河南中医药大学信息技术学院（智能医疗行业学院）智能医学工程专业《互联网医疗服务开发》课程

第05章：JavaScript基础

冯顺磊

河南中医药大学信息技术学院（智能医疗行业学院）

河南中医药大学信息技术学院智能医疗教研室

https://aitcm.hactcm.edu.cn

2025/3/7

本章概要

- 概述
- 调用方法
- 基础语法
- 函数
- 对象
- DOM

1. 概述 1.1 JavaScript是什么?

- JavaScript 是一种**高级的、解释型的编程语言**，主要用于网页开发，也广泛应用于服务器端编程、移动应用开发、游戏开发等领域。

- 历史
  - JavaScript 由 Netscape 公司的 Brendan Eich 在 1995 年开发，最初被称为 Mocha，后来改名为 LiveScript，最终定名为 JavaScript。
  - 当时的主要目的是为了给网页添加动态效果和交互性，随着时间的推移，JavaScript 逐渐成为网页开发中不可或缺的一部分。
- 特点
  - **动态类型**：在声明变量时不需要指定数据类型，变量的数据类型会在运行时根据赋值自动确定。
  - **事件驱动**：可以通过监听和响应各种事件，如鼠标点击、键盘输入等来实现交互效果。
  - **跨平台**：可以在不同的操作系统和设备上运行，只要有支持 JavaScript 的浏览器或运行环境。
  - **面向对象**：支持面向对象编程，允许开发者使用对象、类、继承等概念来组织和管理代码。
  - **解释执行**：代码在运行时由解释器逐行解释执行，不需要事先编译。

1. 概述 1.2 JavaScript能实现什么?

- JavaScript 作为一种功能强大的编程语言，在多个领域都有广泛应用，能实现的功能非常丰富。

- 网页开发领域
  - **动态交互效果**：能实现网页元素的动态显示与隐藏、菜单的展开与收缩、图片的轮播切换等效果，还能让网页响应用户的鼠标操作，如鼠标悬停时显示提示信息、点击按钮触发动画等，增强用户与网页的交互性。
  - **表单验证**：在用户提交表单前，对输入内容进行合法性验证，如验证邮箱格式是否正确、密码是否符合强度要求、手机号码是否有效等，及时反馈错误信息，提高数据的准确性和完整性。
  - **页面动态更新**：无需刷新整个页面就能更新部分内容，通过 AJAX 技术与服务器进行数据交互，实现实时加载数据、动态更新页面内容，如实时显示新闻资讯、股票行情等。

1. 概述 1.2 JavaScript能实现什么?

- 服务器端编程领域
  - **构建 Web 服务器**：使用 Node.js 可以轻松搭建 Web 服务器，处理 HTTP 请求和响应，实现路由功能，根据不同的 URL 路径返回相应的内容，如构建博客网站、电子商务网站的后端服务器。
  - **数据库操作**：能与各种数据库（如 MySQL、MongoDB 等）进行交互，执行数据的增删改查操作，实现用户注册登录信息的存储和验证、订单数据的管理、文章内容的存储等功能。
  - **中间件开发**：可以开发中间件来实现日志记录、身份验证、数据缓存等功能，增强服务器的性能和安全性，提高应用的可扩展性和维护性。
- 移动应用开发领域
  - **跨平台应用开发**：利用 ReactNative 等框架，使用 JavaScript 开发跨平台的移动应用，一套代码可以同时运行在 iOS 和 Android 平台上，提高开发效率，降低开发成本。
  - **与原生功能交互**：能通过框架提供的 API 访问手机的原生功能，如摄像头、麦克风、GPS 定位、通讯录等，实现拍照、录音、导航、获取联系人等功能。

1. 概述 1.2 JavaScript能实现什么?

- 游戏开发领域
  - **2D 游戏开发**：借助 Phaser 等游戏开发框架，使用 JavaScript 可以创建各种 2D 游戏，如平台跳跃游戏、射击游戏、益智游戏等，实现游戏角色的移动、碰撞检测、动画效果、游戏逻辑等功能。
  - **游戏逻辑实现**：负责处理游戏中的各种逻辑，如计分系统、关卡设计、游戏规则判断等，为玩家提供完整的游戏体验。

- 其他领域
  - **桌面应用开发**：使用 **Electron 框架**，JavaScript 可以用于开发桌面应用程序，将 Web 技术与本地系统功能相结合，开发出跨平台的桌面应用，如文本编辑器、文件管理器等。
  - **物联网应用**：在物联网领域，JavaScript 可用于与物联网设备进行通信和交互，实现对设备的控制和数据采集，如通过网页或手机应用控制智能家居设备的开关、温度调节等。
  - **数据可视化**：结合 **D3.js**、**ECharts.js** 等库，JavaScript 可以将数据以各种图表（如柱状图、折线图、饼图等）和图形的形式进行可视化展示，帮助用户更直观地理解数据。

1. 概述 1.2 JavaScript能实现什么?

- JavaScript能实现什么?
  - 使用其编程特性实现功能
    - 在变量中储存值。
    - 操作一段文本（在编程中称为“字符串”（string））。
    - 运行代码以响应网页中发生的特定事件。
    - 其他更多
  - 对接浏览器API实现功能
    - **文档对象模型 API 能通过创建、移除和修改 HTML，为页面动态应用新样式等手段来操作 HTML 和 CSS。**比如当某个页面出现了一个弹窗，或者显示了一些新内容（像上文小演示中看到那样），这就是 DOM 在运行。
    - **地理位置 API 获取地理信息。**这就是为什么地图可以找到你的位置，而且标示在地图上。
    - **画布（Canvas） 和 WebGL API 可以创建生动的 2D 和 3D 图像。**人们正运用这些 web 技术制作令人惊叹的作品。参见 Chrome Experiments 以及 webglsamples。
    - **诸如 HTMLMediaElement 和 WebRTC 等影音类 API 让你可以利用多媒体做一些非常有趣的事，**比如在网页中直接播放音乐和影片，或用自己的网络摄像头获取录像，然后在其他人的电脑上展示（试用简易版截图演示以理解这个概念）。
  - 对接第三方API实现功能
    - 调用天气API实现天气预报
    - 调用微信接口实现登录集成
    - 调用图像识别接口实现图片信息读取
    - 等等

2. 调用方法 2.1 在网页中使用JavaScript

- 在网页中调用JavaScript
  - 内联脚本：可以在 HTML 标签的onclick、onload等事件属性中直接编写 JavaScript 代码。

```
<button onclick="alert('Hello, World!')">点击我</button>
```

2. 调用方法 2.1 在网页中使用JavaScript

- 在网页中调用JavaScript
  - 内部脚本：在 HTML 页面的<script>标签内编写 JavaScript 代码。通常将<script>标签放在<head>或<body>标签内。

```html
<!DOCTYPE html>
<html>

<head>
    <title>内部脚本示例</title>
    <script>
        function sayHello() {
            alert('Hello, World!');
        }
    </script>
</head>

<body>
    <button onclick="sayHello()">点击我</button>
</body>

</html>
```

