

第 3 章



JavaScript 基础

3.1 JavaScript 简介

在万维网发展的早期,Web 网页是由文字和图片构成的静态文档,缺乏在浏览器中加载网页后的动态交互能力。1995 年,网景公司工程师布兰登·艾克开发了能够在网页中动态组装图片和文字的 LiveScript 脚本语言(最初代号 Mocha),并将其嵌入网景公司的 Navigator 浏览器中。

在 LiveScript 语言的基础上,网景公司在 1995 年 9 月推出了 JavaScript 脚本语言,它赋予了浏览器更强大的页面交互能力,JavaScript 语言被默认集成到 Navigator 浏览器的 2.0 版本中。JavaScript 脚本语言的出现,使得静态网页开始真正向动态化方向发展。

同年,微软公司推出了 Internet Explorer 浏览器,引发了微软与网景公司的浏览器大战。微软对 Navigator 解释器进行逆向工程后,开发了 JScript 脚本。JScript 也是 JavaScript 的一种实现,这两个语言版本在浏览器端共存意味着 JavaScript 在发展初期缺乏统一的语言标准,同期就有网景的 JavaScript 和微软的 JScript。除此之外,微软也在网页技术上加入了不少专属对象,使得不少网页在非微软平台及浏览器中无法正常显示。

1997 年,网景公司将 JavaScript 提交给欧洲计算机制造商协会(European Computer Manufacturers Association, ECMA),随后该组织发布了 ECMAScript 标准,这也是 JavaScript 标准化的第一个版本。1999 年,ECMA 发布了 ECMAScript 3 版本,这一版本为 JavaScript 的发展奠定了坚实的基础,引入了正则表达式、异常处理等新特性。随着时间的推移,ECMAScript 标准不断演进,JavaScript 语言也逐渐成熟,伴随着 Node.js 等服务器端运行环境的出现,JavaScript 也从服务于前端的脚本语言,发展成为一门支持多平台的大型应用程序开发语言。

作为一种跨平台、面向对象的轻量级编程语言,JavaScript 主要用于创建动态和交互式的网页内容,是现代前端开发的核心技术之一,与 HTML、CSS 并称为前端开发的三大基石。作为一门多范式的编程语言,JavaScript 支持面向对象、函数式和指令式编程等多种编程范式,核心特性说明如下。

- 脚本语言：在浏览器中运行，用于精准控制和修改网页内容。
- 解释型语言：无须编译成机器码，由浏览器或 JavaScript 引擎直接解析执行。
- 弱类型语言：变量类型可动态变化，声明变量时无须指定类型。
- 多范式：支持多种编程风格，包括面向对象、指令式和函数式编程。
- 跨平台：可在各种 Web 浏览器和服务器环境中运行。
- 事件驱动：在浏览器中通过事件机制响应用户操作，实现交互逻辑。

总的来说，JavaScript 作为一种轻量级、高级的脚本语言，主要用于创建动态和交互式的网页内容，并且已经扩展到服务器端、移动应用和游戏开发等领域。作为 Web 开发的核心技术之一，它与 HTML 和 CSS 共同协作：HTML 负责定义网页的内容结构，CSS 处理样式和布局，而 JavaScript 则控制网页的行为和交互。无论是前端、后端还是跨平台开发，JavaScript 都是现代开发者必须掌握的基础工具。

3.2 JavaScript 语法基础

3.2.1 变量与数据类型

变量是用于存储数据的容器，JavaScript 使用 `var`、`let` 来声明变量。`var` 和 `let` 声明的变量最大的区别是作用域不同，使用 `var` 声明的变量具有函数作用域（在全局环境下为全局作用域），使用 `let` 声明的变量具有块级作用域。变量声明后，可以使用等号（`=`）为其赋值，也可以在声明变量时就进行赋值，如下所示。

```
//声明并赋值
var name = "Alice";
let age = 30;
const PI = 3.14;

//先声明后赋值
var city;
city = "New York";
```

