1. JavaScript进阶：从传统语法到ES6的革新与高质量代码实践

基本JavaScript语法回顾

变量

- 使用 `var` 声明变量。

```
var name = "John";
```

数据类型

基础类型

- String
- Number
- Boolean
- Null
- Undefined

Object

- Object

使用 `typeof` 来检查类型

💡 **function:** 在JavaScript中，函数（`function`）是一种特殊类型的对象。尽管如此，使用 `typeof` 操作符对函数进行检查会返回 `"function"`，而不是 `"object"`。

例如：

```
console.log(typeof []) // object function myFunction() { return "Hello!"; } console.log(typeof myFunction); // 输出 "function"
```

如果你想验证函数确实是一个对象，你可以使用 `instanceof` 操作符：

```
console.log(myFunction instanceof Object); // 输出 true
```

这里，`myFunction instanceof Object` 返回 `true`，因为在JavaScript中，函数是对象。所以，虽然 `typeof` 对于函数会返回 `"function"`，你可以通过 `instanceof Object` 来验证它们确实也是对象。

💡 **null:** `typeof null` 返回 `"object"` 是JavaScript的一个历史遗留问题。JavaScript的第一个版本是在极其紧张的时间表下开发的，因此它包含了一些设计缺陷和不一致性，这其中就包括 `typeof null` 返回 `"object"` 的行为。在JavaScript的底层实现中，数据值是基于类型标签和值来表示的。对于对象，这个标签是 0。由于 `null` 表示的是一个空指针，具有所有位都设置为0的二进制值，因此它的类型标签也是0。这就是为什么 `typeof null` 返回 `"object"` 的原因：因为它的数据结构在底层表示为一个对象。

运算符

- 算术运算符：`+,-,*,/`
- 比较运算符：`==`，`===`，`!=`，`!==`，`<`，`>`，`<=`，`>=`

严格相等和值相等

- `0 == null`
- `null == undefined`
- `5 == '5'`

💡 在JavaScript，当用 == 做比较时，JS 首先会尝试将两边的值转换为同种类型，再比较值

条件语句

- `if`, `else if`, `else`

```
if (age > 18) { // 逻辑 } else { // 逻辑 }
```

循环

- `for`, `while`

```
for(var i = 0; i < 10; i++) { // 逻辑 }
```

函数

- 使用 `function` 关键字定义。

```
function greet(name) { return "Hello, " + name; }
```

对象

- 对象字面量语法。

```
var person = { name: "John", age: 30 };
```

数组

- 使用方括号 `[]`。

```
var colors = ["red", "green", "blue"]; console.log(colors[0]) // red
```

💡 **数组和对象的区别**

`[]` 是一个语法糖，自动声明一个 key 为自增数字的对象, 数组有多种内建方法，如 `push()`，`pop()`，`shift()`，`unshift()`，`splice()`，`slice()` 等。

```
var colorsArray = ['red', 'green', 'blue'] var colorsObject = {
0: 'red', 1: 'green', 2: 'blue', } console.log(colorsObject[0])
// red
```

`this` 关键字

在JavaScript中，`this` 是一个特殊的关键字，**它在函数被调用时自动定义**，提供一个额外的方式来访问函数执行上下文（context）。简而言之，`this` 是对拥有某个方法或者作为函数调用上下文的对象的一个引用。

- **被调用时：** `o1.fn` → `o1.fn()`
- **自动定义：谁调用指向谁**

基本使用场景

1. **全局上下文**: 在非函数代码中，`this` 指向全局对象（在浏览器中是`Window`，在Node.js中是`global`，在严格模式下是 `undefined`）。

```
console.log(this); // Window
```

2. **函数上下文**: 在普通函数调用中，`this` 通常指向全局对象（非严格模式）或 `undefined`（严格模式）。

```
function myFunction() { console.log(this); } myFunction(); // Window o
r undefined
```

3. **对象方法**: 当函数作为对象的一个属性（即方法）被调用时，`this` 指向这个对象。

```
var obj = { name: "John", greet: function() { console.log(this.name);
} }; obj.greet(); // "John"
```

4. **构造函数**: 使用 `new` 调用函数时，`this` 指向一个新创建的对象。

```
function Person(name) { this.name = name; } var john = new Person("Joh
n"); console.log(john.name); // "John"
```

5. **事件处理**: 在DOM事件处理中，`this` 通常指向触发事件的元素。

```
button.addEventListener("click", function() { console.log(this); // bu
tton element });
```

了解 `this` 的这些基本用法是掌握JavaScript的关键部分，尤其是在处理对象和事件、构造函数等方面。注意，`this` 的行为是动态的，取决于函数是如何被调用的，而不是在哪里被定义或声明的。

