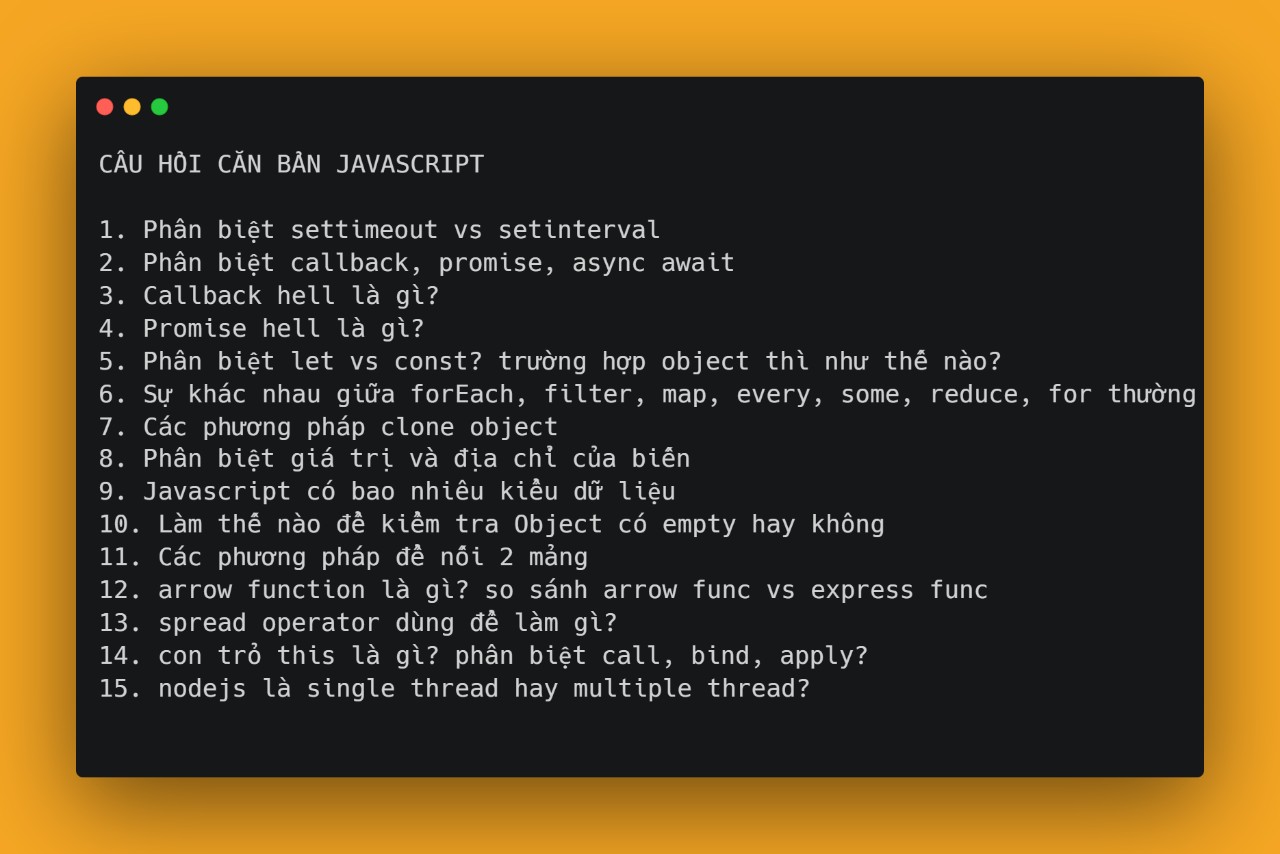
Câu hỏi căn bản JavaScript



Câu trả lời

[**Câu 1. Phân biệt settimeout và setinterval** 2](#_Toc116748289)

[**Câu 2. Phân biệt callback, promise, async await** 3](#_Toc116748290)

[**Câu 3. Callback hell là gì?** 4](#_Toc116748291)

[**Câu 4. Promise hell là gì?** 5](#_Toc116748292)

[**Câu 5. Phân biệt let và const, trường hợp Object thì làm thế nào?** 6](#_Toc116748293)

[**Câu 6. Sự khác nhau giữa forEach, filter, every, some, reduce, for thường** 7](#_Toc116748294)

[**Câu 7. Các phương pháp clone object** 9](#_Toc116748295)

[**Câu 8. Phân biệt giá trị và địa chỉ của biến** 11](#_Toc116748296)