除此之外，JavaScript 声明常量需要用到 `const` 关键字，它具有块级作用域，其值在声明后不能被重新赋值。

JavaScript 的数据类型共分为八种，分别是基本数据类型 `String`、`Number`、`Boolean`、`Undefined`、`Null`、`Symbol`、`BigInt` 和引用数据类型 `Object`。其中，基本数据类型是不可变的，其值存储在栈内存中，如下所示。

```
const age = 25;           //Number 类型
const name = "Alice";     //String 类型
const hasPermission = false; //Boolean 类型
let x;                    //Undefined 类型
```

```
const emptyValue = null;           //Null 类型
const id = Symbol("id");           //Symbol 类型
const bigNum = BigInt(1234)        //BigInt 类型
```

引用数据类型的值是可变的,存储在堆内存中,变量仅存储指向堆内存的引用,赋值时复制的是引用地址。JavaScript 的引用数据类型为 Object,其内置子类型主要包括 Array 和 Function。其中,Object 是最基本的引用类型,用于存储键值对集合,如下所示。

```
const person = {
  name: "Tom",
  age: 30,
  isStudent: false
};
```

Array 则属于有序的元素集合,元素可以是任意数据类型,通过索引访问,如下所示。

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]);           //输出 apple
```

Function 表示可执行的代码块,也是对象的一种,可作为参数或返回值出现,如下所示。

```
function add(a, b) {
  return a + b;
}
console.log(add(2, 3));           //输出 5
```

除此之外,在 JavaScript 语言中,诸如 Date、RegExp、Map 和 Set 也可以理解成是一个对象,它们共同丰富了 JavaScript 语言的数据类型。

3.2.2 运算符

JavaScript 运算符是用于执行变量或值之间运算的特殊符号,按照其功能大体可以分为算术运算符、赋值运算符、比较运算符、逻辑运算符、位运算符和其他特殊运算符六类。其中,算术运算符是用于执行基本数学运算的操作符,操作数为数字类型,包括 +、-、*、/、%、++、-- 等,如下所示。

```
let a=2; a++                       //自加
let b=5; b--                       //自减
7 % 3                              //取余
```

需要说明的是,自增、自减运算符遵循以下规则:前置时先运算后取值,后置时先取值后运算。

赋值运算符主要用于给变量赋值,可结合算术运算符简化写法,常见的赋值运算符包括`=`、`+=`、`-=`、`*=`、`/=`和`%=`等几种,如下所示。

```
let x = 10;
let y = 5;
x += y;                                     //输出 15
```

比较运算符主要用于比较两个值,返回值为布尔值。常见的比较运算符包括`==`、`!=`、`>`、`<`、`>=`、`<=`等几种,如下所示。

```
5 !== 5                                     //false
5 >= 5                                       //true
5 != 3                                       //true
```

逻辑运算符主要用于组合或反转布尔值,返回值为布尔值或运算的结果。常见的逻辑运算符包括`&&`、`||`和`!`等,如下所示。

```
let x=6;
let y=3;
x == 5 || y == 5                             //输出 false
x < 10 && y > 1                               //输出 true
```

位运算符主要用于对数字的二进制位进行操作,日常开发中的使用频率较低。常见的位运算符包括`&`、`^`、`~`、`<<`、`>>`等,如下所示。

```
3 & 5                                         //011 & 101 = 001 → 1
3 ^ 5                                         //011 ^ 101 = 110 → 6
3 << 1                                        //011 << 1 = 110 → 6
6 >> 1                                        //110 >> 1 = 011 → 3
```

需要说明的是,在 JavaScript 中使用运算符需要遵循运算符的优先级,优先级高的运算符会先执行,最高优先级包括圆括号、成员访问和 `new` 运算符。

3.2.3 表达式

在 JavaScript 中,表达式是由变量、常量、运算符、函数等组合而成的代码片段,其核心特征是能被求值并返回结果。表达式是构建 JavaScript 逻辑的基础,常见于变量赋值、条件判断、函数参数等场景。

根据组成和功能的不同,JavaScript 的表达式大体可以分为原始表达式、算术表达式、字符串表达式、逻辑表达式、比较表达式、函数调用表达式、对象与数组表达式以及条件表达式等几种。其中,原始表达式是最简单的表达式,能够直接返回自身的值,无须计算,如下所示。

