Tìm hiểu về Synchronous và Asynchronous trong Javascript

June 17, 2019


```
const x = 2;

if(x == true){
    console.log('x == true');
}
else if (x == false) {
    console.log('x == false');
}
else if(x){
    console.log('x is truthy?!');
}

// x is truthy?!
```

Giới thiệu

JavaScript là ngôn ngữ lập trình Single-thread (đơn luồng), có nghĩa là tại 1 thời điểm chỉ có thể xử lý 1 lệnh. Nó đơn giản khi viết code vì không phải lo về các vấn đề khi chạy song song (Ví dụ luồng chính phải đợi các luồng con trả về kết quả để tổng kết).

Giờ thì bạn hãy tưởng tượng client gửi request lấy dữ liệu từ một API. Ở đây có thể xảy trường hợp server có thể mất thời gian để xử lý request (Hoặc tệ hơn là server không trả về kết quả) nếu ở đây đợi đến khi server trả về kết quả mới chạy tiếp thì nó sẽ khiến trang web không phản hồi.

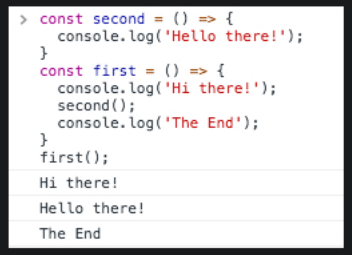
Vậy Javascript mới tạo ra Asynchronous để giúp chúng ta làm việc này (như callbacks, Promises, async/await) giúp luồng chạy của web không bị chặn lại khi đợi request. Thôi không dài dòng nữa bây giờ chúng ta hãy bắt đầu về Synchronous và Asynchronous nào.

Javascript Synchronous hoạt động như thế nào?

Bây giờ chúng ta có 1 đoạn code như sau:

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

Các bạn hãy dự đoán kết quả sẽ in ra như thế nào? Vâng và đây là kết quả, các bạn cùng xem nhé:



Javascript thực thi lệnh theo thứ tự main -> first() -> console.log('Hi there!') -> second() -> console.log('Hello there!')- > (Kết thúc second) -> console.log('The End') -> (Kết thúc first) -> (Kết thúc main). Với main ở đây là luồng chạy của chương trình. Và để chương trình chạy được như thế thì cần đến cái gọi là **call stack**.

> **Sử Dụng Destructuring Assignment Trong Javascript**

Call stack: Đúng như tên gọi nó là ngăn xếp chứa các lệnh được thực thi. Với nguyên tắc LIFO (Last In First Out – Vào sau thì ra trước). Và vì Javascript là ngôn ngữ đơn luồng nên chỉ có 1 **call stack** này để thực thi lệnh. Chúng ta có thể mô tả lại quá trình chạy lệnh trên theo sơ đồ sau:

Vậy đấy chính là cách mà Javascript Synchronous thực hiện

Javascript Asynchronous hoạt động như thế nào?

Chúng ta có đoạn code sau để minh họa cho Javascript Asynchronous:

Chúng ta có đoạn code sau để minh họa cho Javascript Asynchronous.

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

Mình xin giải thích. Ở đây **networkRequest** có sử dụng **setTimeout** để giả lập cho hành động gửi 1 request đến API. Và đây là kết quả

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

```
Hello World                                    VM7561:6
The End                                        VM7561:8
undefined
Async Code                                     VM7561:3
```

Để giải thích cho javascript asynchronous chúng ta cần biết thêm về Event loop, web APIs và Message queue. Mình xin lưu ý là đây không phải là của javascript mà nó là 1 phần của trình biên dịch javascript của browser.

> **Javascript - Một số mẹo giúp chuyển đổi String sang Number**

1. **Event loop**: Nhiệm vụ của Event loop là xem trong **call stack** có trống hay không. Nếu như nó trống thì sẽ xem tiếp trong **message queue** có callback nào đang đợi không. Nếu có thì nó sẽ đẩy callback đó vào **call stack**.
2. **Message queue**: Hay còn gọi là Task queue, Callback queue. Đây là hàng đợi của các task khi sử dụng Web APIs. Và các task sẽ được lấy ra theo quy tắc FIFO (First In First Out – Vào trước ra trước) và sẽ được event loop đưa lên **call stack** để thực thi.
3. **Web APIs**: Bao gồm setTimeout, DOM Events (Ví dụ sự kiện click button, ...), ...