面试题

```
var value = 'value' var o1 = { value: 'o1', fn: function() { return
this.value } } console.log(o1.fn()) // o1 var fn = o1.fn
console.log(fn()) // 'value' | undefined (严格模式) var o2 = { value:
'o2', fn: o1.fn, } console.log(o2.fn()) // o2 var o3 = { value: 'o3', fn:
function() { var fn = o2.fn return fn() } } console.log(o3.fn()) //
'value' | undefined (严格模式) o3.fn = o2.fn console.log(o3.fn()) // o3
```

变量提升（Hoisting）

变量提升是 JavaScript 中的一种特殊行为，该机制会在代码执行前将所有的变量和函数声明“提升”到它们所在的作用域的顶部。

工作机制

在代码执行之前，JavaScript 引擎会进行编译，这个过程中会找到所有的变量和函数声明，并将它们移动（逻辑上）到当前作用域的顶部。

提升的内容

- 使用 `var` 声明的变量会被提升，但只有声明部分被提升，初始化（赋值）不会。
- 函数声明也会被提升，包括函数体。

不被提升的内容

- 函数表达式不会被提升。

示例

变量提升

```
console.log(a); // 输出 "undefined" var a = 5; console.log(a); // 输出 "5"
var a console.log(a) // 输出 "undefined" a = 5; console.log(a); // 输出
"5"
```

在这个示例中，变量 `a` 的声明部分被提升了，所以第一个 `console.log(a);` 输出的是 `undefined` 而不是抛出一个错误。

函数声明提升

```
console.log(foo()); // 输出 "Hello" function foo() { return "Hello"; }
```

在这个示例中，整个函数 `foo` （包括其体）都被提升了，所以能够在函数声明之前调用它

函数表达式不被提升

函数表达式是将一个函数赋值给一个变量：

```
var foo = function() { return "Hello"; };
```

函数表达式不会被提升。如果你尝试在声明之前调用这样的函数，你会得到一个错误。

```
console.log(foo()) // TypeError: foo is not a function  var foo = functio
n() { return "Hello"; };
```

```
var foo console.log(foo()) foo = function() { return "Hello"; };
```

了解变量提升机制对于避免一些常见的 JavaScript 编程错误非常重要。特别是在函数和循环内部，不理解提升机制可能会导致意外的结果。

作用域

```
var fns = []; for (var i = 0; i < 3; i++) { fns[i] = function () { return i } } console.log(fns[1]()) // 3
```

```
var fns; var i; fns = [] for (i = 0; i < 3; i++) { fns[i] = function () { return i } } console.log(fns[1]()) // 1 X -> 3
```

```
var fns; var i; fns = [] i = 0 if (i < 3) { fns[i] = function () { return i } i ++ } fns[0] = function () { return i } i = 1 if (i < 3) { fns[i] = function () { return i } i ++ } fns[1] = function () { return i } i = 2 if (i < 3) { fns[i] = function () { return i } i ++ } fns[2] = function () { return i } i = 3 if (i < 3) { ... } console.log(fns[1]()) // 3 console.log(fns[0]()) // 3
```

高质量代码

有人说我写的代码像诗一样优雅！

- 可读性 Readable
- 可维护性 Maintainable
- 可复用性 Reusable

ES6（ECMAScript 2015）

1. `let` 和 `const`

解决的痛点: 变量提升, 块级作用域和不可变性

ES6

```
console.log(x) // Error let x = 10; // 块级作用域 const y = 20; // 块级作用域 + 不可变 y = 10 // Error const fns = []; for (let i = 0; i < 3; i++) { fns[i] = function () { return i } } console.log(fns[1]()) // 1
```

非ES6

```
var x = 10; // 函数作用域，易引起错误
```

`const` 不可以被重新赋值

```
const fns = [] fns[0] = function() { return 0 } fns = [] // Error const obj = {} obj.name = 'o1' obj = {} // Error
```

在JavaScript中，`let` 提供了块级作用域，这意味着变量在声明它的块或语句内是可见的，而在外部则是不可见的。

基础块级作用域

```
{ let x = 1; // x 在这个块内可见 console.log(x); // 输出 1 } // console.log(x); // ReferenceError: x is not defined, 因为 x 在这里不可见
```

在 `if` 语句中的块级作用域

```
if (true) { let y = 2; // y 只在这个 if 块内可见 console.log(y); // 输出 2 } // console.log(y); // ReferenceError: y is not defined, 因为 y 在这里不可见
```

