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
2025/12/3

本章概要

- 概述
- 语法
- DOM
- BOM
- 常用内置对象
- 事件系统

1. 概述

- 随着互联网的不断发展，网站已不再局限于展示静态信息。增强网页交互性、提升用户体验，已然成为用户最基本的需求。因此，深入了解并学习JavaScript，不仅是Web前端开发者提升网站用户体验、增强用户黏性的必要条件，更是其必须掌握的重要技术能力之一。
- JavaScript 最初的创建目的是**“使网页更生动”**。用这种编程语言编写的程序被称作脚本，它们能够直接嵌入网页的 HTML 代码中，在页面加载时自动执行。脚本以纯文本形式提供和运行，无需进行特殊准备或编译。

1. 概述

- 如今，JavaScript 不仅能够在浏览器中运行，还可以在服务端执行，甚至能在任何搭载了 JavaScript 引擎的设备上运行。浏览器中嵌入的 JavaScript 引擎，有时也被称作“JavaScript 虚拟机”。不同的引擎有各自独特的“代号”，例如：
  - **V8：** Google 开发的开源 JavaScript 引擎，首次发布于 2008 年，广泛用于 Google Chrome 和 Node.js。它的设计目标是提供高性能和高效的内存管理，Chrome、Opera 和 Edge 都是用此引擎。
  - **SpiderMonkey：** Mozilla 开发的 JavaScript 引擎，首次发布于 1995 年，是第一个 JavaScript 引擎，主要用于 Firefox 浏览器。
  - **JavaScriptCore（也称Nitro）：** Apple 开发的 JavaScript 引擎，广泛用于 Safari 和 WebKit 项目中，首次发布于 2002 年。
  - **Chakra：** Microsoft 开发的 JavaScript 引擎，首次发布于 2009 年，主要用于旧版 Edge 浏览器。ChakraCore 是其开源版本，可以在不同平台上使用。
  - **Rhino：** Mozilla 开发的基于 Java 的 JavaScript 引擎，首次发布于 1997 年。作为一个纯 Java 实现的引擎，Rhino 允许在 Java 应用中嵌入和执行 JavaScript 代码。
  - **JerryScript：** Samsung 开发的超轻量级 JavaScript 引擎，专为资源受限的物联网设备设计，首次发布于 2015 年。

1. 概述

- 现代 JavaScript 是一门“安全的”编程语言。由于其最初专为浏览器开发，无需对内存或 CPU 进行底层访问，因此该语言并未提供相关功能。JavaScript 的功能在很大程度上取决于其运行环境。
- 例如，Node.js 支持允许 JavaScript 读取/写入任意文件，执行网络请求等的函数。
- 浏览器中的 JavaScript 可以做与网页操作、用户交互和 Web 服务器相关的所有事情，具体如下。
  - 在网页中添加新的 HTML 代码，修改网页既有内容及样式。
  - 响应用户操作，包括鼠标点击、指针移动以及按键按下等行为。
  - 向远程服务器发起网络请求，进行文件的下载与上传（即所谓的 AJAX 和 COMET 技术）。
  - 获取或设置 cookie，向访问者提问或发送消息。
  - 存储客户端数据（即“本地存储”）。

1. 概述

- 为保障用户的信息安全，浏览器对 JavaScript 的功能进行了限制。此举旨在防止恶意网页窃取用户的个人信息或破坏用户数据，具体情况如下。
  - 网页中的 JavaScript 无法对硬盘上的任意文件进行读取、写入、复制和执行操作，因为它不具备直接访问操作系统的功能。
  - 现代浏览器虽允许 JavaScript 进行一些与文件相关的操作，但这些操作受到严格限制。仅当用户做出特定行为时，JavaScript 才能够对文件进行操作，例如用户将文件“拖放”至浏览器中，或者通过 <input> 标签选择文件。
  - 尽管存在多种与相机、麦克风及其他设备进行交互的方式，但这些操作均需获得用户的明确许可。因此，启用 JavaScript 的网页不会在用户不知情的情况下启动网络摄像头获取信息并发送出去。
  - 不同的标签页或窗口通常相互独立。不过，在某些情况下也会存在关联，例如一个标签页通过 JavaScript 打开另一个标签页。但即便如此，若两个标签页所打开的并非同一网站（域名、协议或端口任意一项不同），它们之间将无法进行通信。这便是所谓的“同源策略”。为解决“同源策略”带来的问题，两个标签页均需包含特定的 JavaScript 代码以处理该问题，并且都要允许数据交换。本教程将对这部分相关知识进行讲解。此限制同样是出于保障用户信息安全的考虑。例如，用户打开的 http://jd.com 网页绝不能访问 http://www.baidu.com（另一标签页打开的网页），更不能从中窃取信息。
  - JavaScript 能够便捷地通过互联网与当前页面所在的服务器进行通信，但从其他网站或域的服务器接收数据的能力有所受限。尽管可以实现，但需要远程服务器在 HTTP 头中给出明确许可，这同样是为了确保用户的信息安全。