2. 调用方法 2.1 在网页中使用JavaScript

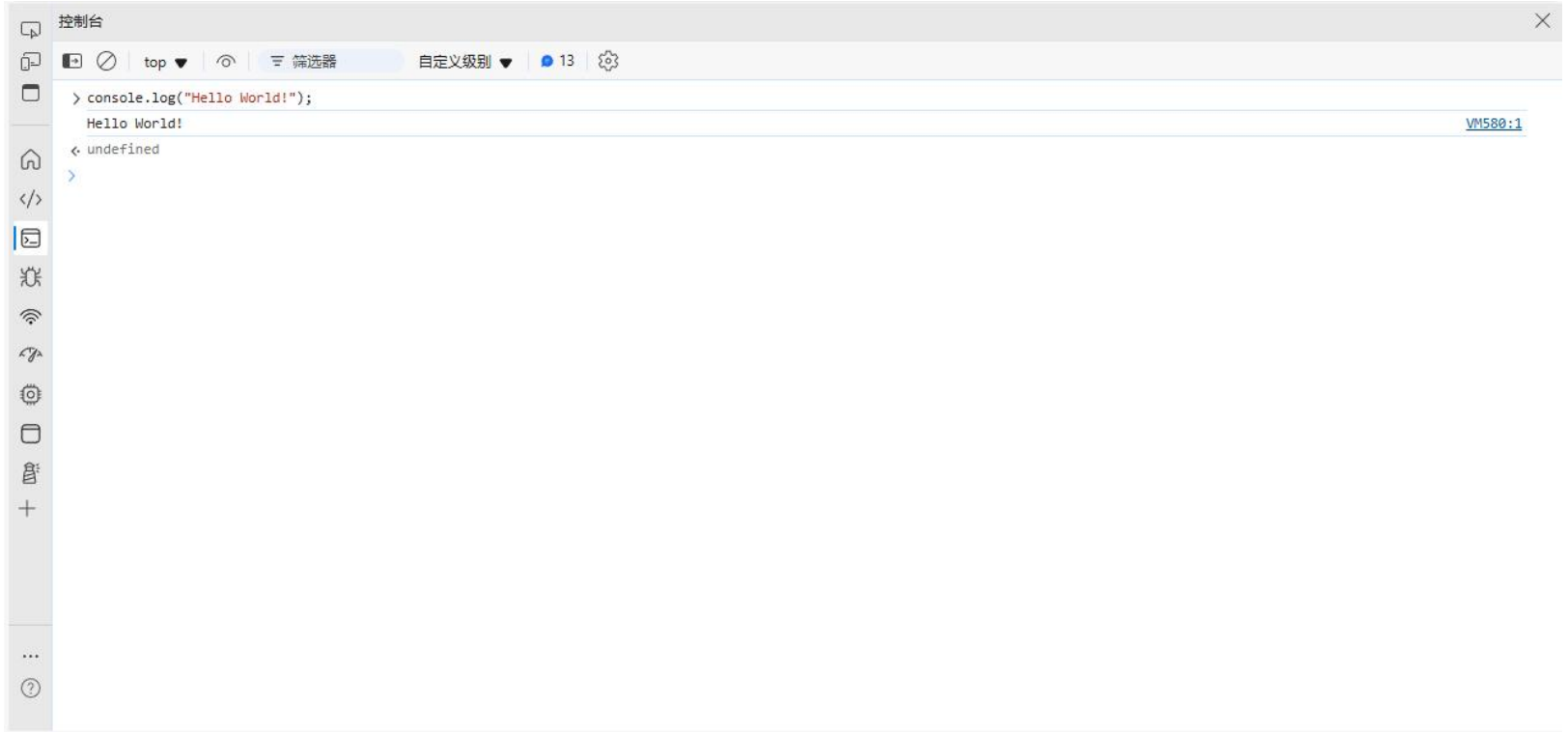
- 在网页中调用JavaScript
  - 外部脚本：将 JavaScript 代码写在一个独立的“.js”文件中，然后在 HTML 页面中通过<script>标签的src属性引入该文件。

```html
<!DOCTYPE html>
<html>

<head>
    <title>外部脚本示例</title>
    <script src="script.js"></script>
</head>

<body>
    <button onclick="sayHello()">点击我</button>
</body>

</html>
```

```javascript
function sayHello() {
    alert('Hello, World!');
}
```

2. 调用方法 2.2 在浏览器控制器中调用JavaScript

- 在浏览器控制台中调用 JavaScript
  - 直接输入：在浏览器中按F12打开开发者工具，切换到控制台（Console）选项卡，就可以直接输入 JavaScript 代码并立即执行。比如输入console.log('Hello, Console!')，会在控制台输出相应内容。
  - 调用页面函数：如果页面中有已经定义好的 JavaScript 函数，也可以在控制台中直接调用。例如页面中有一个sum函数用于计算两个数的和，在控制台中输入sum(2, 3)即可调用该函数并得到结果。



3. 基础语法 3.1 语句

- JavaScript的每个语句以“;”结束，语句块用“{...}”。但是，JavaScript并不强制要求在每个语句的结尾加“;”，浏览器中负责执行JavaScript代码的引擎会自动在每个语句的结尾补上“;”。

```
// JavaScript语句
var x = 1;

//JavaScript语句
'Hello, world';

//JavaScript语句，不建议写到一行
var x = 1; var y = 2;

//JavaScript语句集合
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
}
```

```
//JavaScript语句集合
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
    if (x < y) {
        z = 4;
    }
    if (x > y) {
        z = 5;
    }
}
```

3. 基础语法 3.2 注释

- 使用“//”开头进行单行注释
- 使用“/*...*/”进行多行注释

3. 基础语法 3.3 数据类型

- 数据类型
  - 值类型(基本类型)：字符串（String）、数字(Number)、布尔(Boolean)、空（Null）、未定义（Undefined）、Symbol。
  - 引用数据类型（对象类型）：对象(Object)、数组(Array)、函数(Function)，还有两个特殊的对象：正则（RegExp）和日期（Date）。
    - 函数在 JavaScript 中也被视为对象，因为它们具有属性和方法，可以被赋值给变量、作为参数传递和从函数中返回。
    - 数组本质上也是一种特殊的对象，它的属性是数字索引，用于存储有序的数据集合

- JavaScript 拥有动态类型，变量的数据类型可以使用 typeof 操作符来查看。

```
var x;               // x 为 undefined
var x = 5;           // 现在 x 为数字
var x = "John";      // 现在 x 为字符串
```

```
typeof "John"                 // 返回 string
typeof 3.14                   // 返回 number
typeof false                  // 返回 boolean
typeof [1,2,3,4]              // 返回 object
typeof {name:'John', age:34}  // 返回 object
```

3. 基础语法 3.3 数据类型

- 在 JavaScript 中，字符串（String）是一种基本数据类型，用于表示文本数据。
- 字符串的创建
  - 单引号和双引号：可以使用**单引号（'）或双引号（"）**来创建字符串。

```
let str1 = 'Hello';
let str2 = "World";
```

  - 模板字符串：使用**反引号（`）创建，它支持字符串插值和多行字符串**。

```
let name = 'John';
let message = `Hello, ${name}!
This is a multi - line string.`;
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - length：返回字符串的长度，即字符串中字符的数量。

```
let str = 'JavaScript';
console.log(str.length); // 输出: 10
```

  - charAt()：返回指定索引位置的字符。索引从 0 开始。

```
let str = 'Hello';
console.log(str.charAt(1)); // 输出: e
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - concat()：用于连接两个或多个字符串，并返回一个新的字符串。

```
let str1 = 'Hello';
let str2 = ' World';
let result = str1.concat(str2);
console.log(result); // 输出: Hello World
```

  - indexOf()：返回指定字符串在原字符串中第一次出现的索引，如果未找到则返回 -1。

```
let str = 'Hello World';
console.log(str.indexOf('World')); // 输出: 6
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - lastIndexOf()：返回指定字符串在原字符串中最后一次出现的索引，如果未找到则返回 -1。

```
let str = 'Hello Hello';
console.log(str.lastIndexOf('Hello')); // 输出: 6
```

  - slice()：提取字符串的一部分并返回一个新的字符串。它接受两个参数，分别是起始索引和结束索引（可选）。

```
let str = 'Hello World';
console.log(str.slice(0, 5)); // 输出: Hello
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - substring()：与slice()类似，用于提取字符串的一部分。但它不接受负索引。

```
let str = 'Hello World';
console.log(str.substring(6, 11)); // 输出: World
```

