JavaScript Research

1. Scope là gì?

Trong Javascript chúng ta định nghĩa Scope nghĩa là phạm vi hoạt động của biến, đối tượng và function. Khi nào chúng tồn tại và khi nào có thể gọi được chúng.

Có 2 loại scope trong javascript là Global (phạm vi toàn cục) và Local (phạm phi nhất định).

1. Global Scope là gì?

Các biến khai báo ngoài các function được gọi là global. Biến này được dùng chung cho các function, các function có thể thấy được biến này và thay đổi giá trị của biến này.

|  |
| --- |
| var userName = "Bill";  function modifyUserName() {  userName = "Steve";  };  function showUserName() {  console.log(userName);  };  console.log(userName); // display Bill    modifyUserName();  showUserName();// display Steve |

Biến userName là global nên function modifyUserName và showUserName có thể thấy được và sử dụng được.

Các biến khai báo trong function nhưng không có từ khoá var, thì cũng được xem là biến có scope global.

|  |
| --- |
| function createUserName() {  userName = "Bill";  }  function modifyUserName() {  if(userName)  userName = "Steve";  };  function showUserName() {  console.log(userName);  }    createUserName();  showUserName(); // Bill  modifyUserName();  showUserName(); // Steve |

1. Local scope là gì?

Các biến được khai báo bên trong function và có từ khoá var được gọi là local. Biến local chỉ được sử dụng bên trong function. Các function khác bên ngoài không thể sử dụng được biến local này.

|  |
| --- |
| function createUserName() {  var userName = "Bill";  }  function showUserName() {  console.log(userName);  }  createUserName();  showUserName(); // throws error: userName is not defined |

Biến userName chỉ được sử dụng trong function createUserName mà thôi, các function khác không thấy được biến này.

1. Function scope

Nó cũng tương tự như local scope là đều nằm trong function. Sự khác biệt của function scope vs local scope là function scope ko dc đặt trong biểu thức

Cặp dấu {} như if, while, for trong khi khái niệm local scope thì có.

|  |
| --- |
| function A() {  var a = 1; //local & function scope  if(a == 1){  var b = 1 //local scope  }  }; |

* Có thể thấy rằng local scope có thể là function scope nhưng function scope chưa chắc đã là local scope

1. Lexical Scope

Bất cứ biến/object/ function được định nghĩa trong parent scope, đều có thể được truy cập bởi các scope con nhỏ hơn.VD:

|  |
| --- |
| var Favourite = function () {      var food = "icecream";           //parent scope        var myFavouriteFood = function () {          console.log("I love " + food);// ‘I love icream’      };      myFavouriteFood();  };  Favourite(); |

Tương tự :

|  |
| --- |
| var name = 'Xung';  var scope1 = function () {  // name is available here  var scope2 = function () {  // name is available here too  var scope3 = function () {  // name is also available here!  };  };  }; |

Nhưng ko thể truy cập ngược lại từ function con ra function cha.

|  |
| --- |
| function A() {  console.log(name);  function B() {  var name = “Xung”;  }  } |

1. Scope Chain

Khái niệm scope chain là khái niệm : khi một biến scope được gọi mà ko dc khai báo ở scope hiện tại thì js sẽ tự tìm đến biến đó ở scope cha. Nếu scope cha ko có thì sẽ báo lỗi còn có thì sẽ có 2 trường hợp: 1 là undefined hoặc là sẽ tìm dc.

VD:

|  |
| --- |
| function show() {      console.log(text); //undefined    }  show();  var text = “ashiba”; |

Mà muốn xuất ra từ ashiba thì phải khai báo ở trên

|  |
| --- |
| var text = “ashiba”;  function show() {      console.log(text); //’ashiba’    }  show(); |

? Tại sao khi khai báo biến text ở dưới function show() vs vd 1 thì nó vẫn chạy và ko báo lỗi mặc dù kq là undefined ?

* Bởi vì trong JS nó chạy theo cơ chế Hosting. Vậy cơ chế hosting là gì?

1. Hosting trong JS

**Hoisting** là một cơ chế Javascript. Các biến và hàm trong JS mà chúng ta khai báo ở thứ tự trước hoặc sau thì khi có dòng code gọi đến hàm hoặc biến đó thì cũng sẽ gọi dc kể cả khi ta viết nó sau code thực thi.