1. 概述

- JavaScript 于 1995 年诞生，几乎与互联网同步出现；而 Node.js 则诞生于 2009 年，较 JavaScript 晚了约 15 年。
- 在 Node.js 问世前，JavaScript 仅能在浏览器环境中运行，作为网页脚本使用，主要用于为网页增添特效或与服务器进行通信。Node.js 出现后，JavaScript 得以脱离浏览器的束缚，如同其他编程语言一样，可以直接在计算机上自由运用，不再受限于浏览器环境。
- Node.js 既非一门全新的编程语言，也不是一个 JavaScript 框架，而是一套用于支持 JavaScript 代码执行的运行环境。从编程术语角度而言，Node.js 是一个 JavaScript 运行时（Runtime）。
- 如今，JavaScript 除了应用于 Web 前端编程（即网页编程）外，还具备诸多应用场景，例如：
  - 开发网站后台，这一领域原本是 PHP、Java、Python、Ruby 等编程语言的专长；
  - 开发 GUI 程序，即常见的带界面的电脑软件，如 QQ、360、迅雷等；
  - 开发手机 APP，涵盖 Android APP 和 iOS APP；
  - 开发 CLI 工具，也就是无界面的命令行程序。

1. 概述

- 所谓运行时，指的是程序在运行过程中所依赖的一系列组件或工具。将这些组件与工具打包提供给程序员，程序员便能顺利运行自己编写的代码。
- 对于 JavaScript 而言，其在运行期间依赖以下组件：
  - (1) 解释器
    - JavaScript 作为一种脚本语言，采用边解释边运行的方式，仅对当前使用的源代码进行编译，这一过程由解释器负责完成。若缺少解释器，JavaScript 代码仅为普通的纯文本文件，无法被计算机识别。
  - (2) 标准库
    - 在 JavaScript 代码里，会调用一些内置函数，这些函数并非自行编写，而是标准库所提供的。
  - (3) 本地模块
    - 本地模块是指预先编译好的模块，它们以二进制文件形式存在，其内部结构与可执行文件并无显著差异，只是无法独立运行。实际上，这些本地模块就是动态链接库（在 Windows 系统下为 .dll 文件）。若有使用 C 语言、C++ 等编译型语言的经验，将能更深入地理解本地模块的概念。JavaScript 的很多功能都需要本地模块的支持，比如：
      - Cookie 作为存储于用户计算机中的小型文件，用于存储少量用户数据。使用 Cookie 需有文件操作模块的支持。
      - Ajax 作为一种网络操作，可借助互联网从服务器请求数据，此操作需有网络库的支持。
      - 通过逐步跟踪代码执行流程以发现逻辑错误的过程称为调试，该过程需要调试器（Debugger）的支持。
      - JavaScript 若要操作 HTML，需要 HTML 解析模块预先构建起 DOM 树。
    - 本地模块通常封装通用功能，对性能要求较高，因此常采用编译型语言实现，如 C 语言、C++、汇编语言等。

