当引擎处理微任务队列中的任务的时候，如果有其他的 Promise 变为 resolved，它也将会被添加到微任务队列的末尾，并且也将在消息队列之前执行。无论这个微任务队列执行多久，消息队列（宏任务队列）都需要等待。

例如：

```
console.log('Script start');
setTimeout(() => {
  console.log('setTimeout');
}, 0);
new Promise((resolve, reject) => {
    resolve('Promise 1 resolved');
  }).then(res => console.log(res));
new Promise((resolve, reject) => {
  resolve('Promise 2 resolved');
  }).then(res => {
       console.log(res);
       return new Promise((resolve, reject) => {
         resolve('Promise 3 resolved');
       })
     }).then(res => console.log(res));
console.log('Script End');

// 输出结果
/*
        Script start
  Script End
  Promise 1 resolved
```

```
  Promise 2 resolved
  Promise 3 resolved
  setTimeout
*/
```

所有的微任务队列都会在消息队列执行之前执行。在执行任何消息队列中的任务之前，首先微任务队列必须为空。

结论

到这里我们已经掌握了 JavaScript 中的异步任务时如何工作的，还有一些其他的概念，比如 **调用栈，事件循环，消息队列/宏任务队列，微任务队列**，它们共同组成了 JavaScript 的运行时环境。尽管这些概念不是必须的，但是知道它们也是很有用处的。