  - substr()：从指定的索引开始提取指定长度的字符串。不过该方法已被标记为不推荐使用，建议使用slice()或substring()。

```
let str = 'Hello World';
console.log(str.substr(6, 5)); // 输出: World
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - toUpperCase()和toLowerCase()：分别用于将字符串转换为大写和小写。

```
let str = 'Hello';
console.log(str.toUpperCase()); // 输出: HELLO
console.log(str.toLowerCase()); // 输出: hello
```

  - trim()：去除字符串两端的空白字符。

```
let str = '   Hello   ';
console.log(str.trim()); // 输出: Hello
```

3. 基础语法 3.3 数据类型

- 字符串的属性和方法
  - replace()：用于替换字符串中的部分内容。它接受两个参数，第一个是要替换的字符串或正则表达式，第二个是替换的内容。

```
let str = 'Hello World';
console.log(str.replace('World', 'JavaScript'));
// 输出: Hello JavaScript
```

  - split()：将字符串分割成数组。它接受一个分隔符作为参数。

```
let str = 'Hello,World';
let arr = str.split(',');
console.log(arr); // 输出: ['Hello', 'World']
```

3. 基础语法 3.3 数据类型

- 字符串的比较
  - 可以使用比较运算符（如==、===、<、>等）来比较字符串。比较是基于字符的 Unicode 值进行的。

```
let str1 = 'apple';
let str2 = 'banana';
console.log(str1 < str2); // 输出: true
```

示例5-3-1：字符串操作

3. 基础语法 3.3 数据类型

- 数据类型：数字
  - 在 JavaScript 中，数字（Number）是一种基本数据类型，用于表示数值。
  - 数字的表示
    - 整数：可以直接使用整数来表示，如 1、-20、0 等。

```
let num1 = 10;
let num2 = -5;
```

    - 浮点数：用于表示小数，如 3.14、-0.5 等。

```
let float1 = 3.14;
let float2 = -0.25;
```

3. 基础语法 3.3 数据类型

- 数据类型：数字
  - 数字的表示
    - 科学计数法：对于非常大或非常小的数字，可以使用科学计数法表示。例如 1e3 表示 1 * 10^3，即 1000；2e-4 表示 2 * 10^(-4)，即 0.0002。

```
let largeNum = 1e3;
let smallNum = 2e-4;
```

  - 特殊数值
    - Infinity 和 -Infinity：分别表示正无穷和负无穷。当进行一些数学运算导致结果超出了 JavaScript 所能表示的最大数值范围时，就会得到 Infinity 或 -Infinity。例如，一个正数除以 0 会得到 Infinity，一个负数除以 0 会得到 -Infinity。

```
let positiveInfinity = 1 / 0;
let negativeInfinity = -1 / 0;
```

3. 基础语法 3.3 数据类型

- 数据类型：数字
  - 特殊数值
    - NaN：即 “Not a Number”，表示不是一个有效的数字。通常在进行无效的数学运算时会得到 NaN，例如 0 / 0、对非数字字符串进行数学运算等。NaN 与任何值（包括它自身）比较都不相等，要判断一个值是否为 NaN，可以使用 isNaN() 函数。

```
let result = 0 / 0;
console.log(isNaN(result)); // 输出：true
```

  - 数字的常用方法
    - toString()：将数字转换为字符串。可以接受一个可选的参数，表示转换的进制，默认是十进制。

```
let num = 10;
let str = num.toString();
let binaryStr = num.toString(2); // 转换为二进制字符串
```

3. 基础语法 3.3 数据类型

- 数据类型：数字
  - 数字的常用方法
    - toFixed()：将数字转换为指定小数位数的字符串。它会进行四舍五入操作。

```
let floatNum = 3.14159;
let fixedStr = floatNum.toFixed(2); // 保留两位小数，输出："3.14"
```

    - parseInt()：将字符串解析为整数。它会忽略字符串开头的空格，从第一个非空格字符开始解析，直到遇到非数字字符为止。如果第一个字符就不是有效的数字字符，则返回 NaN。还可以接受一个可选的第二个参数，表示进制。

```
let str1 = "123abc";
let int1 = parseInt(str1); // 输出：123
let str2 = "0x10";
let int2 = parseInt(str2, 16); // 按十六进制解析，输出：16
```

3. 基础语法 3.3 数据类型

- 数据类型：数字
  - 数字的常用方法
    - parseFloat()：将字符串解析为浮点数，与 parseInt() 类似，但可以解析包含小数点的数字。

```
let str = "3.14abc";
let float = parseFloat(str); // 输出：3.14
```

  - 数字的常用属性
    - Number.MAX_VALUE：表示 JavaScript 中能表示的最大正数，约为 1.7976931348623157e+308。
    - Number.MIN_VALUE：表示 JavaScript 中能表示的最小正数，约为 5e-324。
    - Number.POSITIVE_INFINITY：等同于 Infinity。
    - Number.NEGATIVE_INFINITY：等同于 -Infinity。
    - Number.NaN：等同于 NaN。

3. 基础语法 3.3 数据类型

- 数据类型：布尔
  - 在 JavaScript 里，布尔（Boolean）类型是一种基本数据类型，它只有两个值：true 和 false，主要用于逻辑判断。
  - 布尔值的使用场景
    - 条件判断：在 if、while 等语句中，根据布尔值来决定是否执行特定的代码块。

```
let isAdult = true;
if (isAdult) {
    console.log('你已成年');
} else {
    console.log('你未成年');
}
```

3. 基础语法 3.3 数据类型

- 数据类型：布尔
  - 布尔值的使用场景
    - while 和 do...while 循环会依据布尔表达式的结果来决定是否继续循环。

```
let count = 0;
while (count < 5) {
    console.log(count);
    count++;
}
```

3. 基础语法 3.3 数据类型

- 数据类型：布尔
  - 布尔类型的转换：在 JavaScript 中，许多操作会隐式地将其他数据类型转换为布尔类型。以下是一些常见的数据类型转换为布尔值的规则
    - 以下值转换为布尔值时为 false：
      - false：布尔类型的 false 本身。
      - 0 和 -0：数值类型的零和负零。
      - '' 和 ""：空字符串。
      - null：表示空对象指针。
      - undefined：变量未定义时的值。
      - NaN：表示非数字。

```
console.log(Boolean(false)); // false
console.log(Boolean(0)); // false
console.log(Boolean('')); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN)); // false
```

3. 基础语法 3.3 数据类型

- 数据类型：null
  - null 表示一个空对象指针，通常用于显式地表示一个变量原本应该指向一个对象，但目前没有指向任何对象，是开发者主动设置的空值。
  - 使用场景
    - 初始化对象变量：当你需要声明一个变量来存储对象，但暂时还没有具体的对象可以赋值时，可以将其初始化为 null。

```
let myObject = null;
// 后续根据条件为 myObject 赋值具体对象
if (someCondition) {
    myObject = { name: 'John', age: 30 };
}
```

3. 基础语法 3.3 数据类型

- 数据类型：null
  - null 表示一个空对象指针，通常用于显式地表示一个变量原本应该指向一个对象，但目前没有指向任何对象，是开发者主动设置的空值。
  - 使用场景
    - 释放对象引用：当你不再需要一个对象，并且希望释放该对象占用的内存时，可以将引用该对象的变量赋值为 null。这样，在垃圾回收机制运行时，该对象所占用的内存就可能被回收。

```
let oldObject = { value: 10 };
// 不再需要 oldObject 了
oldObject = null;
```

3. 基础语法 3.3 数据类型

- 数据类型：undefined
  - undefined 表示变量已声明但未赋值，或者函数没有返回值，又或者访问对象不存在的属性时返回的值，它更多地是一种默认的、未定义的状态。