```
"hello";           //字符串字面量
true;              //布尔字面量
this;              //返回上下文对象
```

算术表达式是由算术运算符连接而成的表达式,默认返回计算后的数值,如下所示。

```
20 - 3 * 4;         //混合运算,返回 8
(100 / 2) % 7;      //取余,返回 1
```

逻辑表达式是由逻辑运算符连接而成的表达式,返回布尔值或运算的实际结果,如下所示。

```
10 > 5 || 3 > 8;     //逻辑或,返回 true
!(2 === "2");       //逻辑非,返回 true
0 || "default";     //返回 "default"
```

函数调用表达式表示调用函数时的表达式,默认返回函数的执行结果,若无返回则默认返回 undefined,如下所示。

```
Math.sqrt(16);      //调用内置函数
function add(a, b) { return a + b; }
add(2, 3);          //调用自定义函数
```

条件表达式又名三元表达式,是根据条件返回两个表达式中一个结果的表达式,如下所示。

```
let score = 85;
let result = score >= 60 ? "B" : "C";
```

总的来说,表达式是指一段可以计算出结果并执行相应操作的代码。理解不同类型的表达式及其组合规则,是编写高效代码的基础。实际项目开发中,合理拆分复杂表达式可提高代码的可阅读性和可维护性。

3.3 程序控制结构

JavaScript 的程序控制结构用于控制代码的执行顺序,包括顺序结构、分支结构和循环结构三种基本类型。

3.3.1 顺序结构

顺序结构是最基础的程序控制结构,代码按照从上到下的顺序依次执行,是程序默认的执行方式。例如,下面的 JavaScript 代码中,所有的语句都按照顺序从上到下执行。

```
let a = 10;
let b = 20;
let sum = a + b;
console.log("sum =", sum);
```

上面的这段代码,每一行都按照其在文件中的位置,从上到下依次执行。这种最直观的逐行执行方式,便是程序开发中最基础的顺序结构。

3.3.2 分支结构

JavaScript 的分支结构是根据条件判断执行不同的代码块的程序流程控制方式,常见的分支结构包括 if...else 语句、switch 语句和三元运算符,它们适用于各种条件判断场景。

if...else 语句是最常用的条件分支结构,它根据条件表达式的真假来决定是否执行代码块。if...else 语句根据其判断条件,又可以细分为单条件、多条件和嵌套判断等场景。单条件语句是最基础的分支结构,它根据布尔表达式的值来决定是否执行一段代码,如下所示。

```
let score = 80;
if (score >= 60) {
    console.log("及格");
}
```

当 if 语句的条件不满足时,可以使用 else 块来执行另一段代码,如下所示。

```
let age = 16;
if (age >= 18) {
    console.log("需要购票");
} else {
    console.log("需成人陪同");
}
```

当有多个条件需要判断时,可以使用 else if 语句分支来逐一进行判断,最后用 else 来处理所有条件都不满足的情况,如下所示。

```
let score = 82;
if (score >= 90) {
    console.log("优秀");
} else if (score >= 80) {
    console.log("良好");
} else if (score >= 60) {
    console.log("及格");
} else {
    console.log("不及格");
}
```

switch 语句适用于根据一个表达式的不同值来执行不同的代码块,通常用于处理多个固定选项的场景,如下所示。

```
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("新的一周开始");
    break;
  case "Friday":
    console.log("马上周末了");
    break;
  default:
    console.log("工作日");
}
```

需要说明的是,case 语句使用严格相等(===)来比较,如果省略 case 语句中的 break 会导致程序的穿透,即继续执行下一个 case 语句。

三元运算符是 if...else 语句的简洁表达式写法,它使用一个条件表达式来选择两个值中的一个,适用于简单判断的场景,如下所示。

```
let isMember = true;
let discount = isMember ? 0.8 : 1;
```

总的来说,if...else 语句提供单分支、双分支和多分支的判断执行逻辑,而 switch 语句则适用于根据一个变量的多个可能值来执行不同代码块的场景。

3.3.3 循环结构

循环结构用于重复执行某段代码,直到满足终止条件,循环结构由循环条件和循环体构成。JavaScript 的循环结构包括 for 循环、while 循环、do...while 循环、for...in 循环和 for...of 循环。for 循环是最常用的循环结构,适合已知循环次数的场景,语法清晰明了,如下所示。

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

while 循环适用于循环次数不确定的场景,需要先判断条件后再决定是否执行循环体,如下所示。