1. 概述

- Node.js 运行时主要由 V8 引擎、标准库以及本地模块构成，其中，本地模块的数量从底层决定了 Node.js 功能的强弱。
  - V8 引擎
    - V8 引擎就是 JavaScript 解释器，负责解析和执行 JavaScript 代码。
    - V8 引擎借鉴了 Java 虚拟机和 C++ 编译器的众多技术，它将 JavaScript 代码直接编译成原生机器码，并且使用了缓存机制来提高性能，这使得 JavaScript 的运行速度可以媲美二进制程序。
  - 本地模块
    - Node.js 集成了大量高性能的本地模块，这些库采用 C/C++ 语言实现，如libuv、nmp、http_parser、zlib等等。
    - Node.js 可直接在计算机上运行 JavaScript 代码，并赋予 JavaScript 强大功能。正因如此，其本地模块与浏览器中的运行时存在显著差异，甚至可以说几乎毫无关联。Node.js 几乎完全摒弃了浏览器环境，自行构建了一套全新的 JavaScript 运行时。
  - 标准库
    - 本地模块采用 C/C++ 语言编写，而 Node.js 主要面向 JavaScript 开发者。因此，有必要对本地模块的 C/C++ 接口进行封装，为开发者提供一套简洁高效的 JavaScript 接口，同时确保该接口在不同平台（操作系统）上具备一致性。此套 JavaScript 接口即 Node.js 标准库。标准库的优劣，包括其优雅性与强大功能，直接决定了 Node.js 的易用性，进而对 Node.js 的市场表现产生显著影响。

1. 概述

- 不同个体对功能的需求存在差异，JavaScript 的语法难以满足所有人的要求。因此，近期涌现出众多新语言，这些语言在浏览器中执行前，均会被编译为 JavaScript。现代化工具使得编译过程迅速且无缝，允许开发者使用其他语言编写代码，随后自动将其转换为 JavaScript，具体语言示例如下：
  - CoffeeScript 作为 JavaScript 的语法糖，引入了更为简洁的语法，有助于编写更清晰、精炼的代码，通常受到 Ruby 开发者的青睐。
  - TypeScript 着重添加“严格的数据类型”，以简化开发流程，更好地支持复杂系统的构建，由微软开发。
  - Flow 同样引入了数据类型，不过采用了不同的实现方式，由 Facebook 开发。
  - Dart 是一门独立的编程语言，拥有自身的引擎，可在非浏览器环境（如手机应用）中运行，也能被编译成 JavaScript，由 Google 开发。
  - Brython 是一个 Python 到 JavaScript 的转译器，使得能够在不使用 JavaScript 的情况下，用纯 Python 编写应用程序。
  - Kotlin 是一种现代、简洁且安全的编程语言，其编写的应用程序可在浏览器和 Node 环境中运行。

2. 语法

- 在 HTML 文档中嵌入 JavaScript 代码
  - 若要在 HTML 页面中嵌入 JavaScript 脚本，需使用 <script> 标签。用户可直接在 <script> 标签内编写 JavaScript 代码，具体步骤如下。
  - 新建 HTML 文档，保存为 index.html。
  - 在 <head> 标签内插入一个 <script> 标签。
  - 为 <script> 标签设置type="text/javascript"属性，在现代浏览器中，默认情况下，<script> 标签的脚本类型为 JavaScript，故而可以省略 type 属性。然而，若要兼容早期版本的浏览器，则需设置该 type 属性。
  - 在 <script> 标签内输入 JavaScript 代码document.write("<h1>Hello World</h1>");。
  - 保存网页文档，在浏览器中预览。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>第一个JavaScript程序</title>
    <script type="text/javascript">
        document.write("<h1>第一个JavaScript程序示例</h1>");
    </script>
</head>
<body></body>
</html>
```


2. 语法

- 在脚本文件中编写 JavaScript 代码
  - JavaScript 脚本文件属于文本文件，其扩展名为.js，可通过任意文本编辑器进行编辑。以下为新建 JavaScript 文件的具体步骤。
  - 新建文本文件，保存为 demo.js。注意，扩展名为.js，表示该文本文件是 JavaScript 类型的文件。
  - 打开 demo.js 文件，在其中编写如下 JavaScript 代码。