```
// 变量声明但未赋值
let myVariable;
console.log(myVariable); // 输出: undefined

// 函数没有返回值
function myFunction() {}
let result = myFunction();
console.log(result); // 输出: undefined

// 访问对象不存在的属性
let myObj = { name: 'Alice' };
console.log(myObj.age); // 输出: undefined
```

3. 基础语法 3.3 数据类型

- 数据类型：undefined
  - 使用场景
    - 变量声明未赋值：当你声明一个变量但还没有为其赋具体值时，该变量的默认值就是 undefined。
    - 函数参数缺失：如果调用函数时没有提供某个参数，那么该参数在函数内部的值就是 undefined。
  - null 和 undefined 在使用 == 比较时相等，但使用 ===（严格相等）比较时不相等，因为它们的数据类型不同。

3. 基础语法 3.3 数据类型

- 数据类型：Symbol
  - Symbol 主要用于创建唯一的标识符，可通过 Symbol() 函数来创建，该函数可接受一个可选的字符串参数，用于对 Symbol 进行描述，此描述仅用于调试，不会影响 Symbol 的唯一性。

```
// 创建一个没有描述的 Symbol
const sym1 = Symbol();
// 创建一个带有描述的 Symbol
const sym2 = Symbol('mySymbol');
console.log(sym1); // Symbol()
console.log(sym2); // Symbol(mySymbol)
```

3. 基础语法 3.3 数据类型

- 数据类型：Symbol
  - **唯一性**：即使使用相同的描述创建多个 Symbol，它们也是互不相等的，这确保了使用 Symbol 作为对象属性键时不会发生冲突。

```
const symA = Symbol('test');
const symB = Symbol('test');
console.log(symA === symB); // false
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 在 JavaScript 中，对象（Object）是一种非常重要的引用数据类型，用于存储和组织数据。它可以包含各种数据类型的值，并且可以通过属性名来访问这些值。
  - 对象的创建
    - 使用花括号 {} 来创建对象，这是最常见和便捷的方式。可以在花括号内定义对象的属性和方法。

```
const person = {
    name: 'John',
    age: 30,
    sayHello: function() {
        console.log(`Hello, my name is
${this.name}.`);
    }
};
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的创建
    - 使用 new Object() 来创建一个空对象，然后可以动态地添加属性和方法。

```
const car = new Object();
car.make = 'Toyota';
car.model = 'Corolla';
car.getInfo = function() {
    return `${this.make} ${this.model}`;
};
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的属性和方法
    - 属性访问：可以使用点号 . 或方括号 [] 来访问对象的属性。

```
console.log(person.name); // 使用点号访问
console.log(person['age']); // 使用方括号访问
```

    - 属性赋值：同样可以使用点号或方括号来为对象的属性赋值。

```
person.age = 31;
person['name'] = 'Jane';
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的遍历
    - for...in 循环：用于遍历对象的可枚举属性，包括对象自身的属性和继承的属性。

```
for (let key in person) {
    console.log(`${key}: ${person[key]}`);
}
```

    - Object.keys()：返回一个由对象自身可枚举属性组成的数组，然后可以使用数组的遍历方法。

```
const keys = Object.keys(person);
keys.forEach(key => {
    console.log(`${key}: ${person[key]}`);
});
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的遍历
    - Object.entries()：返回一个由对象自身可枚举属性的键值对组成的数组，方便同时获取键和值。

```
const entries = Object.entries(person);
entries.forEach(([key, value]) => {
    console.log(`${key}: ${value}`);
});
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的删除属性：可以使用 delete 操作符来删除对象的属性。

```
delete person.age;
console.log(person.age); // undefined
```

3. 基础语法 3.3 数据类型

- 数据类型：对象
  - 对象的合并：可以使用 Object.assign() 方法将一个或多个源对象的所有可枚举属性复制到目标对象。

```
const target = { a: 1 };
const source = { b: 2 };
const merged = Object.assign(target, source);
console.log(merged); // { a: 1, b: 2 }
```

示例5-3-2：对象操作

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 在 JavaScript 中，数组（Array）是一种特殊的对象，用于存储多个值的有序集合，这些值可以是不同的数据类型。
  - 数组的创建：
    - 使用方括号 [] 来创建数组，这是最常用的方式，可以直接在方括号内列出数组的元素。
    - 可以使用 new Array() 构造函数来创建数组。

```
const fruits = ['apple', 'banana', 'cherry'];
const mixedArray = [1, 'hello', true, null];

// 创建一个包含指定元素的数组
const numbers = new Array(1, 2, 3);
// 创建一个指定长度的空数组
const emptyArray = new Array(5);
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组元素的访问与修改：
    - 访问元素：数组的元素通过索引来访问，索引从 0 开始。可以使用方括号 [] 加上索引值来获取对应位置的元素。
    - 修改元素：同样使用方括号 [] 加上索引值来修改数组中指定位置的元素。

```
const colors = ['red', 'green', 'blue'];
console.log(colors[0]); // 输出: red

colors[1] = 'yellow';
console.log(colors); // 输出: ['red', 'yellow',
'blue']
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 添加和删除元素
    - push()：在数组的末尾添加一个或多个元素，并返回新的数组长度。
    - pop()：移除数组的最后一个元素，并返回该元素。
    - unshift()：在数组的开头添加一个或多个元素，并返回新的数组长度。
    - shift()：移除数组的第一个元素，并返回该元素。

```
const animals = ['cat', 'dog'];
const newLength = animals.push('bird');
console.log(animals); // 输出: ['cat', 'dog', 'bird']
console.log(newLength); // 输出: 3


const lastAnimal = animals.pop();
console.log(animals); // 输出: ['cat', 'dog']
console.log(lastAnimal); // 输出: 'bird'

const newLength2 = animals.unshift('rabbit');
console.log(animals); // 输出: ['rabbit', 'cat', 'dog']
console.log(newLength2); // 输出: 3

const firstAnimal = animals.shift();
console.log(animals); // 输出: ['cat', 'dog']
console.log(firstAnimal); // 输出: 'rabbit'
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组的合并与分割
    - concat()：用于合并两个或多个数组，返回一个新的数组，原数组不会被修改。
    - slice()：从数组中提取指定范围的元素，返回一个新的数组，原数组不会被修改。

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

const numbers = [1, 2, 3, 4, 5];
const sliced = numbers.slice(1, 3);
console.log(sliced); // 输出: [2, 3]
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组的排序与反转
    - sort()：对数组的元素进行排序，默认是按照字符串的 Unicode 码点进行排序。
    - reverse()：反转数组中元素的顺序，原数组会被修改。

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

const numbers = [1, 2, 3, 4, 5];
const sliced = numbers.slice(1, 3);
console.log(sliced); // 输出: [2, 3]
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组的遍历
    - for 循环：最基本的遍历数组的方式，可以精确控制遍历的过程。

```
const fruits2 = ['apple', 'banana', 'cherry'];
for (let i = 0; i < fruits2.length; i++) {
    console.log(fruits2[i]);
}
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组的遍历
    - forEach() 方法：这是数组的一个内置方法，接受一个回调函数作为参数，对数组的每个元素执行一次回调函数。

```
fruits2.forEach((fruit) => {
    console.log(fruit);
});
```

3. 基础语法 3.3 数据类型

- 数据类型：数组
  - 数组的遍历
    - for…of 循环：这是 ES6 引入的一种遍历可迭代对象（包括数组）的方式，语法简洁。

```
for (const fruit of fruits2) {
    console.log(fruit);
}
```

示例5-3-3：数组操作

3. 基础语法 3.4 条件判断

- 在 JavaScript 中，条件判断用于根据不同的条件执行不同的代码块，主要有 if...else 语句、switch 语句以及三元运算符
  - if...else 语句