```
let sum = 0;
let i = 1;
while (i <= 100) {
  sum += i;
}
```

```
    i++;  
  }  
  console.log("sum:", sum);
```

do...while 循环与 while 类似,但它会先执行一次循环体,然后再判断条件,如下所示。

```
let num;  
do {  
  num = prompt("请输入大于 10 的数字:");  
} while (num <= 10);
```

for...in 循环适用于遍历对象的可枚举属性,不推荐用于数组场景,如下所示。

```
const person = { firstName: "John", lastName: "Doe" };  
for (let key in person) {  
  console.log(person[key]);  
}
```

除此之外,ES6 还新增了 for...of 循环,用于遍历可迭代对象(如数组、字符串、Map、Set 等)的值,如下所示。

```
const fruits = ["Apple", "Banana", "Orange"];  
for (let fruit of fruits) {  
  console.log(fruit);  
}
```

总的来说,JavaScript 的程序控制结构由顺序、分支、循环三大结构构成。顺序结构是默认的执行方式,按从上到下的代码顺序执行;分支结构用于处理选择逻辑的场景,包括 if...else 语句、switch 语句;循环结构则用于处理重复逻辑的场景,包括 for 循环、while 循环、do...while 循环等。

3.4 函数

3.4.1 函数定义

函数是 JavaScript 最基本的组成部分之一,是一段可重复使用的代码块,常用于执行特定的任务或计算值。在 JavaScript 中,有多种方式来定义函数,常见的有使用 function 关键字声明函数、使用函数表达式来声明函数,其中 ES6 新增的箭头函数是函数表达式的一种简洁语法形式。

使用 function 关键字来声明函数是函数定义最常见的方式,具有函数提升特性,即可以在函数声明前调用,如下所示。

```
function square(number) {  
    return number * number;  
}  
console.log(square(3))
```

函数表达式是将函数赋值给变量的函数定义方式,必须先定义后才能调用,如下所示。

```
const subtract = function(a, b) {  
    return a - b;  
};  
console.log(subtract(5, 2));
```

箭头函数是 ES6 引入的匿名函数表达式语法,无参数对象,不能作为构造函数使用,如下所示。

```
const multiply = (a, b) => a * b;  
console.log(multiply(3, 4));
```

3.4.2 函数作用域

函数的作用域是指在函数内部创建的变量的可见范围,该范围内的变量只能在函数内部访问,无法在外部访问。函数作用域由函数定义并由函数内的代码块组成,是 JavaScript 最初的两种作用域类型之一,另一种是全局作用域。在 ES6 引入块级作用域之前,JavaScript 使用 var 关键字声明的变量具有函数作用域的特性,如下所示。

```
var num1 = 20;  
var num2 = 3;  
  
function multiply() {  
    return num1 * num2;  
}  
console.log(multiply());
```

定义在全局作用域中的函数可以访问所有定义在全局作用域中的变量。在其他函数中定义的函数,也可以访问其父函数作用域中的变量,以及父函数能够访问的其他任何变量。

3.4.3 函数参数

JavaScript 函数的参数是函数与外部交互的接口,用于接收外部传入的数据。JavaScript 的参数分为形参和实参两种,形参是函数定义时声明的变量,实参则是调用函数时传入的值,如下所示。

```
function add(a, b) {                                //形参 a, b
    return a + b;
}

add(2, 3);                                           //实参 2, 3
```

实际使用过程中,形参和实参的数量可以不匹配,当实参多于形参时,多余的实参不会被形参接收,但可通过函数的 `arguments` 对象获取;当实参少于形参时,未被传入的形参值默认为 `undefined`。

在 ES6 语法中,JavaScript 支持在函数形参中直接设置默认值的方式,避免了在函数体内部进行 `undefined` 的判断,如下所示。

```
function greet(name = "Guest", message = "Hello") {
    console.log(`${message}, ${name}!`);
}

greet();
greet("Alice");
greet("Bob", "Hi");
```

除此之外,在不确定参数的数量时可以将参数打包成一个数组,这也是 ES6 支持的剩余参数语法,语法格式为“...变量名”,如下所示。