**VD:**

|  |
| --- |
| show(); // ‘ashiba’  function show() {      console.log("ashiba");  } |

Nhưng lưu ý vs biến. Ta sẽ lấy VD ở trên:

|  |  |
| --- | --- |
| var text;  function show() {      console.log(text); //undefined    }  show();  text = “ashiba”; | function show() {      console.log(text); //undefined    }  show();  var text = “ashiba”; |

Đều ra là 1 kết quả tương tự là undefined.

🡺Cho thấy rằng hàm thì có thể viết tùy ý dc nhưng đối vs biến thì phải cẩn thận thứ tự đặt biến.

Có 1 lưu ý nhỏ là: Nếu ko ghi gì hết thì mặc định kiểu là var.

|  |
| --- |
| text = "ashiba"  console.log(text)   //’ashiba’ |

Ta thử ghi ngược lại 2 dòng này vs biến kiểu var, let hoặc ko có.

|  |  |
| --- | --- |
| // Non type  console.log(text)   //Error  text = "ashiba" | //LET  console.log(text)   //Error  let text = "ashiba" |

|  |
| --- |
| //VAR  console.log(text)   //Undefined  var text = "ashiba" |

* Như ta có thể thấy, nếu ko ghi kiểu dữ liệu thì nó sẽ ngầm định là var, nhưng 2 kiểu này vẫn sẽ có sự khác biệt thông qua ví dụ trên.

1. Closure

Ta hãy xem đoạn code và giải thích sau:

// Tạo một biến toàn cục

var counter = 0;

//hàm đếm

function makeCounter() {

return counter += 1;

}

// Gọi hàm

makeCounter();

console.log(counter); // Kết quả: 1

makeCounter();

console.log(counter); // Kết quả: 2

// Thử thay đổi biến counter từ bên ngoài

counter = 10;

console.log(counter); // Kết quả: 10

\_Như ta thấy trong ví dụ trên, giá trị của biến **counter** có thể được thay đổi từ bất kỳ đâu trong chương trình mà không cần gọi hàm ***makeCounter().***

\_Nếu ta ko gọi hàm để thay đổi, mà ta thay đổi giá trị của biến counter trực tiếp bên ngoài thì biến đó cũng sẽ bị thay đổi. Điều này có thể sẽ ảnh hưởng đến việc kiểm soát dữ liệu sau này vô tình sẽ bị thay đổi giá trị của biến.

**Ta hãy thử làm bằng cách ko đặt biến global scope thì điều gì sẽ xảy ra:**

function makeCounter() {

// Tạo biến cục bộ

var counter = 0;

// Thao tác với biến counter

return counter += 1;

}

// Gọi hàm

console.log(makeCounter()); // Kết quả: 1

console.log(makeCounter()); // Kết quả: 1

\_Như ta thấy thì điều này đã làm được điều là ko cho biến counter dc thay đổi bên ngoài mà chỉ dc thay đổi khi gọi hàm makeCounter() nhưng kết quả thì lại ko nhưng mong đợi.

\_Vì mỗi khi hàm makeCounter() được gọi thì biến cục bộ bên trong hàm đó sẽ set lại bằng 0.

Vậy nếu chúng ta vẫn muốn đạt kết quả tương tự như ví dụ 1 mà lại không thay đổi biến counter từ bên ngoài thì làm thế nào?

* SỬ DỤNG CLOSURE

Về cơ bản, **Closure** là một **hàm bên trong** (**inner function**) có quyền truy cập vào phạm vi của hàm mẹ, ngay cả sau khi hàm mẹ đã thực thi xong.

Điều này được thực hiện bằng cách **tạo một hàm bên trong một hàm khác**. Hãy xem ví dụ sau để xem nó hoạt động như thế nào:

function makeCounter() {

var counter = 0; //Được hiểu như là this.counter = 0

// Hàm bên trong hàm

function make() { //return function make(){} cũng được

counter += 1; //this.counter += 1;

return counter; //return this.counter;

}

return make;

}

Khi ta gọi hàm makeCounter thì sẽ ko có chuyện gì xảy ra

makeCounter();

mà ta phải gắn cái function makeCounter cho một biến khác chứa nó và dùng biến đó như 1 function

/\* Thực thi hàm makeCounter() và lưu trữ giá trị

được trả về bằng biến myCounter \*/

var myCounter = makeCounter();

console.log(myCounter()); // Kết quả: 1

console.log(myCounter()); // Kết quả: 2

console.log(makeCounter()()); // Kết quả: 1

console.log(makeCounter()()); // Kết quả: 1

\_Biến myCounter được hiểu như là đối tượng lưu giá trị của biến counter.