```
let num = 10;
if (num >= 0) {
    if (num % 2 === 0) {
        console.log('这是一个非负偶数');
    } else {
        console.log('这是一个非负奇数');
    }
} else {
    console.log('这是一个负数');
}
```

3. 基础语法 3.4 条件判断

- 在 JavaScript 中，条件判断用于根据不同的条件执行不同的代码块，主要有 if...else 语句、switch 语句以及三元运算符
  - if...else 语句switch 语句

```
let day = 3;
switch (day) {
    case 1:
        console.log('星期一');
        break;
    case 2:
        console.log('星期二');
        break;
    case 3:
        console.log('星期三');
        break;
    case 4:
        console.log('星期四');
        break;
    case 5:
        console.log('星期五');
        break;
    case 6:
        console.log('星期六');
        break;
    case 7:
        console.log('星期日');
        break;
    default:
        console.log('无效的日期');
}
```

3. 基础语法 3.4 条件判断

- 在 JavaScript 中，条件判断用于根据不同的条件执行不同的代码块，主要有 if...else 语句、switch 语句以及三元运算符
  - 三元运算符

```
let age = 20;
let message = age >= 18? '成年' : '未成年';
console.log(message);
```

3. 基础语法 3.5 循环

- 在 JavaScript 中，循环结构用于重复执行一段代码，直到满足特定条件为止。
  - for 循环：for 循环是最常用的循环结构之一，适用于已知循环次数的情况。它由初始化表达式、条件表达式和更新表达式三部分组成。

```
// 打印 0 到 4 的数字
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

3. 基础语法 3.5 循环

- 在 JavaScript 中，循环结构用于重复执行一段代码，直到满足特定条件为止。
  - while 循环：while 循环在每次循环开始前检查条件表达式的值，只要条件为 true，就会一直执行循环体。

```
// 打印 0 到 4 的数字
let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

3. 基础语法 3.5 循环

- 在 JavaScript 中，循环结构用于重复执行一段代码，直到满足特定条件为止。
  - do...while 循环：do...while 循环与 while 循环类似，但它会先执行一次循环体，然后再检查条件表达式的值。这意味着循环体至少会执行一次。

```
// 打印 0 到 4 的数字
let j = 0;
do {
    console.log(j);
    j++;
} while (j < 5);
```

3. 基础语法 3.5 循环

- 在 JavaScript 中，循环结构用于重复执行一段代码，直到满足特定条件为止。
  - for...in 循环：for...in 循环用于遍历对象的可枚举属性，包括对象自身的属性和继承的属性。

```
const person = {
    name: 'John',
    age: 30,
    job: 'Developer'
};

for (let key in person) {
    console.log(key + ': ' + person[key]);
}
```

3. 基础语法 3.5 循环

- 在 JavaScript 中，循环结构用于重复执行一段代码，直到满足特定条件为止。
  - break 语句：break 语句用于立即终止当前所在的循环，并跳出循环体。
  - continue 语句：continue 语句用于跳过当前循环体中剩余的代码，直接进入下一次循环。

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break;
    }
    console.log(i);
}
```

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        continue;
    }
    console.log(i);
}
```

4. 函数 4.1 函数定义

- 函数声明：使用 function 关键字，后面紧跟函数名、参数列表和函数体。
  - 函数提升：函数声明会被提升到当前作用域的顶部，这意味着你可以在函数声明之前调用该函数。

```
function functionName(parameter1, parameter2, ...) {
    // 函数体，包含要执行的代码
    return result; // 可选的返回语句
}
```

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

let sum = add(3, 5);
console.log(sum); // 输出：8
```

4. 函数 4.1 函数定义

- 函数表达式：函数表达式是将函数赋值给一个变量，这个函数可以是匿名函数，也可以是具名函数。
  - 没有函数提升：函数表达式不会被提升，因此必须在定义之后才能调用。

```
const subtract = function(a, b) {
    return a - b;
};

let difference = subtract(8, 3);
console.log(difference); // 输出: 5
```

```
const power = function calculatePower(base, exponent) {
    let result = 1;
    for (let i = 0; i < exponent; i++) {
        result *= base;
    }
    return result;
};

let value = power(2, 3);
console.log(value); // 输出: 8
```

```
// 下面这行代码会报错，因为函数表达式未定义
// let quotient = divide(10, 2);

const divide = function(a, b) {
    return a / b;
};

let quotient = divide(10, 2);
console.log(quotient); // 输出: 5
```

4. 函数 4.2 函数参数

- 在 JavaScript 中，函数参数是函数定义时用于接收外部传入值的变量，它们在函数内部可以被使用来执行特定的操作。
- 形式参数与实际参数
  - 形式参数：在函数定义时声明的参数，它们是函数内部的占位符，代表着调用函数时将传入的值。
  - 实际参数：在调用函数时传递给函数的具体值。

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

```
let result = add(3, 5);
```

4. 函数 4.2 函数参数

- 默认参数：ES6 引入了默认参数的特性，允许为函数参数设置默认值。当调用函数时没有提供该参数的值，将使用默认值。

```
letfunction greet(name = 'Guest') {
    console.log(`Hello, ${name}!`);
}

greet(); // 输出: Hello, Guest!
greet('John'); // 输出: Hello, John! result = add(3, 5);
```

4. 函数 4.2 函数参数

- 剩余参数：剩余参数使用 ... 语法，允许函数接受不定数量的参数，并将这些参数收集到一个数组中，剩余参数必须是参数列表中的最后一个参数。

```
function sum(...numbers) {
    let total = 0;
    for (let num of numbers) {
        total += num;
    }
    return total;
}

console.log(sum(1, 2, 3)); // 输出: 6
console.log(sum(1, 2, 3, 4, 5)); // 输出: 15
```

4. 函数 4.2 函数参数

- 参数解构：当函数参数是一个对象或数组时，可以使用解构赋值来提取其中的属性或元素。
  - 对象参数解构
  - 数组参数解构

```
function printPersonInfo({ name, age }) {
    console.log(`Name: ${name}, Age: ${age}`);
}

const person = { name: 'Alice', age: 25 };
printPersonInfo(person); // 输出: Name: Alice, Age: 25
```

```
function printFirstTwo([first, second]) {
    console.log(`First: ${first}, Second: ${second}`);
}

const arr = [10, 20, 30];
printFirstTwo(arr); // 输出: First: 10, Second: 20
```

4. 函数 4.2 函数参数

- 参数传递方式：在 JavaScript 中，参数传递方式分为值传递和引用传递。
  - 值传递：对于基本数据类型（如数字、字符串、布尔值等），参数传递是值传递。这意味着函数内部对参数的修改不会影响到外部的原始值。
  - 引用传递：对于引用数据类型（如对象、数组等），参数传递是引用传递。函数内部对参数的修改会影响到外部的原始对象。

```
function changeValue(num) {
    num = num + 10;
    console.log(num); // 输出: 15
}

let original = 5;
changeValue(original);
console.log(original); // 输出: 5
```

```
function changeObject(obj) {
    obj.value = 20;
    console.log(obj.value); // 输出: 20
}

let myObj = { value: 10 };
changeObject(myObj);
console.log(myObj.value); // 输出: 20
```

4. 函数 4.2 函数参数

- 参数的类型检查：JavaScript 是弱类型语言，函数参数没有明确的类型声明。但在实际开发中，有时需要对参数的类型进行检查，以确保函数的正确执行。

```
function divide(a, b) {
    if (typeof a!== 'number' || typeof b!== 'number') {
        throw new Error('Both arguments must be numbers.');
    }
    if (b === 0) {
        throw new Error('Division by zero is not allowed.');
    }
    return a / b;
}

try {
    console.log(divide(10, 2)); // 输出: 5
    console.log(divide('10', 2)); // 抛出错误
} catch (error) {
    console.error(error.message);
}
```