```
function sum(...numbers) {
    return numbers.reduce((total, n) => total + n, 0);
}

console.log(sum(1, 2));
console.log(sum(1, 2, 3, 4));
```

需要说明的是,普通函数内部有一个 `arguments` 对象,是包含所有实参的类数组对象,可以对其执行遍历操作,以获取传入参数的值,如下所示。

```
function logArgs() {
    console.log(arguments);                //[1, "a", true]
    console.log(arguments.length);         //3
    console.log(arguments[0]);             //1
}

logArgs(1, "a", true);
```

JavaScript 函数的参数传递方式为按值传递,对于基本数据类型,传递的是其值的副本;对于引用类型,传递的是引用地址的副本。这意味着在函数内部修改基本类型参数的值

不会影响外部变量,但修改引用类型参数所指向对象的属性会影响外部对象,如下所示。

```
//基本数据类型,按值传递
let a = 10;
function changeNum(x) {
  x = 20;
}
changeNum(a);
console.log(a); //输出 10

//引用类型,传递地址副本
let obj = { name: "Alice" };
function changeObj(o) {
  o.name = "Bob";
  o = { name: "Charlie" };
}
changeObj(obj);
console.log(obj.name); //输出 Bob
```

总的来说,JavaScript 的函数参数具有高度灵活性,支持默认值、剩余参数等特性,能够适应各种场景的参数传递需求。理解参数的传递方式和特殊对象,能够帮助我们写出简洁、灵活的函数。实际开发中,合理使用剩余参数,可以显著提升代码的可读性和易用性。

3.4.4 函数返回值

函数返回值是函数执行完成后输出的结果,通过 `return` 语句进行指定。JavaScript 的函数返回值是函数与外部环境交互的重要方式,决定了函数调用后能得到的数据。在 JavaScript 语法中,当函数执行到 `return` 语句时,函数会停止运行,并将 `return` 后面的值作为函数的最终结果返回给调用者,如下所示。

```
function add(a, b) {
  return a + b;
}

const result = add(2, 3);
console.log(result); //输出 5
```

如果没有 `return` 语句,或者 `return` 后不跟任何值,函数将默认返回 `undefined`,如下所示。

```
function greet() {
  console.log("Hello");
}

const res = greet();
console.log(res); //输出 undefined
```

函数的返回值可以是任何数据类型,包括基本数据类型、引用类型,甚至是函数本身,如下所示。

```
function createAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
  
const add5 = createAdder(5);  
console.log(add5(3));           //输出 8  
console.log(add5(10));          //输出 15
```

需要说明的是,在箭头函数中返回对象时,需使用圆括号包裹对象字面量,目的是避免与函数体`{}`的混淆。

```
//错误写法: {}会被解析为函数体  
const createObj = () => { name: "Alice" };  
console.log(createObj());  
  
//正确写法: 用 () 包裹对象字面量  
const createObj = () => ({ name: "Alice" });  
console.log(createObj());
```

此外,如果函数返回的是有意义的值,务必显式使用 `return` 语句,否则可能导致逻辑错误。

```
function getRealValue() {  
  "Hello";  
}  
console.log(getRealValue());           //输出 undefined
```

3.5 数据结构

JavaScript 的集合是用于存储一组有序或无序数据的数据结构,主要包括数组(Array)、对象(Object)以及 ES6 新增的 Map、Set 等。

3.5.1 数组

数组是一种有序的集合数据结构,用于存储多个元素,元素可以是任意数据类型。作为一种特殊的对象,数组具有有序性、异构性和可变性等特征。可以通过索引访问数组中的元素。

在 JavaScript 语言中,最简洁的数组定义方式是使用方括号[]进行定义,如下所示。

```
const emptyArr = [];           //空数组
const numbers = [1, 2, 3, 4];  //包含元素的数组
const mixed = [1, true, { name: "Alice" }]; //异构数组
```

除此之外,也可以使用 new Array()方式创建一个数组,适合已知长度或动态生成数组的场景,如下所示。