[**Câu 9. JavaScript có bao nhiêu kiểu giữ liệu** 12](#_Toc116748297)

[**Câu 10. Làm thế nào để kiểm trả object có empty hay không?** 14](#_Toc116748298)

[**Câu 11. Các phương pháp để nối hai mảng** 15](#_Toc116748299)

## **Câu 1. Phân biệt settimeout và setinterval**

* Hàm setTimeout() thường được sử dụng khi mình muốn thực hiện method sau khoảng thời gian nào đó (mili giây)

setTimeout ( expression, timeout );

Như cú pháp trên thì expression sẽ được thực hiện sau timeout (mili giây)

setTimeout(() => {

    console.log("Hello")

}, 1000);

Vị dụ khi load trang thì sau 1000 mili giây sẽ in ra chữ “Hello”

* Hàm setinterval sẽ dùng khi mình muốn thực hiện method liên tục sau khoảng thời gian chờ nào đó.

setInterval ( expression, time );

Như cú pháp trên thì expression sẽ được thực hiện sau mỗi time (mili giây)

setInterval(() => {

    console.log("Bye")

}, 2000)

Ví dụ sao khi load trang 2 giây sẽ in ra chữ “Bye”, và cứ sau 2 giây lại in ra chữ “Bye”.

## **Câu 2. Phân biệt callback, promise, async await**

Điểm chung là đều dùng để sử lí bất đồng bộ

* Callback là một function được gọi ra khi một thao tác bất đồng bộ có kết quả đã sẵn sàng (kết quả này có thể là dữ liệu trả về hoặc lỗi xảy ra khi thao tác)

Callback thường được sử dụng nhiều trong Jquery:

$("#button").click(function() {

    alert("Click")

})

Thì hàm callback là:

function() {

    alert("Click")

}

* Promíe nó có cú pháp rõ ràng hơn

const myPromise = new Promise(function(resolve, reject) {

    // code here

    if (codeIsFine) {

        resolve('fine')

    } else {

        reject('error')

    }

})

myPromise

    .then(function whenOk(response) {

        console.log(response)

        return response

    })

    .catch(function notOk(err) {

        console.error(err)

})

Hãy phân tích đoạn code trên:

Một promise được khởi tạo với một function có câu lệnh resolve hoặc reject

Đặt đoạn code bất đồng bộ trong function Promise resolve khi mọi thứ xảy ra như mong muốn Nếu không thì reject

Khi resolve được chạy thì đoạn code trong .then sẽ thực thi Khi reject được chạy thì .catch sẽ được trigger

* Async sẽ thông báo rằng function sẽ xử lý bất đồng bộ, và await sẽ được dùng để báo chúng ta muốn đợi kết quả của thao tác bất đồng bộ trong một function có đánh dấu async.

## **Câu 3. Callback hell là gì?**

Callback hell kiểu như đã là mình gọi callback quá nhiều lần, sẽ làm code bị xấu. Ví dụ:

const animateAll = (animate) => {

        setTimeout(() => {

            animate(words[0]);

            setTimeout(() => {

                animate(words[1]);

                setTimeout(() => {

                    animate(words[2]);

                }, 1000)

            }, 1000)

        }, 1000)

    }

Làm thế nào để thoát khỏi callback hell?

* JavaScript cung cấp một cách dễ dàng để thoát khỏi callback hell. Điều này được thực hiện bởi hàng đợi sự kiện (event queue) và các promise.
* Một promise là một đối tượng (object) được trả về (return) từ bất kỳ hàm không đồng bộ (asynchronous function) nào, mà các hàm callback (callback function) có thể được thêm vào dựa trên kết quả của hàm trước đó.
* Promise sử dụng phương thức (method) .then() để gọi các callback không đồng bộ (async callback). Chúng ta có thể nối (chain) bao nhiêu callback tùy thích và thứ tự (order) cũng được duy trì nghiêm ngặt.
* Promise sử dụng phương thức .fetch() để tìm nạp một đối tượng từ mạng (network). Nó cũng sử dụng phương thức .catch() để bắt (catch) bất kỳ ngoại lệ (exception) nào khi bất kỳ khối (block) nào bị lỗi.
* Vì vậy, những promise này được đưa vào hàng đợi sự kiện để chúng không chặn (block) mã JS tiếp theo. Ngoài ra khi kết quả được trả về, hàng đợi sự kiện sẽ kết thúc hoạt động của nó.
* Ngoài ra còn có các từ khóa (keyword) và phương thức hữu ích khác như async, wait, settimeout() để đơn giản hóa và sử dụng tốt hơn các callback.