\_Giá trị của **myCounter** bản chất là hàm ***make()*** bên trong, việc mỗi lần gọi hàm ***myCounter()*** sẽ là gọi hàm ***make()*** bên trong kia luôn chứ ko cần thông qua makeCounter() nữa.

\_Hàm make() ở đây là hàm closure(hàm ngoài return đến hàm trong thay đổi và lưu giá trị thì hàm đó sẽ là closure). Closures lưu trữ nội bộ các tham chiếu đến các biến bên ngoài của chúng và có thể truy cập và cập nhật các giá trị của chúng.

P/s: Làm thêm để xem kết quả tương tự:

function makeCounter() {

var counter = 0;

// Hàm bên trong hàm

function make() {

counter += 1;

function extra() {

counter += 1;

return counter;

}

}

return make;

}

/\* Thực thi hàm makeCounter() và lưu trữ giá trị

được trả về bằng biến myCounter \*/

var myCounter = makeCounter();

console.log(myCounter()()); // Kết quả: 1

console.log(myCounter()()); // Kết quả: 2

Nếu ta tạo 2 biến khác nhau cùng gọi hàm makeCounter() thì sao, kết quả bên dưới ha.

var myCounter = makeCounter();

console.log(myCounter()); // Kết quả: 1

console.log(myCounter()); // Kết quả: 2

console.log(myCounter()); // Kết quả: 3

var hisCounter = makeCounter();

console.log(hisCounter()); // Kết quả: 1

console.log(hisCounter()); // Kết quả: 2

1. Scope and this

Mỗi scope lại bind giá trị khác nhau cho this tùy thuộc vào vị trí nó được gọi tới. Mặc định thì this bind đến object toàn cục nhất window. Ta cùng xem cách gọi hàm khác nhau cho ra kết quả của this khác nhau:

|  |
| --- |
| 1. var myFunction = function () {   console.log(this); // this = global, [object Window]  };  myFunction();   1. var myObject = {};   myObject.myMethod = function () {  console.log(this); // this = Object { myObject }  };  myObject.myMethod();   1. var nav = document.querySelector('.nav'); // <nav class="nav">   var toggleNav = function () {  console.log(this); // this = <nav> element  };  nav.addEventListener('click', toggleNav, false); |

Tại ví dụ 1) thì this có phạm vi là toàn cục nên khi chạy sẽ là object global.

Tại ví dụ 2) this sẽ bind tới object toàn cục nhất lúc này là myObject.myMethod và kết quả là this: Object.

Tại ví dụ 3) this bind tới object toàn cục nhất lúc này là toggleNav và kết quả là in ra <nav class="nav"></nav>

Trong một số trường hợp nhất định Scope vẫn có thể thay đổi và giá trị this cũng thay đổi theo:

|  |
| --- |
| var nav = document.querySelector('.nav'); // <nav class="nav">  var toggleNav = function () {  console.log(this); // <nav> element  setTimeout(function () {  console.log(this); // [object Window]  }, 1000);  };  nav.addEventListener('click', toggleNav, false); |

Kết quả đạt được là trong Scope của hàm toggleNav thì this sẽ in ra là <nav class="nav"></nav>. Ngoài ra, Scope của setTimeout thì thisin ra là objectWindow.

1. This and Bind

Phương thức bind() trả về một hàm mới, truyền đối tượng vào trong bind như sau:

|  |
| --- |
| function Hello() {      console.log("My name is " + *this*.name +", " + "I want to say " + *this*.hello);  }  let A = {      name: 'A',      hello: "Ashiba"  }  let B = {      name: 'B',      hello: "Ashibon"  }  Hello(); //My name is undefined, I want to say undefined  var str\_A = Hello.bind(A)  str\_A();  //My name is A, I want to say Ashiba  var str\_B = Hello.bind(B)  str\_B();  //My name is B, I want to say Ashibon |

\_Ta gọi hàm Hello khơi khơi thì nó sẽ ko hiểu dc this bên trong nó là của đối tượng nào cả.

\_Ta phải gán biểu thức bind hàm Hello() và trả về 1 hàm mới như trên.

\_Truyền đối tượng nào vào bind thì this sẽ hiểu this đang đại diện cho đối tượng đó và truy xuất đến thuộc tính bên trong đối tượng đó.