```
const arr1 = new Array(5);      //长度为 5 的空数组
const arr2 = new Array(1, 2, 3); //等价于 [1, 2, 3]
```

在 ES6 语法中,还可以使用 Array.of()方式来创建一个包含指定元素的数组,以解决 Array 构造函数对单个数字参数的特殊处理问题,如下所示。

```
Array.of(5);                    //等价于 [5]
Array.of(1, 2, 3);              //等价于 [1, 2, 3]
```

当需要访问与修改数组中的元素时,可以通过索引进行操作,超出索引范围的访问将返回 undefined,如下所示。

```
const fruits = ["apple", "banana", "cherry"];

console.log(fruits[0]);          //输出 apple
console.log(fruits[3]);          //输出 undefined

fruits[1] = "grape";
console.log(fruits);             //输出 ["apple", "grape", "cherry"]
```

可以通过 length 属性来获取数组元素的个数,同时可以通过修改 length 的大小来截断或扩展数组,如下所示。

```
const arr = [1, 2, 3];
console.log(arr.length);

//截断数组
arr.length = 2;
console.log(arr);                //输出 [1, 2]

//扩展数组
arr.length = 5;
console.log(arr);                //输出 [1, 2, empty × 3]
```

数组提供了大量内置方法用于修改原数组和返回新数组。其中,修改原数组的方法包

括 `push()`、`pop()`、`shift()`、`splice()`、`reverse()`、`sort()`等。使用 `push()` 往数组的末尾添加元素的示例,如下所示。

```
const arr = [1, 2];
arr.push(3, 4);
console.log(arr);                                //输出 [1, 2, 3, 4]
```

不修改原数组,返回一个新数组的方法包括 `slice()`、`concat()`、`map()`、`filter()`、`reduce()`、`flat()`等。使用 `concat()` 合并数组的示例,如下所示。

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const merged = arr1.concat(arr2, 5);
console.log(merged);                             //输出 [1, 2, 3, 4, 5]
```

除此之外,数组还提供了查询、遍历等操作方法。总的来说,数组作为常见的集合数据结构,支持动态操作、多种遍历方法,掌握数组的核心 API 对于处理集合数据非常重要。

3.5.2 对象

对象是一种包含一系列键值对的集合,用于存储数据和构建更复杂的实体,JavaScript 的对象是构建复杂数据结构和实现面向对象编程的基础。对象中的键称为属性名,值称为属性值,值可以是任意数据类型,包括基本类型、函数和其他对象等。对象最直观的定义方式是使用花括号 `{}` 进行定义,如下所示。

```
const person = {
  name: "Alice",
  "age": 20,
  address: { city: "Beijing" },
  sayHello: function() {
    console.log(`Hello, I'm ${this.name}`);
  }
};
```

当然,也可以使用 `new Object()` 构造函数的方式来创建一个对象,适合动态生成对象的场景,如下所示。

```
const obj = new Object();
obj.name = "Bob";
obj.age = 22;
obj.sayHi = function() {
  console.log("Hi");
};
```

除此之外,对于使用 class 定义的类,还可以通过 new 关键字实例化对象,如下所示。

```
class Car {
  constructor(brand, price) {
    this.brand = brand;
    this.price = price;
  }

  drive() {
    console.log(`${this.brand} is driving`);
  }
}

const myCar = new Car("Tesla", 259000);
myCar.drive();
```

当需要操作对象的属性时,可以先获取对象的属性再进行操作。获取对象的属性有两种方式,一种是使用点语法,另一种是使用方括号语法,如下所示。

```
const person = { name: "Alice", "full name": "Alice Smith" };
console.log(person.name);           //点语法
console.log(person["name"]);        //方括号语法
```

添加或修改对象属性的值时,只需要将值直接赋给属性即可,如下所示。

```
const obj = { a: 1 };
obj.b = 2;
obj.a = 100;
console.log(obj);           //输出{ a: 100, b: 2}
```

如果想要检查对象的属性是否存在,可以使用 in 运算符或使用 hasOwnProperty() 方法,如下所示。

```
const obj = { a: 1 };
console.log("a" in obj);
console.log(obj.hasOwnProperty("a"));
```