## **Câu 4. Promise hell là gì?**

Tuy nói Promise giúp tổ chức code, và tránh callback hell, nhưng nếu viết code không khô thoáng và sạch sẽ thì anh em vẫn rơi vào Promise Hell. Do đó lúc nào chúng ta cũng phải dùng kotexcode.

Promise Hell

funA()

    .then(function(tien) {

        return funB(tien)

            .then(function(tien\_con\_lai) {

                return funC(tien\_con\_lai)

                    .then(function() {

                        return hue\_oi();

                    })

            })

    })

Viết lại thành Kotexcode

funA()

    .then(funB)

    .then(funC)

    .catch(console.error.bind(console));

## **Câu 5. Phân biệt let và const, trường hợp Object thì làm thế nào?**

* Một trong những nguyên nhân khiến let có thể thay thế var để xử lý vấn đề nêu trên là vì biến let được khai báo sẽ có scope là block scoped chứ không phải globally hay locally scoped.

let greeting = "say Hi";

let times = 4;

if (times > 3) {

    let hello = "say Hello instead";

    console.log(hello); // "say Hello instead"

}

console.log(hello); // hello is not defined

* Tương tự với let cũng có scope là block scoped, và hoisting thì chúng ta có thêm 1 kiểu khai báo biến nữa là const. Trong biến const nếu trường hợp kiểu của biến là primitive (bao gồm string, number, boolean, null, và undefined) thì chúng ta sẽ không thể tái khai báo hay cập nhật giá trị mới để thay thế cho giá trị trước đó của biến.

const greeting = "say Hi";

greeting = "say Hello instead"; // error : Assignment to constant variable.

const greeting = "say Hi";

const greeting = "say Hello instead"; // error : Identifier 'greeting' has already been declared

* Đối với trường hợp kiểu biến là reference (bao gồm object, array, và function) thì tuy không thể tái khai báo hay cập nhật giá trị của biến nhưng chúng ta vẫn có thể cập nhật giá trị cho thuộc tính của biến đó.

const greeting = {

    message : "Hello",

    number : "five"

}

greeting.message = "say Hello instead";

console.log(greeting); // {message:"say Hello instead",number:"five"}

## **Câu 6. Sự khác nhau giữa forEach, filter, every, some, reduce, for thường**

* For thường: for thường sử dụng môt biến để chạy theo chỉ số trong một array, từ đó xác định được các phần tử trong mảng:

var array = [1, 2, 3];

for (var i = 0; i < array.length; i++) {

    console.log(i);

}

* ForEach: For dùng để chạy lần lượt các phần tử trong một mảng, (đoạn này em không biết giải thích sau, kiểu nó giống như một list-linker, mỗi phần tử nó sẽ nối đến phần tử sau nó:

var array = [1,2,3];

array.forEach(function(i){

  console.log(i);

});

* Every: đây là một phương thức dành cho đối tượng mảng trong javascript. Công dụng của hàm này là giúp kiểm tra tất cả các phần tử trong mảng có thõa mãn một điều kiện nào đó hay không. Nếu tất cả phần tử đều thỏa thì sẽ trả về true, ngược lại nếu chỉ cần một phần tử không thỏa thôi là nó sẽ trả về false.
* Some: Hàm some trong js có nhiệm vụ lặp qua tất cả các phần tử của mảng, mỗi lần lặp nó sẽ truyền giá trị của phần tử đang lặp vào hàm callback. Chỉ cần hàm callback return true là hàm some sẽ return true. Ngược lại, nếu duyệt hết mảng mà không có return true nào thì hàm some sẽ return false.
* Reduce: là một phương thức sẵn có được sử dụng để thực thi một hàm lên các phần tử của mảng (từ trái sang phải) với một biến tích lũy để thu về một giá trị duy nhất. Là một phương thức quan trọng hay sử dụng trong lập trình hàm.

const data = [5, 10, 15, 20, 25];

const res = data.reduce((total,currentValue) => {

  return total + currentValue;

});

console.log(res); // 75

* Filter: Hàm filter được tích hơp sẵn trong đối tượng mảng giống như hàm map trong javascript. Về cách thức hoạt động thì nó giống hàm map, nghĩa là nó sẽ có một tham số dạng function và function này sẽ xử lý cho mỗi lần lặp. Nếu function return true thì phần tử lần lặp đó được chấp nhận, ngược lại sẽ không được chấp nhận.