在 `for` 循环中的块级作用域

```
for (let i = 0; i < 3; i++) { // i 只在这个 for 循环块内可见 console.log
(i); // 输出 0, 1, 2 } // console.log(i); // ReferenceError: i is not defi
ned, 因为 i 在这里不可见
```

使用`let`可以让你更精确地控制变量的作用范围，从而减少错误和使代码更易于维护。

2. 箭头函数（Arrow Functions）

解决的痛点: 简洁的语法和词法作用域的`this`

```
const add = (a, b) => { return a + b; } function add(a, b) { return a +
b; } var add = function(a, b) { return a + b; }
```

单行 return 可以被省略

```
const add = (a, b) => a + b const add = (a, b) => { console.log(a, b);
return a + b; } const add = (a, b) => console.log(a, b); return a + b; //
不合法
```

使用箭头函数，你可以避免`this`的问题，因为箭头函数中的`this`会永远指向定义时的上下文。

```
function Person() {  this.age = 0;  setInterval(() => {  this.age++;  //
`this` 正确地指向了 Person 实例  }, 1000); }  var p = new Person(); functio
n Person() {  this.age = 0;  setInterval(function growUp() {  this.age+
+;  // 这里的 `this` 指向全局对象，不是 Person 实例  }, 1000); }  var p = ne
w Person()
```

3. 模板字符串（Template Literals）

解决的痛点: 字符串拼接和多行字符串

ES6

```
const text = `Hello ${name}`; const text = `Hello, my name is ${name}, ${age} years old, ${title} at ${company}`
```

非ES6

```
// Hello, my name is Long Zhao, 32 years old, Enginering Lead at Qantas var name = 'Long Zhao'; var age = 32; var title = 'Enginering Lead' var company = 'Qantas' var test = 'Hello, my name is ' + name + ', ' + age + ' years old, ' + title + ' at ' + company var text = "Hello " + name;
```

```
var multi = 'line1, /n line2' const multi = `line1 line2 line3 line4 `
```

4. 解构赋值（Destructuring）

解决的痛点: 繁琐的对象和数组元素提取

ES6

```
const { a, b } = obj; const [c, d] = arr; var student = { id: 'S0', name: 'Alice', age: 23, address: { line1: '', line2: '', city: 'Melbourne', state: 'VIC', postCode: '3000', country: { code: 'AU', name: 'Australia', }, }, } const { id: studentId, address: { line1, line2, city, state, postCode, country: { code, name }, } ...rest } = student; console.log(rest) const fruits = ['apple', 'banana', 'watermalone']; const [a, b, c] = fruits; const [a, ...rest] = fruits; console.log(a) // apple console.log(rest) // ['banana', 'watermalone']
```

非ES6

```
var student = { id: 'S0', name: 'Alice', age: 23, address: { line1: '', l
ine2: '', city: 'Melbourne', state: 'VIC', postCode: '3000', country: { c
ode: 'AU', name: 'Australia', }, }, } var studentId = student.id var name
= student.name var age = student.age var address = student.address var a
= obj.a; var b = obj.b; var c = arr[0]; var d = arr[1];
```

5. 默认参数（Default Parameters）

解决的痛点: 参数默认值的繁琐设置

ES6

```
function foo(a = 10) { return a } console.log(foo()) // 10 console.log(fo
o(5)) // 5
```

非ES6

```
function foo(a) { if (!a) { a = 10 } return a }
```

JavaScript 版本升级与 ECMA

JavaScript 语言的标准是由 ECMA International（欧洲电脑制造商协会）通过其技术委员会（TC39）管理的。这个标准被称为 ECMAScript。

1. **为什么不能硬升级**：因为网上有太多旧的网站和代码，硬升级会让这些都出问题。另外，不同的浏览器和运行环境也会受影响。
2. **提案是什么**：通过提案（或建议）的方式，大家可以慢慢适应新功能，不会一下子弄坏现有的代码。
3. **ECMA 的角色**：ECMA 是一个组织，负责管理 JavaScript（其实叫 ECMAScript）的标准。新功能要经过多个步骤和测试，才会被加入标准。

面试题：我们为什么需要 ES6，ES6给我们带来的主要好处是什么？

ES6 主要的提升的是引进了很多新的语法糖而不是(性能提升/运行效率)，比如 classname，解构赋值，默认参数和模版字符串，这些提升旨在提高 JavaScript 的**代码质量**，通过新的语法糖，提高代码1.**可读**，2.**可复用**和3.**可维护性**，从而提高 JavaScript 的开发和维护效率。