在 JavaScript 语法中,对象是可以继承的。在继承关系中,每个对象都有一个隐式的原型对象,对象可访问其原型对象上的属性和方法,多层原型的引用关系会形成原型链,如下所示。

```
const parent = { greet: () => console.log("Hello") };
const child = Object.create(parent);
child.name = "Alice";
```

```
child.greet();  
console.log(child.name);
```

除此之外,ES6 还支持复制对象属性和合并对象,如下所示。

```
const obj1 = { a: 1, b: 2 };  
  
const obj2 = { ...obj1 };           //浅拷贝  
const obj3 = { ...obj1, b: 3, c: 4 }; //合并对象  
console.log(obj3);                 //输出{ a: 1, b: 3, c: 4 }
```

可以发现,对象作为一种键值对的存储结构,可以通过对象字面量、类或构造函数进行创建,常用操作包括属性的访问、修改、删除和遍历等。对象作为一种基础的数据结构,是开发者必须掌握的基础知识。

3.5.3 Map

Map 是 ES6 新增的键值对存储数据结构,类似于其他编程语言的哈希表或字典。它与对象的主要区别在于,Map 的键可以是任意类型,而对象的键只能是字符串或 Symbol 类型;同时 Map 是有序的,而对象的属性顺序则是不确定的。

在 JavaScript 语言中,创建 Map 需要使用 `new Map()` 构造函数,创建 Map 时可以传入一个二维数组进行初始化,如下所示。

```
const map = new Map();  
  
const initMap = new Map([  
  ["name", "Alice"],  
  [123, "number"],  
  [true, "boolean"]  
]);  
console.log(initMap.size);
```

获取存储在 Map 中的值需要传入对应的键,若键名不存在则默认返回 `undefined`,如下所示。

```
const map = new Map([["name", "Bob"]]);  
console.log(map.get("name"));  
console.log(map.get("age"));           //返回 undefined
```

添加或修改键值对需要用到 `set()` 方法,该方法会返回 Map 对象本身。添加或修改键值对时,如果键已存在则覆盖原有值,如下所示。

```
const map = new Map();
```

```
map.set("age", 20);
map.set({ id: 1 }, "user");
map.set("gender", "female").set("hobby", "coding");
map.set("age", 21);
```

如果需要删除键值对,只需要调用 `delete()` 方法传入需要删除的键即可,删除键值对的返回值为布尔类型,成功删除返回 `true`,键名不存在返回 `false`,如下所示。

```
const map = new Map([["name", 20]]);
map.delete("name");
console.log(map.has("name"));
```

作为一种集合数据类型,Map 也提供了多种遍历方式,常见的有 `for...of` 循环、`forEach` 循环,均按插入顺序执行。其中,`for...of` 循环可遍历 `entries()`、`keys()` 或 `values()` 返回的迭代器,如下所示。

```
const map = new Map([
  ["name", "Alice"],
  ["age", 20]
]);
//遍历键值对
for (const [key, value] of map) {
  console.log(`${key}: ${value}`);
}
//遍历键
for (const key of map.keys()) {
  console.log(key);
}
//遍历值
for (const value of map.values()) {
  console.log(value);
}
```

在实际的项目开发中,还经常会碰到 Map 与数组之间相互转换的场景,如下所示。

```
const map = new Map([["a", 1], ["b", 2]]);
const arr = Array.from(map);
const newMap = new Map(arr);
```

总的来说,Map 作为一种灵活高效的键值对集合,相比对象类型来说,提供了更高效的键值查找功能,适合用在动态键值对、非字符串键或对插入顺序有严格要求的开发场景。

3.5.4 Set

Set 是 ES6 新增的数据结构,其核心特性是只存储唯一值,不允许重复的元素存在。与

数组相比,Set 适合用来处理需要去重或快速判断元素是否存在的场景。

在 JavaScript 中,创建 Set 需要使用 `new Set()` 构造函数,创建时可以传入一个可迭代对象进行初始化,如下所示。

```
const set = new Set();
const numSet = new Set([1, 2, 3, 3, 4]);
console.log(numSet.size);           //输出 4
```