4. 函数 4.3 函数调用

- 普通调用：直接使用函数名加上括号，并在括号内传入必要的参数。

```
// 函数声明
function greet(name) {
    return `Hello, ${name}!`;
}

// 函数调用
const message = greet('John');
console.log(message); // 输出: Hello, John!

// 函数表达式
const add = function(a, b) {
    return a + b;
};

// 函数调用
const sum = add(3, 5);
console.log(sum); // 输出: 8

// 箭头函数
const multiply = (x, y) => x * y;

// 函数调用
const product = multiply(4, 6);
console.log(product); // 输出: 24
```

4. 函数 4.3 函数调用

- 方法调用：当函数作为对象的属性时，称为对象的方法，通过对象名和点号来调用该方法。

```
const person = {
    name: 'Alice',
    sayHello: function() {
        return `Hello, my name is ${this.name}.`;
    }
};

// 方法调用
const greeting = person.sayHello();
console.log(greeting); // 输出: Hello, my name is Alice.
```

4. 函数 4.3 函数调用

- 构造函数调用：使用 new 关键字调用函数，这种函数被称为构造函数，用于创建对象实例。
  - 当使用 new 调用构造函数时，会执行以下步骤：
    - 创建一个新对象。
    - 将新对象的原型指向构造函数的 prototype 属性。
    - 将构造函数的 this 绑定到新对象上。
    - 执行构造函数内部的代码。
    - 如果构造函数返回一个对象，则返回该对象；否则返回新创建的对象。

```
// 构造函数
function Car(make, model) {
    this.make = make;
    this.model = model;
    this.getInfo = function() {
        return `${this.make} ${this.model}`;
    };
}

// 构造函数调用
const myCar = new Car('Toyota', 'Corolla');
const carInfo = myCar.getInfo();
console.log(carInfo); // 输出: Toyota Corolla
```

4. 函数 4.3 函数调用

- 间接调用：可使用 call()、apply() 和 bind() 方法间接调用函数，这些方法允许你指定函数内部 this 的值。
  - call() 方法：call() 方法接受一个 this 值作为第一个参数，后续参数为函数的参数列表。

```
function introduce(city) {
    return `My name is ${this.name} and I'm from ${city}.`;
}

const person1 = { name: 'Bob' };

// 使用 call() 调用函数
const introduction = introduce.call(person1, 'New York');
console.log(introduction);
// 输出: My name is Bob and I'm from New York.
```

4. 函数 4.3 函数调用

- 间接调用：可使用 call()、apply() 和 bind() 方法间接调用函数，这些方法允许你指定函数内部 this 的值。
  - apply() 方法：apply() 方法与 call() 类似，不同之处在于它接受一个数组作为函数的参数。

```
function calculateSum() {
    let sum = 0;
    for (let i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

const numbers = [1, 2, 3, 4];

// 使用 apply() 调用函数
const result = calculateSum.apply(null, numbers);
console.log(result); // 输出: 10
```

4. 函数 4.3 函数调用

- 间接调用：可使用 call()、apply() 和 bind() 方法间接调用函数，这些方法允许你指定函数内部 this 的值。
  - bind() 方法：bind() 方法创建一个新的函数，在调用时将 this 值绑定到指定的对象，并可以预设部分参数。

```
function greetGuest(greeting) {
    return `${greeting}, ${this.name}!`;
}

const guest = { name: 'Eve' };

// 使用 bind() 创建一个新函数
const greetEve = greetGuest.bind(guest, 'Welcome');

// 调用新函数
const welcomeMessage = greetEve();
console.log(welcomeMessage); // 输出: Welcome, Eve!
```

4. 函数 4.3 函数调用

- 间接调用：可使用 call()、apply() 和 bind() 方法间接调用函数，这些方法允许你指定函数内部 this 的值。
  - 立即调用函数表达式（IIFE）：这是一种特殊的函数调用方式，函数定义后立即执行。通常使用匿名函数并将其包裹在括号中，然后在后面加上括号进行调用。

```
// 匿名函数表达式
(function() {
    const message = 'This is an IIFE.';
    console.log(message);
})(); // 输出: This is an IIFE.

// 带参数的 IIFE
(function(name) {
    console.log(`Hello, ${name}!`);
})('Jack'); // 输出: Hello, Jack!
```

示例5-4-1：函数

5. 对象 5.1 标准对象

- 日期对象（Date）
  - Date 对象用于处理日期和时间，能创建日期实例、获取和设置日期时间的各个部分。

```
// 创建当前日期和时间的 Date 对象
const now = new Date();
console.log(now);

// 获取年份
const year = now.getFullYear();
console.log(year);

// 设置月份（0 表示 1 月）
now.setMonth(5);
```

示例5-5-1：Date对象

5. 对象 5.1 标准对象

- 数学对象（Math）
  - Math 对象提供了大量的数学常量和方法，用于进行各种数学计算。
    - 数学常量
      - Math.PI：圆周率。
    - 数学方法
      - Math.random()：生成一个 0 到 1 之间的随机数。
      - Math.abs()：返回一个数的绝对值。

```
console.log(Math.PI); // 输出: 3.141592653589793

const randomNum = Math.random();
console.log(randomNum);

const absNum = Math.abs(-10);
console.log(absNum); // 输出: 10
```

5. 对象 5.1 标准对象

- 正则表达式对象（RegExp）
  - RegExp 对象用于处理正则表达式，可进行字符串的匹配、查找、替换等操作。

```
// 创建正则表达式对象
const regex = /abc/g;
const str = 'abcdefabc';
// 查找匹配项
const matches = str.match(regex);
console.log(matches);
```

示例5-5-2：RegExp对象

5. 对象 5.1 标准对象

- JSON 对象
  - JSON 对象用于处理 JSON（JavaScript Object Notation）数据，提供了 JSON.stringify() 和 JSON.parse() 两个重要方法。

```
const obj = { name: 'John', age: 30 };
// 将对象转换为 JSON 字符串
const jsonStr = JSON.stringify(obj);
console.log(jsonStr);

// 将 JSON 字符串转换为对象
const newObj = JSON.parse(jsonStr);
console.log(newObj);
```

示例5-5-3：JSON对象

5. 对象 5.2 面向对象

- JavaScript 是一种基于原型的面向对象编程语言，它支持面向对象编程的概念，如封装、继承和多态。
- 对象的创建
  - 对象字面量：创建对象最简单的方式，使用花括号 {} 来定义对象的属性和方法。

```
const person = {
    name: 'John',
    age: 30,
    sayHello: function() {
        console.log(`Hello, my name is ${this.name}.`);
    }
};
person.sayHello();
```

5. 对象 5.2 面向对象

- 对象的创建
  - 构造函数：使用 function 关键字定义一个构造函数，通过 new 关键字创建对象实例。构造函数内部使用 this 关键字来设置对象的属性和方法。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello = function() {
        console.log(`Hello, my name is ${this.name} and I'm
${this.age} years old.`);
    };
}
const newPerson = new Person('Alice', 25);
newPerson.sayHello();
```

5. 对象 5.2 面向对象

- 对象的创建
  - 类（ES6 引入）：ES6 引入了 class 关键字，提供了更简洁、更接近传统面向对象语言的方式来定义对象。类实际上是构造函数的语法糖。

```
class Animal {
    constructor(name, species) {
        this.name = name;
        this.species = species;
    }
    makeSound() {
        console.log('Some sound');
    }
}
const dog = new Animal('Buddy', 'Dog');
dog.makeSound();
```

5. 对象 5.2 面向对象

- 封装：封装是将数据（属性）和操作数据的方法（行为）捆绑在一起，并对外部隐藏对象的内部实现细节。在 JavaScript 中，可以通过函数作用域和闭包来实现封装。