课后作业

实现 `calculateTax`

1. 已知税表 https://www.ato.gov.au/Rates/Individual-income-tax-rates/
2. 计算输入的 Personal Income 所需要缴纳的税

```
// 你的朋友今天过来问：你知道澳大利亚个人所得税是怎么算的吗？我新找的一份工作年薪是 6w5，我要交多少税 // 你的大脑飞速运转：1. 先找到这个表 2. 在表中找到相应的范围 3. 计算 // 你：(65000 - 45000) * 0.325 + 5092 function calculateTax(income) { // 先找到这个表，array const TAX_TABLE = [ { min: 0, max: 18200, rate: 0, base: 0 }, { min: 18200, max: 45000, rate: 0.19, base: 0 }, { min: 45000, max: 120000, rate: 0.325, base: 5092 }, { min: 120000, max: 180000, rate: 0.37, base: 29467 }, { min: 180000, max: Number.POSITIVE_INFINITY, rate: 0.45, base: 51667 }, ] // 在表中找到相应的范围 // const row = TAX_TABLE.find(({ min, max }) => income > min && income <= max) let row for (i = 0; i <= TAX_TABLE.length; i ++) { if (income > TAX_TABLE[i].min && income <= TAX_TABLE[i].max) { row = TAX_TABLE[i] break } } // 计算 const { base, rate, min } = row const tax = (income - min) * rate + base return tax } calculateTax(210000) // incomeTax calculateTax(income) { let tax = 0; if (income <= 18200) { tax = 0; } else if (income <= 45000) { tax = (income - 18200) * 0.19; } else if (income <= 120000) { tax = 5092 + (income - 45000) * 0.325; } else if (income <= 180000) { tax = 29467 + (income - 120000) * 0.37; } else { tax = 51667 + (income - 180000) * 0.45; } return tax; } const TAX_RATE_TABLE = { 18200:{'base': 0, 'rate':0.19}, 45000:{'base': 0, 'rate':0.325}, 120000:{'base': 5092, 'rate':0.37}, 180000:{'base': 29467, 'rate':0.45}, 180001:{'base': 51667, 'rate':0.45} }; function calculateTax(income) { let tax = 0; for (let [threshold, { base, rate }] of Object.entries(TAX_RATE_TABLE)) { if (income > threshold) { tax = base + (income - threshold) * rate; } } return tax; }
```

实现 `getStops`

1. 给定输入的行程数据
2. 计算相应的 Stops

```
function getStops(flights) { // ... } const flights = [{ origin: 'MEL', destination: 'PVG' }]; getStops(flights) // Direct const flights = [{ origin: 'MEL', destination: 'HKG' }, { origin: 'HKG', destination: 'PVG' }]; getStops(flights) // 1 Stop getStops(flights) // 2 Stops -> n Stops // Multi City [MEL - SIG, SIG - HKG, HKG - PVG, PVG - HND, ...] // 12 STOPS -> Dream Trip // 24 STOPS -> Arount the World // 20 -> A // 21 -> C // 50 -> ABC // 100个变种 // A function getStops(flights) { const stops = flights.length - 1 if (stops === 0){ return 'Direct'; } else if (stops === 1){ return "1 Stop" } else if (stops === 12) { return "Dream Trip" } else if (stops === 24) { return "Around the World" } else { return `${stops} Stops` } } // B // SOD // 符合人类思考方式 // 是不是单一原则，可不可以把这个代码分成多块代码区域，每块代码有自己的责任，同时不同块代码的责任互相不重复 // 是不是开关原则，可不可以找到核心代码块，后续扩展可不可以不碰核心代码块 // 是不是依赖注入原则，有没有依赖，依赖是不是注入进来的？ // 开放的逻辑 const STOPS_MAP = { 0: 'Direct', 1: 'Stop', 12: 'Dream Trip', 24: 'Around the World', } // 不同的 stops 给 不同的结果 // key - value // Object // 表 // Array function getStops(f) { const stops = f.length - 1 // 找到 stops const result = STOPS_MAP[stops] || `${stops} Stops`; // 封闭的逻辑，谁都不能改这个逻辑 return result } // C function getStops(flights) { switch(flights.length) { case 1: return 'Direct' case 2: return '1 Stop' case 12: return 'Dream Trip' case 24: return 'Around the World' default: return `${flights.length - 1} Stops` } }
```

```
const plural = (string, count) => { if (count > 1) { return `${string}s` } return string } // 字典数据 const SPECIAL_STOPS_MAPPING = { 0: 'Direct', 12: 'Dream Trip', 24: 'Around the World', } const getStops = (flights) => { if (!isValid(flights) { return 'N/A' } const stops = flights.length - 1; return SPECIAL_STOPS_MAPPING[stops] || `${stops} ${plural('Stop', stops)}` }
```

编程世界的黄金法则：要写的像诗一样优雅！

- 可读
- 可复用