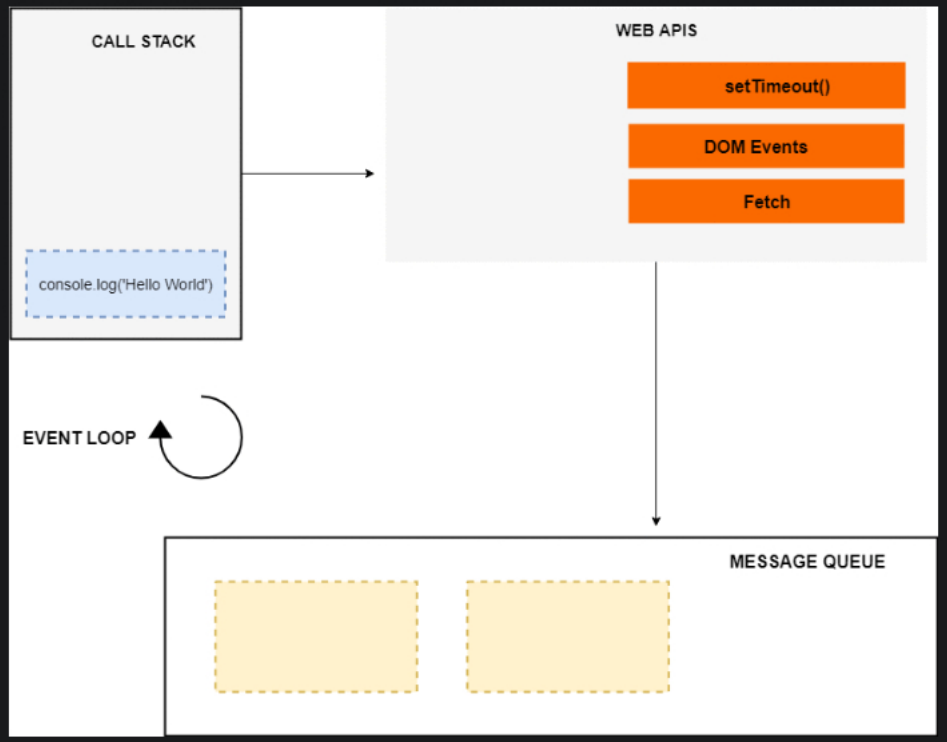
Sau khi đã biết được những khái niệm trên mình xin giải thích lại khối code ở trên (Sẽ rất khó hiểu đây

- Đầu tiên `console.log('Hello World')` sẽ được đưa vào **call stack** để thực thi. Sau khi thực thi

xong sẽ in log ra console. Và đẩy ra khỏi stack

- Tiếp đến `networkRequest()` được đưa vào stack. Và trong `networkRequest` có sử dụng `setTimeout` nên `setTimeout` được đưa vào stack. Nhưng ở đây callback trong setTimeout cần thời gian sau 2 giây mới thực hiện. Sau khi thực thi xong lần lược lấy `setTimeout` và `networkRequest` ra khỏi stack. Sau 2 giây callback của setTimeout được đưa vào **message queue** (Việc chờ và lấy ra khỏi stack thực hiện đồng thời).
- `console.log('The End')` được đưa lên stack để thực thi. Và lại được đưa ra khỏi stack.
- Event loop thấy stack trống nên đưa callback trong queue lên stack để thực thi code. `console.log('Async Code')` được thực thi và in ra console. Sau đó `console.log('Async Code')` được lấy khỏi stack. Vậy là đã kết thúc quá trình chạy đoạn code trên

Thật là khó hiểu đúng không? Vậy bạn hãy xem hình mô tả dưới đây



Tới đây nếu bạn đã thực sự hiểu thì xin chúc mừng bạn. Còn nếu bạn vẫn chưa hiểu thì hãy xem lại ví dụ ở link này Click vào đây

P/s: Có thể chỉnh tốc độ chậm lại để dễ quan sát hơn.

ES6 Job Queue/ Micro-Task queue

ES6 đã giới thiệu khái niệm **job queue/micro-task queue** được Promise sử dụng. Sự khác biệt

giữa **message queue** và **job queue** là **job queue** có mức độ ưu tiên cao hơn **message queue**, điều đó có nghĩa là các công việc trong **job queue/micro-task queue**sẽ được thực hiện trước **message queue**.

Chúng ta hãy xem ví dụ dưới đây:

```
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');
```

Bạn dự đoán kết quả thử rồi hãy xem kết quả nhé. Và đây là kết quả

```
Script start
Script End
Promise resolved
setTimeout
```

Chúng ta thấy rằng Promise được thực hiện trước setTimeout vì callback ở Promise được lưu bên trong **job queue/micro-task queue** có mức độ ưu tiên cao hơn **message queue**.

Ví dụ tiếp theo

```
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');
```

Kết quả:

```
Script start
Script End
Promise 1 resolved
Promise 2 resolved
setTimeout 1
setTimeout 2
```

Kết luận

Chúng ta đã tìm hiểu cách JavaScript Synchronous và JavaScript Asynchronous hoạt động và các khái niệm như **call stack**, **event loop**, **message queue/task queue** và **job queue/micro-task queue**. Hy vọng bài viết này giúp ích được cho các bạn.

TechTalk via viblo.asia

> **Một số lệnh Javascript console hữu ích**

> **Nắm rõ JAVA LAMBDA EXPRESSION cho người mới bắt đầu**

TAGS javascript

CHIA SẺ Facebook Twitter Thích 745 Tweet

TECH Talk

VỀ CHÚNG TÔI

• Giấy phép thiết lập Mạng xã hội số 569/GP-BTTTT do Bộ Thông Tin và Truyền Thông cấp.
Trụ sở: 179 Đường Nguyễn Đình Chính, Phường 11, Quận PN, TP.HCM

Liên hệ: hieuld@applancer.net - Tel: 028 6264 5022

THEO CHÚNG TÔI