

Scope

- 1. Scope[]?
- 2. Scope Chain & Closure

1. Scope란 ?

변수 선언과 할당, 함수 선언과 호출, 대입 연산, 비교 연산, 논리 연산, 산술 연산

변수 선언과 할당, 함수 선언과 호출, 대입 연산, 비교 연산, 논리 연산, 산술 연산
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

Scope

변수 선언과 할당, 함수 선언과 호출, 대입 연산, 비교 연산, 논리 연산, 산술 연산
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

“ **Global Scope** : 전역 범위 (Global Scope) : 프로그램 전체에서 접근 가능한 범위
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

“ **Local Scope** : 지역 범위 (Local Scope) : 함수 내부에서만 접근 가능한 범위
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

“ **Block Scope** : 블록 범위 (Block Scope) : {}로 둘러싸인 블록 내에서만 접근 가능한 범위 (let, const)
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

“ **Function Scope** : 함수 범위 (Function Scope) : 함수 내부에서만 접근 가능한 범위
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.
이러한 연산들이 이루어지는 환경을 'Scope'라고 하며, 'Scope'는 'Context'라고도 불린다.

1. 全局作用域 (Global Scope)

在 JavaScript 中，全局作用域是指代码在浏览器或 Node.js 环境中运行时的默认作用域。

1) 变量声明：在全局作用域中声明的变量，可以在整个代码文件中访问。

```
let clock = "Wall Clock"; // 全局变量

function livingRoom() {
  console.log(clock); // "Wall Clock"
}

function kitchen() {
  console.log(clock); // "Wall Clock"
}

livingRoom();
kitchen();
```

2. 函数作用域 (Function Scope)

在 JavaScript 中，函数作用域是指代码在函数内部运行时的作用域。

2) 变量声明：在函数内部声明的变量，只能在函数内部访问。

3) 变量提升：在函数内部，变量声明会被提升到函数顶部。

```
function livingRoom() {
  let sofa = "Sofa";
  console.log(sofa); // "Sofa"
}

livingRoom();
console.log(sofa); // Error: sofa is not defined
```

3. 块级作用域 (Block Scope)

在 JavaScript 中，块级作用域是指代码在代码块内部运行时的作用域。

if (true) {
 console.log("Remote Control");
}

console.log(remote); // Error: remote is not defined

```
if (true) {  
  let remote = "Remote Control";  
  console.log(remote); // "Remote Control"  
}  
  
console.log(remote); // Error: remote is not defined
```

4. 词法作用域 (Lexical Scope)

JavaScript 使用词法作用域，即变量在代码中的位置决定了其作用域。

以下代码展示了词法作用域如何影响变量的访问。

1. 全局作用域：在代码的最外层定义的变量，可以在整个程序中访问。

```
let light = "Ceiling Light";  
  
function livingRoom() {  
  console.log(light); // "Ceiling Light"  
}  
  
livingRoom();
```

5. 变量提升

JavaScript 会在代码执行前，将所有的变量声明提升到代码的顶部。

2. 局部作用域：在函数或块级级联语句中定义的变量，只能在定义该变量的作用域内访问。

```
function livingRoom() {  
  let note = "Notebook";  
  
  if (true) {  
    let pen = "Pen";  
    console.log(note); // "Notebook" (在 livingRoom 函数作用域内)  }  
}
```

```
console.log(pen); // "Pen"
```

```
}
```

```
console.log(pen); // Error: pen is not defined
```

```
}
```

```
livingRoom();
```

```
// 00 000: 0 0000 00 00.
```

```
// 00 000: 00 0000 00 00.
```

```
// 00 000: 00 00000 00 00.
```

```
// 000 000: 0000 0 000 0000 000 00.
```

2. Scope Chain & Closure

1. 作用域链(Scope Chain)