var numbers = [100, 200, 300, 400, 500];

var selected\_number = numbers.filter(function(value) {

    if (value == 300) {

        return false;

    }

    return true;

});

console.log(selected\_number); // 100, 200, 400, 500

## **Câu 7. Các phương pháp clone object**

* Sử dụng Spread (nhóm Shallow Copy):

const obj\_1 = {

    username: "HaiZuka",

    getUsername() {

        return this.username;

    }

};

const obj\_2 = {...obj\_1 };

obj\_1.username = "10";

console.log("obj\_2", obj\_2); // {username: "HaiZuka", getUsername: ƒ}

* Sử dụng Object.assign() (nhóm Shallow Copy):

const obj\_1 = {

    username: "chamdev.com",

    info: {

        address: "https://chamdev.com"

    },

    getUsername() {

        return this.username;

    }

};

const obj\_3 = Object.assign({}, obj\_1);

obj\_1.age = 10;

console.log("obj\_2", obj\_3); // {username: "haizuka.com", info: {address: "https://haizuka.com"}, getUsername: ƒ}

* Sử dụng phương thức JSON (nhóm Deep Clone)

const obj\_1 = {

    username: "haizuka.com",

    info: {

        address: "https://haizuka.com"

    },

    getUsername() {

        return this.username;

    }

};

const obj\_4 = JSON.parse(JSON.stringify(obj\_1));

obj\_1.age = 10;

console.log("obj\_4", obj\_4); // {username: "haizuka.com", info: {address: "https://haizuka.com"}}

* Sử dụng thư viện bên thứ 3 – Lodash (nhóm Deep Clone)

const \_ = require("lodash");

const obj\_1 = {

    username: "haizuka.com",

    info: {

        address: "https://haizuka.com"

    },

    getUsername() {

        return this.username;

    }

};

const obj\_5 = \_.cloneDeep(obj\_1);

const obj\_6 = \_.clone(obj\_1);

obj\_1.age = 10;

obj\_1.info.address = "Not found";

console.log("obj\_5", obj\_5); // {username: "haizuka.com", info: {address: "https://haizuka.com"}, getUsername: ƒ}

console.log("obj\_6", obj\_6; // {username: "haizuka.com", info: {address: "Not found"}, getUsername: ƒ}

## **Câu 8. Phân biệt giá trị và địa chỉ của biến**

* Địa chỉ dùng để chỉ vùng nhớ, nơi lưu trữ giá trị của biến, từ địa chỉ ta có thể thay đổi giá trị của biến đó, và địa chỉ này là duy nhất.

Timeline

Description automatically generated

* Giá trị của biến là một con số, một chuỗi,… là thứ quan trọng nhất của biến, mình có thể thay đổi giá trị bằng nhiều cách, trong đó có thể dùng cách tác động lên địa chỉ của biến đó.

**Câu 9. JavaScript có bao nhiêu kiểu giữ liệu**

JavaScript có 8 kiểu dữ liệu cơ bản

1. Kiểu dữ liệu boolean (kiểu logic)

let isWebLoaded = true; // => Trang web đã được tải xong

console.log(isWebLoaded); // true

let isProgramRunning = false; // Chương trình đang không chạy

console.log(isProgramRunning); // false

1. Kiểu dữ liệu null

let language = null;

console.log(language); // null

1. Kiểu dữ liệu undefined

let language2 = undefined;

console.log(language2); // undefined

let language3;

console.log(language3); // undefined

1. Kiểu dữ liệu number

let n1 = 66; // số nguyên dương

let n2 = -66; // số nguyên âm

let n3 = 3.14; // số thực dương

let n4 = -3.14; // số thực âm

let n5 = 2e3; // => 2\*10^3 = 2000

let n6 = 2e-3; // => 2\*10^(-3) = 0.002

let n7 = 0xff; // số dạng hexa (hệ cơ số 16): 15\*16 + 15 = 255

let n8 = 067; // số dạng octa (hệ cơ số 8): 6\*8 + 7 = 55

let n9 = 0b11; // số dạng nhị phân (hệ cơ số 2): 1\*2 + 1 = 3

1. Kiểu dữ liệu BigInt

Trong JavaScript, kiểu dữ liệu number không thể biểu diễn một số nguyên lớn hơn (2^53-1) (bằng 9007199254740991) và nhỏ hơn -(2^53-1).

