9.执行上下文和作用域链是什么?

1. 核心概念 (Core Concept)

执行上下文 (Execution Context, EC) 是 JavaScript 引擎执行代码时创建的一个抽象环境。它包含了当前代码执行所需的所有信息，如变量、函数、作用域和 `this` 的指向。每当代码执行流进入一个新的可执行代码块时（全局代码、函数调用），就会创建一个新的执行上下文。

作用域链 (Scope Chain) 是一个由当前执行上下文和其所有父级上下文的**词法环境 (Lexical Environment)** 组成的链式结构。当代码需要访问一个变量时，JavaScript 引擎会首先在当前执行上下文的作用域中查找，如果找不到，就会沿着作用域链向上，到父级作用域中查找，直到找到该变量或到达最顶层的全局作用域。

2. 为什么需要理解它们？ (The "Why")

1. **理解变量的查找机制**: 作用域链解释了为什么以及如何在一个函数内部访问到外部函数甚至全局作用域中的变量。这是理解闭包 (Closure) 的基础。
2. **解释 `var` 的行为和闭包的原理**: 执行上下文和作用域链是解释变量提升、`this` 指向以及闭包如何"记住"其词法作用域的底层核心理论。
3. **调试和性能优化的基础**: 理解执行上下文的创建和销毁过程（执行栈），以及作用域链的查找机制，有助于分析代码的内存使用情况（如闭包导致的内存泄漏）和性能瓶颈。

3. API 与用法 (API & Usage)

这两个概念是 JavaScript 引擎的内部机制，没有直接的 API 可以调用，但它们的行为可以通过代码来观察。

执行上下文 (Execution Context)

JavaScript 中有三种主要的执行上下文：

- **全局执行上下文 (Global EC)**: 这是最基础的上下文。任何不在函数内部的代码都在全局上下文中。它会创建一个全局对象（浏览器中的 `window`），并将 `this` 指向这个全局对象。一个程序中只会有一个全局 EC。
- **函数执行上下文 (Functional EC)**: 每当一个函数被调用时，就会为该函数创建一个新的 EC。每个函数都有自己的 EC。
- **Eval 执行上下文**: 执行 `eval()` 函数内部的代码时会创建。

执行栈 (Execution Stack / Call Stack)
JavaScript 引擎使用一个**栈**来管理执行上下文。当代码开始执行时，全局 EC 首先被压入栈底。每当调用一个函数，其函数 EC 就会被创建并压入栈顶。当函数执行完毕，其 EC 就会从栈顶弹出。

```
function first() {
  console.log('Entering first()');
  second();
  console.log('Exiting first()');
}

function second() {
  console.log('Entering second()');
}

// 1. 全局 EC 入栈
first(); // 2. first() EC 入栈
         // 3. second() EC 入栈
         // 4. second() 执行完毕，EC 出栈
         // 5. first() 执行完毕，EC 出栈
// 6. 全局 EC 在程序结束时出栈
```

作用域链 (Scope Chain)

作用域链是在**函数定义时**就已经确定了的，而不是在函数调用时。这被称为**词法作用域 (Lexical Scoping)**。

```
const globalVar = 'I am global';

function outer() {
  const outerVar = 'I am outer';

  function inner() {
    const innerVar = 'I am inner';

    // 当 inner() 执行时，它的作用域链是：
    // 1. inner() 自身的作用域 (可以访问 innerVar)
    // 2. outer() 的作用域 (可以访问 outerVar)
    // 3. 全局作用域 (可以访问 globalVar)
    console.log(innerVar);  // "I am inner"
    console.log(outerVar);  // "I am outer"
    console.log(globalVar); // "I am global"
  }

  inner();
}

outer();
```

- `inner` 函数的词法作用域包含了 `outer` 函数的作用域和全局作用_域_。
- 当 `inner` 函数被调用并创建其执行上下文时，它的作用域链就由其自身的词法环境和指向其外部词法环境（`outer` 的环境）的引用构成，形成一个链式结构。

4. 关键注意事项 (Key Considerations)

1. **词法作用域决定作用域链**: 一个函数的作用域链是在它被**定义**的位置决定的，而不是在它被**调用**的位置。这是 JavaScript 作用域的核心原则。
2. **闭包的本质**: 闭包就是函数和其**词法环境的引用**的组合。当一个函数（如 `inner`）能够记住并访问它被创建时所在的作用域（如 `outer` 的作用域），即使它在 `outer` 函数执行完毕后才被调用，闭包就产生了。作用域链是实现闭包的底层机制。
3. **性能影响**: 作用域链越长，变量的查找时间就越长。深层嵌套的函数会形成较长的作用域链。此外，闭包会使其外部函数的变量对象（词法环境）无法被垃圾回收，可能导致内存占用增加。
4. **`let` / `const` 与作用域**: ES6 的 `let` 和 `const` 引入了块级作用域，这使得作用域的规则更加精细。一个块（`{...}`）也可以形成一个词法环境，成为作用域链的一部分。

5. 参考资料 (References)

- MDN Web Docs: Scope
- MDN Web Docs: Execution context
- You Don't Know JS Yet: Scope & Closures