? Vậy khi ta dùng với arrow function thì các từ khóa this sẽ như thế nào ?

|  |
| --- |
| var Hello = () => {      console.log("My name is " + *this*.name +", " + "I want to say " + *this*.hello);  } |

Kết quả:

|  |
| --- |
| Hello(); //My name is undefined, I want to say undefined  var str\_A = Hello.bind(A)  str\_A();  // My name is undefined, I want to say undefined  var str\_B = Hello.bind(B)  str\_B();  //My name is undefined, I want to say undefined |

\_Các từ khóa this sẽ ko là đại diện cho đối tượng cho A và B nữa vì trong cấu trúc của Arrow function thì từ khóa this **luôn** đại diện cho đối tượng đã xác định trong hàm mũi tên, chứ nó ko còn bị các đối tượng bind(ràng buộc) nữa.

|  |
| --- |
| var hello = () =>  {      console.log("My name is " + A.name +", " + "I want to say " + A.hello);  } |

|  |
| --- |
| var btn = document.getElementById("btn");  hello = () => {  document.getElementById("demo").innerHTML += this;  } //object window  btn.addEventListener("click", hello);  hello = () => {  document.getElementById("demo").innerHTML += btn;  } //object HTMLButtonElement  window.addEventListener("load", hello);  document.getElementById("btn").addEventListener("click", hello); |

🡺Trong arrow function thì từ khóa this chỉ đại diện cho đối tượng nào đc chỉ đích danh trong nó. Nếu ko chỉ đích danh đối tượng trong arrow function thì có **2 trường hợp**.

**TH1**(this của đối tượng thông thường): mặc định sẽ là Object,

**TH2**(this của các đối tượng html như button, document..): mặc định sẽ là object window.

1. Hàm call()

Call gọi một hàm với giá trị của this và các đối số riêng lẻ. Cú pháp:

**function.call(thisArg, arg1, arg2, ...)**

Trong đó:

**thisArg:** Giá trị của this được đưa ra để gọi hàm. Lưu ý rằng this có thể không phải là giá trị thực tế được thấy bởi phương thức: Nếu phương thức là một hàm trong non-strict mode, giá trị null và undefined sẽ được thay thế với global object và các giá trị sơ khai (primitive) sẽ được chuyển thành các đối tượng (objects).

**arg1, arg2, ...argN:** Các đối số cho hàm.

Ví dụ 1: Sử dung hàm call để gọi một hàm ẩn danh (anonymous function)

|  |
| --- |
| var animals = [  { species: 'Lion', name: 'King' },  { species: 'Whale', name: 'Fail' }  ];  for (var i = 0; i < animals.length; i++) {  (function(i) {  this.print = function() {  console.log('#' + i + ' ' + this.species  + ': ' + this.name);  }  this.print();  }).call(animals[i], i);  }  // #1 Lion: King #2 Whale: Fail |

Trong ví dụ 1, chúng ta tạo một hàm ẩn danh và sử dụng hàm **call** để gọi hàm đó nhận mọi đối tượng trong một mảng. Mục đích chính của hàm ẩn danh này là thêm tính năng hàm print cho mọi đối tượng, từ đó các đối tượng này có thể in ra vị trí của chúng trong mảng.

Ví dụ 2:

|  |
| --- |
| function greet() {  var reply = [this.person, 'is a ', this.role].join(' ');  console.log(reply);  }  var x = {  person: 'Phong Chan Xung', role: 'Javascript Developer'  };  greet.call(x); // Phong Chan Xung is a Javascript Developer |

Trong ví dụ 2, ta gọi hàm greet() và giá trị của this trong hàm greet() chính là x.

Ví dụ 3: Sử dụng call để Contructor Chaining cho một đối tượng

**Constructor Chaining** là quá trình gọi một hàm tạo của một lớp từ một hàm tạo khác của cùng một lớp hoặc một lớp khác bằng cách sử dụng đối tượng hiện tại của lớp.

|  |
| --- |
| function Product(name, price) {  this.name = name;  this.price = price;  }  function Food(name, price) {  Product.call(this, name, price);  this.category = 'food';  }  function Toy(name, price) {  Product.call(this, name, price);  this.category = 'toy';  }  var cheese = new Food('feta', 5);  var fun = new Toy('robot', 40);  console.log(cheese); // Food { name: 'feta', price: 5, category: 'food' }  console.log(fun); // Toy { name: 'robot', price: 40, category: 'toy' } |

Trong ví dụ 3, hàm khởi tạo của đối tượng **Product** được định nghĩa với 2 tham số, **name** và **price**. Hai hàm **Food** và **Toy** gọi **Product** với tham số **this** , **name** và **price**. **Product** khởi tạo thuộc tính **name** và **price**, cả 2 hàm này định nghĩa **category**.