Với hầu hết các trường hợp, việc sử dụng kiểu dữ liệu number là quá đủ. Nhưng đôi khi, bạn vẫn cần biểu diễn và tính toán với những số nguyên cực kỳ lớn. Do đó, kiểu dữ liệu BigInt ra đời nhằm giải quyết vấn đề này.

Để biểu diễn số nguyên với kiểu BigInt, bạn c

const reallyBigNumber = 12345678987654321012345678987654321n;

console.log(reallyBigNumber); // 12345678987654321012345678987654321n

1. Kiểu dữ liệu string

const msg1 = 'Đây là string dùng dấu nháy đơn';

const msg2 = "Đây là string dùng dấu nháy kép";

const msg3 = `Đây là string dùng dấu backtick`;

const msg4 = '1 + 2 = ${1 + 2}';

1. Kiểu dữ liệu symbol

Symbol là một kiểu dữ liệu nguyên thủy dùng để tạo ra các giá trị duy nhất (unique value) và bất biến (immutable). Symbol thường được dùng làm key cho kiểu dữ liệu object sau đây.

1. Kiểu dữ liệu object

Object là kiểu dữ liệu tham chiếu. Có thể hiểu object là một tập hợp gồm các cặp key - value (khóa - giá trị).

## **Câu 10. Làm thế nào để kiểm trả object có empty hay không?**

Để kiểm tra object là empty trong ES6, mình sử dụng phương thức Object.keys(). Phương thức này trả về một mảng chứa tất cả các thuộc tính enumerable của object.

Nếu kết quả trả về là mảng rỗng thì suy ra object đó không chứa thuộc tính enumerable nào.

Ví dụ sử dụng Object.keys():

const a = {},

    b = { x: 1 },

    c = { m: "a", n: "b" };

console.log(Object.keys(a)); // []

console.log(Object.keys(b)); // ["x"]

console.log(Object.keys(c)); // ["m", "n"]

Mình có thể viết hàm kiểm tra object có empty hay không như sau:

const isEmpty = (v) => {

    return Object.keys(v).length === 0;

};

## **Câu 11. Các phương pháp để nối hai mảng**

* Dùng hàm push

Thêm lần lượt các phần tử mảng kia và mảng cần ghép.

const array1 = [1, 2, 3];

const array2 = [4, 5, 6];

for (const element of array2) {

    array1.push(element);

}

Có thể dùng spread thay vì vòng for

array1.push(...array2);

* Sử dụng hàm concat

const num1 = [1, 2, 3];

const num2 = [4, 5, 6];

const num3 = [7, 8, 9];

const numbers = num1.concat(num2, num3);

console.log(numbers);

// results in [1, 2, 3, 4, 5, 6, 7, 8, 9]

* Sử dụng spread

const concatenated = [...array1, ...array2];

## **Câu 12. Arrow fuction là gì, so sánh arrow funtion và express function**

Arrow function hoạt động tương tự như Lambdas trong các ngôn ngữ khác như C # hay Python. Bằng cách sử dụng arrow function, chúng ta tránh được việc phải gõ từ khoá function, return và dấu ngoặc nhọn.

// ES5

var multiply = function(x, y) {

    return x \* y;

};

// ES6

var multiply = (x, y) => { return x \* y };

Arrow function thường ngắn gọn hơn function

Với hàm số có 1 tham số, arrow function có thể bỏ qua cặp dấu ngoặc đơn.

// Sử dụng arrow function

let greeting = name => {

    console.log(`Hello, my name is ${name}`);

}

greeting("Hai");

// => Hello, my name is Hai

Sử dụng map với arrow function:

let arr = [1, 2, 3, 4];

let square2 = arr.map(e => e \* e);

console.log(square2); // => [1, 4, 9, 16]

Arrow function không phù hợp làm method cho object

Chính vì arrow function không định nghĩa giá trị this của riêng nó, nên cũng không phù hợp là method cho object.

let obj = {

    a: 1,

    b: () => console.log(this.a, this),

    c: function() {

        console.log(this.a, this);

    }

}

obj.b(); // prints undefined, Window {...} (or the global object)

obj.c(); // prints 1, Object {...}

## **Câu 13. Spread operator dùng để làm gì?**

Spread operator có cú pháp giống với rest paraterter tuy nhiên cả hai có ý nghĩa khác nhau. Rest paramter được sử dụng khi khai báo hàm, ngược lại spread operator được sử dụng trong các câu lệnh, biểu thức hoặc khi gọi hàm.