```
function createCounter() {
    let count = 0;
    return {
        increment: function() {
            count++;
        },
        getCount: function() {
            return count;
        }
    };
}
const counter = createCounter();
counter.increment();
console.log(counter.getCount());
```

5. 对象 5.2 面向对象

- 继承：继承允许一个对象直接使用另一对象的属性和方法，从而实现代码的复用和层次化组织。在 JavaScript 中有多种实现继承的方式。
  - 原型链继承：每个对象都有一个内部属性 [[Prototype]]（在浏览器中可以通过 __proto__ 访问），它指向该对象的原型对象。当访问一个对象的属性或方法时，JavaScript 首先在对象本身查找，如果找不到，就会沿着原型链向上查找。

```
function Parent() {
    this.parentProperty = 'Parent Value';
}
Parent.prototype.parentMethod = function() {
    console.log('This is a parent method.');
};

function Child() {}
Child.prototype = new Parent();

const child = new Child();
console.log(child.parentProperty);
child.parentMethod();
```

5. 对象 5.2 面向对象

- 继承：继承允许一个对象直接使用另一对象的属性和方法，从而实现代码的复用和层次化组织。在 JavaScript 中有多种实现继承的方式。
  - 类继承（ES6）：使用 extends 关键字实现类之间的继承。

```
class Shape {
    constructor(color) {
        this.color = color;
    }
    getColor() {
        return this.color;
    }
}

class Circle extends Shape {
    constructor(color, radius) {
        super(color);
        this.radius = radius;
    }
    getArea() {
        return Math.PI * this.radius * this.radius;
    }
}
const redCircle = new Circle('red', 5);
console.log(redCircle.getColor());
console.log(redCircle.getArea());
```

5. 对象 5.2 面向对象

- 多态：多态是指不同对象对同一消息做出不同响应的能力。在 JavaScript 中，由于其动态类型的特性，多态可以很自然地实现。

```
class Bird {
    fly() {
        console.log('The bird is flying.');
    }
}

class Penguin extends Bird {
    fly() {
        console.log('Penguins can\'t fly.');
    }
}

function makeFly(animal) {
    animal.fly();
}

const bird = new Bird();
const penguin = new Penguin();
makeFly(bird);
makeFly(penguin);
```

5. 对象 5.2 面向对象

- 静态方法和属性：静态方法和属性属于类本身，而不是类的实例。在 ES6 类中，可以使用 static 关键字来定义静态方法和属性。

```
class Calculator {
    static add(a, b) {
        return a + b;
    }
}
console.log(Calculator.add(2, 3));
```

示例5-5-4：JavaScript面向对象

6. DOM 6.1 什么是DOM?

- 在 JavaScript 中，DOM（Document Object Model，文档对象模型）是一种用于表示 HTML 或 XML 文档的树形结构，它将文档解析为一个由节点和对象组成的结构集合，使得 JavaScript 可以动态地访问、修改和操作网页的内容、结构和样式。
- DOM 节点
  - 节点类型：DOM 树由不同类型的节点组成，常见的节点类型有：
    - 元素节点（Element Node）：表示 HTML 元素，如 、<p>、<a> 等。
    - 文本节点（Text Node）：表示元素中的文本内容。
    - 属性节点（Attribute Node）：表示元素的属性，如 id、class、src 等。

6. DOM 6.1 什么是DOM?

- DOM 节点
  - 获取节点：可以使用多种方法来获取 DOM 中的节点。
    - 通过 ID 获取：使用 document.getElementById() 方法，根据元素的 id 属性获取单个元素节点。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <div id="myDiv">This is a div.</div>
    <script>
        const div = document.getElementById('myDiv');
        console.log(div.textContent);
    </script>
</body>

</html>
```

6. DOM 6.1 什么是DOM?

- DOM 节点
  - 获取节点：可以使用多种方法来获取 DOM 中的节点。
    - 通过标签名获取：使用 document.getElementsByTagName() 方法，根据元素的标签名获取一组元素节点，返回一个 HTMLCollection 对象。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <script>
        const paragraphs = document.getElementsByTagName('p');
        for (let i = 0; i < paragraphs.length; i++) {
            console.log(paragraphs[i].textContent);
        }
    </script>
</body>

</html>
```

6. DOM 6.1 什么是DOM?

- DOM 节点
  - 获取节点：可以使用多种方法来获取 DOM 中的节点。
    - 通过类名获取：使用 document.getElementsByClassName() 方法，根据元素的 class 属性获取一组元素节点，返回一个 HTMLCollection 对象。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <div class="myClass">Div 1</div>
    <div class="myClass">Div 2</div>
    <script>
        const divs = document.getElementsByClassName('myClass');
        for (let i = 0; i < divs.length; i++) {
            console.log(divs[i].textContent);
        }
    </script>
</body>

</html>
```

6. DOM 6.1 什么是DOM?

- DOM 节点
  - 获取节点：可以使用多种方法来获取 DOM 中的节点。
    - 通过选择器获取：使用 document.querySelector() 和 document.querySelectorAll() 方法，支持使用 CSS 选择器来获取节点。querySelector() 返回第一个匹配的元素节点，querySelectorAll() 返回所有匹配的元素节点，返回一个 NodeList 对象。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
    </ul>
    <script>
        const firstLi = document.querySelector('ul li');
        console.log(firstLi.textContent);

        const allLis = document.querySelectorAll('ul li');
        allLis.forEach(li => {
            console.log(li.textContent);
        });
    </script>
</body>

</html>
```

6. DOM 6.2 操作DOM

- 节点操作
  - 创建节点：使用 document.createElement() 方法创建新的元素节点，使用 document.createTextNode() 方法创建新的文本节点。

```
const newDiv = document.createElement('div');
const textNode = document.createTextNode('This is a new div.');
newDiv.appendChild(textNode);
```

6. DOM 6.2 操作DOM

- 节点操作
  - 添加节点：使用 appendChild() 方法将一个节点添加到另一个节点的子节点列表末尾，使用 insertBefore() 方法将一个节点插入到另一个节点之前。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <div id="parentDiv">
        <p>Existing paragraph</p>
    </div>
    <script>
        const parentDiv = document.getElementById('parentDiv');
        const newParagraph = document.createElement('p');
        newParagraph.textContent = 'New paragraph';
        parentDiv.appendChild(newParagraph);

        const firstChild = parentDiv.firstChild;
        const anotherParagraph = document.createElement('p');
        anotherParagraph.textContent = 'Another new paragraph';
        parentDiv.insertBefore(anotherParagraph, firstChild);
    </script>
</body>

</html>
```

6. DOM 6.2 操作DOM

- 节点操作
  - 删除节点：使用 removeChild() 方法删除一个节点。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <div id="parentDiv">
        <p id="childParagraph">This is a paragraph to be removed.</p>
    </div>
    <script>
        const parentDiv = document.getElementById('parentDiv');
        const childParagraph = document.getElementById('childParagraph');
        parentDiv.removeChild(childParagraph);
    </script>
</body>

</html>
```

6. DOM 6.2 操作DOM

- 节点操作
  - 修改节点：可以直接修改节点的属性和内容。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <img id="myImage" src="old-image.jpg" alt="Old Image">
    <script>
        const myImage = document.getElementById('myImage');
        myImage.src = 'new-image.jpg';
        myImage.alt = 'New Image';
    </script>
</body>

</html>
```

6. DOM 6.4 事件处理

- 事件是文档或浏览器窗口中发生的特定交互瞬间，如点击按钮、鼠标移动等。可以使用 JavaScript 为 DOM 元素添加事件处理程序。
  - 内联事件处理程序：在 HTML 标签中直接使用事件属性来绑定事件处理程序。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <button onclick="alert('Button clicked!')">Click me</button>
</body>