1. Hàm Apply()

Apply gọi một hàm với giá trị của this và các đối số được truyền vào dưới dạng mảng. Cú pháp:

**function.apply(thisArg, [ argsArray])**

**Trong đó:**

* **thisArg:** Giá trị của this được đưa ra để gọi ***hàm***. Lưu ý rằng this có thể không phải là giá trị thực tế được thấy bởi phương thức: Nếu phương thức là một hàm trong **non-strict mode**, giá trị **null** và **undefined** sẽ được thay thế với **global object**và các giá trị sơ khai (primitive) sẽ được chuyển thành các đối tượng **(objects)**.
* **argsArray:** Một mảng chỉ định các đối số cho hàm, hoặc null hoặc undefined nếu không có đối số nào được cung cấp.

Ví dụ 1: Sử dụng hàm apply với build in function (những hàm được cung cấp sẵn)

Cách sử dụng thường thấy là hàm apply cho phép bạn sử dụng các build in function cho một số tác vụ được viết bằng cách lặp lại mảng:

|  |
| --- |
| const numbers = [5, 6, 2, 3, 7];  // sử dụng hàm Math.min/Math.max và apply  let max = Math.max.apply(null, numbers);  // Tương tự như việc dùng Math.max(numbers[0], ...)  // hoặc Math.max(5, 6, ...)  let min = Math.min.apply(null, numbers);  max = -Infinity, min = +Infinity;  for (let i = 0; i < numbers.length; i++) {  if (numbers[i] > max) {  max = numbers[i];  }  if (numbers[i] < min) {  min = numbers[i];  }  } |

Ví dụ 2: Sử dụng hàm apply để nối 1 mảng vào 1 mảng khác.

|  |
| --- |
| const array = ['a', 'b'];  const elements = [0, 1, 2];  array.push.apply(array, elements);  console.info(array); // [ 'a', 'b', 0, 1, 2 ] |

Trong ví dụ 2, để thêm phần tử vào mảng ta sử dụng **push**. bởi vì **push** chấp nhận một số lượng thay đổi đối số, bạn cũng có thể đẩy nhiều phần tử cùng một lúc. Tuy nhiên, nếu bạn chuyển một mảng để đẩy, nó sẽ thêm mảng đó dưới dạng một phần tử duy nhất, thay vì thêm các phần tử riêng lẻ. Vì vậy, kết quả là một mảng bên trong một mảng.

|  |
| --- |
| const array = ['a', 'b'];  const elements = [0, 1, 2];  array.push(array, elements);  console.info(array); // [ 'a', 'b', [ 0, 1, 2 ] ] |

Vì vậy apply là phụ hợp nhất trong trường hợp này.

1. So sánh call() và apply()

Call

|  |
| --- |
| var person1 = {firstName: 'Thị Phương Thảo', lastName: 'Nguyễn'};  var person2 = {firstName: 'Chấn Xung', lastName: 'Phòng'};  function say(greeting1, greeting2) {  console.log(greeting1 + ',' + greeting2 + ' ' + this.lastName + ' ' + this.firstName);  }  say.call(person1, 'Hello', 'Good morning'); // Hello,Good morning Nguyễn Thị Phương Thảo  say.call(person2, 'Hello', 'Good morning'); // Hello,Good morning Phòng Chấn Xung |

Apply

|  |
| --- |
| var person1 = { firstName: 'Thị Phương Thảo', lastName: 'Nguyễn'};  var person2 = { firstName: 'Chấn Xung', lastName: 'Phòng'};  function say(greeting0, greeting1) {  console.log(greeting1 + ',' + greeting2 + ' ' + this.lastName + ' ' + this.firstName);  }  say.apply(person1, ['Hello', 'Good moring']);// Hello,Good morning Nguyễn Thị Phương Thảo    say.apply(person2, ['Hello', 'Good moring']);// Hello,Good morning Phòng Chấn Xung |

Nhìn chung, hàm call và apply là gần giống nhau. Chúng đều gọi hàm trực tiếp. Chỉ khác ở cách truyền tham số vào (apply truyền vào một array chứa toàn bộ các tham số còn call truyền lần lượt từng tham số).