ES6 cung cấp một toán tử mới gọi là spread operator bao gồm ba dấu chấm (...). Spread operator cho phép bạn trải ra các phần tử của một đối tượng có thể lặp lại, chẳng hạn như một array, map hoặc set.

Trước khi ES6 được ra đời thì cách phổ biến để nối mảng là sử dụng phương thức concat() của một mảng với đối số truyền vào là những mảng khác sẽ được nối với mảng này: VD:

var arr\_1 = [1, 2, 3];

var arr\_2 = [4, 5, 6];

arr\_3 = arr\_1.concat(arr\_2);

console.log(arr\_3); //  [1, 2, 3, 4, 5, 6]

Phía dưới là dùng spread operator

const arr\_1 = [1,3,5];

const arr\_2 = [2,4,6, ...odd];

console.log(arr\_2); // [ 2, 4, 6, 1, 3, 5 ]

## **Câu 14. Con trỏ this là gì, phân biệt call, bind, apply**

This đại diện cho đối tượng trong hàm, class, object

Khi thực thi fuction đó, nó sẽ có Property thischứa item của 1 Object đang gọi tới function này.

Ví dụ trong object:

const Person = {

    ho: 'Phan',

    ten: 'Hai',

    ghepHoTen: function() {

        console.log('Họ và tên', this.ho + " " + this.ten);

    }

}

Person.ghepHoTen();

Call và apply có vẻ rất giống nhau---đều gọi 1 function với 1 giá trị this xác định mà mình gán cho, cùng với những arguments. Điều khác nhau duy nhất giữa 2 method này đó là arguments truyền vào call sẽ được đọc từng giá trị một, trong khi arguments truyền vào apply là 1 array

function longerSummary(genre, year) {

    console.log(

        `${this.title} was written by ${this.author}. It is a ${genre} novel written in ${year}.`

    )

}

longerSummary.call(book, 'dystopian', 1932); //"Brave New World was written by Aldous Huxley. It is a dystopian novel written in 1932."

longerSummary.apply(book, 'dystopian', 1932); //Uncaught TypeError: CreateListFromArrayLike called on non-object at <anonymous>:1:15

Hai method apply() và call() đều là 2 method gọi 1 lần. Tức là chỉ khi gọi 2 hàm này thì giá trị của this mới được gán mới, còn bản chất function đó không hề thay đổi chút nào. Tuy nhiên, đôi khi sẽ có lúc chúng ta muốn sử dụng 1 method nào đó nhiều lần trong code với this của method đó sẽ do bạn gán. thì lúc đó cần sử dụng bind() để tạo ra 1 function mới thỏa mãn yêu cầu trên.

const braveNewWorldSummary = summary.bind(book)

braveNewWorldSummary() //"Brave New World was written by Aldous Huxley"

## **Câu 15. NodeJS là single thread hay multiple thread?**

NodeJS là single thread.

Chart, waterfall chart

Description automatically generated

NodeJS sẽ thực thi chỉ theo một luông.

const http = require('http')

function wait(millisec) {

    var now = new Date;

    while (new Date - now <= millisec) ;

}

http.createServer((req, res)=> {

    if (req.url === '/') {

        res.writeHead(200, {"Content-Type": 'text/html'});

        res.write('hello')

        res.end()

    }

    if (req.url === '/wait') {

        wait(5000)

        console.log('wait');

        res.writeHead(200, {"Content-Type": 'text/html'});

        res.write('Done>>>wait')

        res.end()

    }

    if (req.url === '/timeout') {

        setTimeout(()=> {

            res.writeHead(200, {"Content-Type": 'text/html'});

            res.write('Done>>>timeout')

            res.end()

        }, 5000)

        console.log('timeout');

    }

}).listen(3000, "127.0.0.1", function() {

    console.log('server start at http://127.0.0.1:3000')

})

Sau chạy server và chạy <http://localhost:3000> lập tức in ra chữ “hello”.

Những nếu ta chạy lại và chạy <http://localhost:3000/wait> thì sẽ phải chờ 5 giây bởi hàm wait(), cùng lúc đó ta vào <http://localhost:3000> thì nó vẫn loading do luồng nó đang chạy đang xử lí ở hàm wait(), chứng tỏ NodeJS là single thread.