</html>
```

6. DOM 6.4 事件处理

- 通过 JavaScript 代码为元素的事件属性赋值来绑定事件处理程序。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <button id="myButton">Click me</button>
    <script>
        const myButton = document.getElementById('myButton');
        myButton.onclick = function () {
            alert('Button clicked!');
        };
    </script>
</body>

</html>
```

6. DOM 6.4 事件处理

- 使用 addEventListener() 方法来绑定事件处理程序，这种方式可以为同一个元素的同一个事件绑定多个处理程序。

```html
<!DOCTYPE html>
<html lang="en">

<body>
    <button id="myButton">Click me</button>
    <script>
        const myButton = document.getElementById('myButton');
        myButton.addEventListener('click', function () {
            alert('Button clicked!');
        });
    </script>
</body>

</html>
```

示例5-6-1：DOM操作

7. 浏览器对象

- 在 JavaScript 中，浏览器对象模型（BOM，Browser Object Model）提供了与浏览器窗口进行交互的接口。
  - window 对象：window 对象是浏览器对象模型的顶层对象，代表浏览器窗口。几乎所有全局变量和函数都是 window 对象的属性和方法。
  - document 对象：document 对象是 window 对象的一个属性，代表当前加载的 HTML 文档。可以使用它来访问和操作文档中的元素。
  - location 对象：location 对象提供了有关当前文档的 URL 信息，并且可以用于导航到新的 URL。
  - history 对象：history 对象允许你访问浏览器的会话历史记录，实现前进、后退等操作。
  - navigator 对象：navigator 对象包含有关浏览器的信息，如浏览器名称、版本、操作系统等。

示例5-7-1：浏览器对象

扩展：jQuery

- jQuery 是一个快速、简洁且功能强大的 JavaScript 库，由 John Resig 在 2006 年创建。它简化了 HTML 文档遍历、事件处理、动画效果和 Ajax 交互等操作，使得开发者能够更高效地编写跨浏览器兼容的 JavaScript 代码。
  - 简洁的选择器：jQuery 提供了强大且简洁的选择器语法，允许开发者使用类似于 CSS 选择器的方式来选取 HTML 元素。这大大简化了在文档中查找和操作元素的过程。
  - 动画效果：jQuery 提供了一系列内置的动画方法，能够方便地创建各种动画效果，如显示、隐藏、淡入淡出、滑动等，还支持自定义动画。
  - Ajax 交互：jQuery 封装了复杂的 Ajax 操作，使得开发者可以轻松地与服务器进行异步数据交互，支持多种数据格式（如 JSON、XML 等）。
  - 跨浏览器兼容性：jQuery 处理了不同浏览器之间的兼容性问题，确保代码在各种主流浏览器（如 Chrome、Firefox、Safari、IE 等）中都能正常运行。开发者无需担心不同浏览器对 JavaScript 和 CSS 的实现差异。
  - 插件生态系统：jQuery 拥有丰富的插件生态系统，开发者可以根据自己的需求选择合适的插件来扩展功能。例如，用于表单验证的 jQuery Validation Plugin、用于创建滑块的 jQuery UI Slider 等。这些插件可以帮助开发者快速实现复杂的功能，提高开发效率。

扩展：jQuery

- jQuery 是一个快速、简洁且功能强大的 JavaScript 库，由 John Resig 在 2006 年创建。它简化了 HTML 文档遍历、事件处理、动画效果和 Ajax 交互等操作，使得开发者能够更高效地编写跨浏览器兼容的 JavaScript 代码。
  - 局限性：虽然 jQuery 有很多优点，但随着现代 JavaScript 的发展，尤其是 ES6+ 标准的推出以及原生 JavaScript API 的不断完善，jQuery 的一些优势逐渐被削弱。例如，现代浏览器对原生 JavaScript 的支持越来越好，使得开发者可以使用原生方法完成一些原本需要借助 jQuery 的操作。此外，引入 jQuery 会增加页面的加载时间和代码体积，对于一些轻量级的项目来说可能不是最佳选择。
  - 总的来说，jQuery 在过去很长一段时间内是前端开发中不可或缺的工具，即使在现代前端开发中，它仍然有一定的应用场景，尤其是在一些需要快速开发和处理兼容性问题的项目中。

示例5-8-1：浏览器对象

重点回顾

- 概述
  - 定义：一种高级解释型编程语言，1995 年由 Netscape 公司 Brendan Eich 开发，最初用于为网页添加动态效果和交互性，如今应用广泛。
  - 特点：具有动态类型、事件驱动、跨平台、面向对象和解释执行的特点。
  - 应用领域：涵盖网页开发、服务器端编程、移动应用开发、游戏开发、桌面应用开发、物联网应用和数据可视化等多个领域。
- 调用方法
  - 在网页中使用：有内联脚本、内部脚本和外部脚本三种方式。内联脚本在 HTML 标签事件属性中编写代码；内部脚本在 HTML 页面的<script>标签内编写；外部脚本将代码写在独立“.js” 文件中，通过<script>标签的 src 属性引入。
  - 在浏览器控制台中调用：可直接输入 JavaScript 代码执行，也能调用页面中已定义的函数。
- 基础语法
  - 语句和注释：语句以 “;” 结束（非强制），语句块用 “{...}”；注释分为单行注释 “//” 和多行注释 “/.../”。
  - 数据类型：包括值类型（字符串、数字、布尔、空、未定义、Symbol）和引用数据类型（对象、数组、函数、正则、日期）。不同数据类型有各自的创建方式、属性、方法及比较规则。
  - 条件判断和循环：条件判断有 if...else 语句、switch 语句和三元运算符；循环结构有 for 循环、while 循环、do...while 循环、for...in 循环，还有 break 和 continue 语句用于控制循环。

重点回顾

- 函数
  - 函数定义：有函数声明（具函数提升特性）和函数表达式（无函数提升）两种方式。
  - 函数参数：包括形式参数和实际参数，ES6 引入默认参数、剩余参数，还支持参数解构；参数传递分为值传递（基本数据类型）和引用传递（引用数据类型），开发中可进行参数类型检查。
  - 函数调用：有普通调用、方法调用、构造函数调用、间接调用（call ()、apply ()、bind () 方法）和立即调用函数表达式（IIFE）等方式。
- 对象
  - 标准对象：如日期对象（Date）、数学对象（Math）、正则表达式对象（RegExp）、JSON 对象，分别用于处理日期时间、数学计算、字符串匹配和 JSON 数据。
  - 面向对象：JavaScript 基于原型实现面向对象编程，支持封装、继承和多态。对象创建方式有对象字面量、构造函数和 ES6 的 class；还可定义静态方法和属性。
- DOM
  - 概念：DOM 是表示 HTML 或 XML 文档的树形结构，由元素节点、文本节点、属性节点等组成，可通过多种方法获取节点。
  - 操作 DOM：包括创建、添加、删除和修改节点；还可通过内联事件处理程序、为元素事件属性赋值、addEventListener () 方法为 DOM 元素添加事件处理程序。

重点回顾

- 浏览器对象
  - 浏览器对象模型（BOM）提供与浏览器窗口交互的接口，包括 window 对象（顶层对象）、document 对象（代表 HTML 文档）、location 对象（提供 URL 信息）、history 对象（访问会话历史记录）、navigator 对象（包含浏览器信息）。
- 扩展：jQuery
  - 简介：是一个 JavaScript 库，简化了 HTML 文档遍历、事件处理等操作，具有简洁的选择器、丰富的动画效果、方便的 Ajax 交互、良好的跨浏览器兼容性和丰富的插件生态系统。
  - 局限性：随着现代 JavaScript 发展，其优势部分被削弱，引入它会增加页面加载时间和代码体积。

信创智能医疗系统研发课程体系

河南中医药大学信息技术学院（智能医疗行业学院）

河南中医药大学信息技术学院（智能医疗行业学院）智能医疗教研室

河南中医药大学医疗健康信息工程技术研究所