创建 Set 时,如果传入的数组包含重复的元素,Set 会自动去重,因此上面代码中 numSet 的大小为 4。

如果需要往 Set 中添加元素,那么只需要调用 `add()` 方法即可, `add()` 方法默认会返回 Set 本身,并且插入的顺序和存储的顺序是一致的,如下所示。

```
const set = new Set();
set.add("apple");           //单个添加
set.add(true).add({ name: "Alice" }); //链式添加
```

如果需要删除 Set 中的元素,那么只需要调用 `delete()` 方法传入需要删除的元素即可, `delete()` 方法返回值为布尔类型,成功删除返回 `true`,元素不存在则返回 `false`,如下所示。

```
const set = new Set([1, 2, 3]);
set.delete(2);
console.log(set.has(2));
```

Set 提供了多种遍历方式,如 `for...of` 循环、`forEach` 循环,循环时均按插入顺序执行。下面是使用 `forEach` 循环遍历 Set 元素的示例。

```
const set = new Set([10, 20, 30]);
set.forEach((value, key, set) => {
  console.log(`value:${value},key:${key}`);
});
```

同样的,Set 和数组也可以相互转换,利用这一特性可快速实现数组去重需求,如下所示。

```
const arr = [1, 2, 2, 3, 3, 3];
const set = new Set(arr);
//Set 转换为数组
const newArr = Array.from(set);
```

总的来说,Set 是 JavaScript 中用于存储唯一值的高效数据结构,具有自动去重、有序和快速查找等特性,特别适合用于数据去重、元素存在性判断等开发场景。

3.6 类

3.6.1 创建类

类是 ES6 引入的语法糖,用于实现面向对象编程(Object-Oriented Programming, OOP),本质上是对 JavaScript 构造函数与原型链继承模式的封装,让 JavaScript 代码结构更清晰、更符合传统面向对象语言的编程范式。

JavaScript 使用 `class` 关键字来定义类,其内部可以包含构造函数、方法和属性等成员,如下所示。

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
  
  getInfo() {  
    console.log(`I'm ${this.name}, ${this.age} years old`);  
  }  
}
```

在面向对象编程规范中,可以使用 `new` 关键字来创建类的实例,在创建过程中,可以调用构造函数来初始化类的实例对象,如下所示。

```
const alice = new Person("Alice", 20);  
alice.getInfo();
```

需要说明的是,JavaScript 的类虽然看起来像 Java 或 C++ 等传统的面向对象语言,但其底层却是基于原型链实现的。

3.6.2 构造方法

类的构造方法是一个使用 `constructor` 关键字定义的特殊方法,用于初始化类的实例对象。当通过 `new` 关键字创建类的实例时,构造方法会被自动调用,用于设置实例的初始属性或执行初始化逻辑。每个类只能有一个 `constructor` 方法,如下所示。

```
class User {  
  constructor(name, age) {  
    this.name = name;           //属性 name  
    this.age = age;             //属性 age  
  }  
}
```

```
... //省略其他代码
}
```

若实例无须初始化参数,那么也可以定义无参构造方法,如下所示。

```
class Circle {
  constructor() {
    this.radius = 1;
  }
  ... //省略其他代码
}
```

在构造方法的内部, `this` 关键字用于指向正在被创建和初始化的那个新对象实例。当子类使用 `extends` 关键字继承父类时,子类的构造方法必须显式地调用 `super()` 方法。若子类未定义构造方法,系统会默认生成一个构造方法并自动调用 `super(...arguments)`。

默认情况下,构造方法返回实例对象本身。仅当手动返回引用类型数据(对象、数组、函数等)时,才会覆盖默认的实例返回值;若返回基本类型值,会被引擎忽略,依然返回实例对象,如下所示。

```
class Person {
  constructor() {
    this.name = "Jack";
    return { custom: "Custom Data" };
  }
}
```

3.6.3 继承

继承是面向对象编程的三大核心特性之一,ES6 新增了基于 `class` 与 `extends` 关键字的语法层面继承实现方式。它允许一个类继承另一个类的属性和方法,同时可以扩展或重写父类的功能,以实现代码复用和逻辑扩展。在 JavaScript 中,子类继承父类需要使用 `extends` 关键字,然后通过使用 `super()` 调用父类的构造方法。

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(`${this.name} is eating`);
  }
}
```