JavaScript 引擎在运行时会维护一个作用域链。

作用域链的构建过程是：从当前作用域开始，依次向上查找，直到找到全局作用域。

作用域链的构建顺序是：从当前作用域开始，依次向上查找，直到找到全局作用域。

```
let globalVar = "Global";

function outer() {
  let outerVar = "Outer";

  function inner() {
    let innerVar = "Inner";

    console.log(innerVar); // "Inner" (当前作用域)
    console.log(outerVar); // "Outer" (外层作用域)
    console.log(globalVar); // "Global" (全局作用域)
  }

  inner();
}

outer();
```

► 作用域链的构建顺序是：从当前作用域开始，依次向上查找，直到找到全局作用域。

► 作用域链的构建过程是：从当前作用域开始，依次向上查找，直到找到全局作用域。

2. 闭包(Closure)

闭包是指有权访问其他函数作用域中的变量的函数。

我们使用 `let` 来定义变量，使用 `const` 来定义常量。

```
function createCounter() {
  let count = 0; // 定义一个变量 count，初始值为 0。

  return function () {
    count++; // 每次调用时，count 的值都会增加 1。
    console.log(count);
  };
}

const counter = createCounter();
counter(); // 1
counter(); // 2
counter(); // 3
```

- 我们使用 `let` 来定义变量。
- 我们使用 `const` 来定义常量。
- 我们使用 `let` 来定义变量，使用 `const` 来定义常量。

3. 使用 `let` 和 `const`

ES6 之前我们使用 `var` 来定义变量，使用 `const` 来定义常量。ES6 引入了 `let` 和 `const` 来定义变量和常量。

```
function testVar() {
  if (true) {
    var x = 10;
  }
  console.log(x); // 10 (var 定义的变量在函数范围内是全局的)
}

function testLet() {
  if (true) {
    let y = 20;
  }
  console.log(y); // Error: y is not defined (let 定义的变量只在块级范围内有效)
```

```
}  
  
testVar();  
testLet();
```

▶ var 的变量提升规则是：先扫描整个代码块，将变量声明都提升到代码块的顶部，放到首次遇到 var 语句之前。

▶ let 和 const 的变量提升规则是：不提升。

4. 闭包和函数

闭包是指有权访问其他函数内部变量的那些函数。闭包是函数和有权访问其他函数内部变量的那个函数的组合。闭包是函数和有权访问其他函数内部变量的那个函数的组合。

```
function createResource() {  
  let resource = "Heavy Resource";  
  
  return function () {  
    console.log(resource);  
  };  
}  
  
const useResource = createResource();  
useResource(); // "Heavy Resource"  
  
// 闭包 是指 函数 内部 的变量 在函数 调用后 仍然 存在  
useResource = null;
```

▶ 闭包是指有权访问其他函数内部变量的那些函数。闭包是函数和有权访问其他函数内部变量的那个函数的组合。

▶ 闭包是指有权访问其他函数内部变量的那些函数，闭包是函数和有权访问其他函数内部变量的那个函数的组合。

5. 立即调用函数表达式(IIFE)的应用

IIFE(Immediately Invoked Function Expression)是指立即调用的函数表达式。它是一种立即执行的函数表达式。


```
(function () {  
  let secret = "This is private";  
  console.log(secret); // "This is private"  
})();  
  
console.log(secret); // Error: secret is not defined
```

IIFE 是 一个 立即 执行的 函数。

ES6 引入 了 let, const 来 声明 变量, 变量 声明。

变量 声明: 变量 声明 变量 名称 → 变量 声明 → 变量 声明 变量。

变量: 变量 声明 变量 名称, 变量 声明 变量 名称。

变量 声明 变量: var | let/const 变量 声明 变量。

变量 声明: 变量 声明, 变量 声明 变量 名称 变量 名称 变量 名称。 (变量 声明)

IIFE: 一个 立即 执行的 函数

变量 声明 变量 名称 变量 名称

变量 声明 变量 名称 变量 名称 变量 名称 变量 名称 变量 名称。

```
function createEventHandlers(elements) {  
  elements.forEach((element, index) => {  
    element.addEventListener("click", () => {  
      console.log(`Element ${index} clicked`);  
    });  
  });  
}
```

```
const buttons = document.querySelectorAll("button");  
createEventHandlers(button
```