```
alert("Hello World");
```

  - 保存 JavaScript 文件。JavaScript 文件不能够独立运行，需要导入到网页中，通过浏览器来执行。使用 <script> 标签可以导入 JavaScript 文件。
  - 新建 HTML 文档，保存为 index.html。
  - 在 <head> 标签内插入一个 <script> 标签。定义 src 属性，设置属性值为指向外部 JavaScript 文件的 URL 字符串。代码如下：

```
<script type="text/javascript" src="demo.js"></script>
```

  - 保存网页文档，在浏览器中预览。定义 src 属性的 <script> 标签不应再包含 JavaScript 代码。如果嵌入了代码，则只会下载并执行外部 JavaScript 文件，嵌入代码将被忽略。

2. 语法

- 标识符
  - 所谓标识符（Identifier），即名字。在 JavaScript 里，标识符涵盖变量名、函数名、参数名、属性名、类名等。定义合法的标识符时，需遵循以下强制规则：
    - 标识符的首个字符必须为字母、下划线（_）或者美元符号（$）。
    - 除首字符外，其他位置可使用 Unicode 字符。不过，通常建议仅采用 ASCII 编码的字母，不推荐使用双字节字符。
    - 标识符不能与 JavaScript 的关键字、保留字重复。
    - 标识符可以运用 Unicode 转义序列。例如，字符 a 可用“\u0061”来表示。

2. 语法

- 关键字
  - 关键字（Keyword）指的是 JavaScript 语言内部使用的一组名称（亦可称为命令）。这些名称具备特定用途，用户无法自定义与之同名的标识符，具体说明如下表所示。

break	delete	if	this	while
case	do	in	throw	with
catch	else	instanceof	try	
continue	finally	new	typeof	
debugger	for	return	var	
default	function	switch	void	

2. 语法

- 区分大小写
  - JavaScript 对大小写敏感，因此“Hello”与“hello”属于两个不同的标识符。为避免输入混淆以及语法错误，建议在编写代码时采用小写字符。不过，在以下特殊情形中，可使用大写形式。
    - 构造函数的首字母建议大写。

```
1. d = new Date();  //获取当前日期和时间
2. document.write(d.toString());  // 显示日期
```

    - 如果标识符由多个单词组成，可以考虑使用骆驼命名法（除首个单词外，后面单词的首字母大写）。例如：

```
1. typeOf();
2. printEmployeePaychecks();
```


2. 语法

- 字面量
  - 字面量（Literal），亦称直接量，指的是具体的值，也就是能够直接参与运算或进行显示的值，例如字符串、数值、布尔值、正则表达式、对象字面量、数组字面量以及函数字面量等。

```
1. //空字符串直接量
2. 1  //数值直接量
3. true  //布尔值直接量
4. /a/g  //正则表达式直接量
5. null  //特殊值直接量
6. {}  //空对象直接量
7. []  //空数组直接量
8. function(){}  //空函数直接量，也就是函数表达式
```

2. 语法

- 注释
  - 注释主要是为开发人员提供的，程序执行时会自动忽略注释内容。因此，通常借助注释为代码添加解释说明或描述，以此提升代码的可读性。JavaScript 中的注释定义方式与 C/C++、Java、PHP 等语言一致，支持单行注释和多行注释两种形式。
  - 单行注释
    - 单行注释以双斜杠 “//” 开头，“//” 之后的所有内容均会被视为注释，而对 “//” 之前的内容不产生任何影响，示例代码如下。

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript</title>
</head>
<body>
    <div id="demo"></div>
    <script>
        // 在 id 属性为 demo 的标签中添加指定内容
        document.getElementById("demo").innerHTML = "http://c.biancheng.net/js/";
    </script>
</body>
</html>
```


2. 语法

- 注释
  - 多行注释
    - 多行注释以 /* 起始，以 */ 结束，位于 /* 与 */ 之间的所有内容均会被视为注释内容，示例代码如下。

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript</title>
</head>
<body>
    <div id="demo"></div>
    <script>
        /*
        在 id 属性为 demo 的标签中
        添加指定内容
        */
        document.getElementById("demo").innerHTML = "https://www.baidu.com/";
    </script>
</body>
</html>
```

2. 语法

- 变量作为所有编程语言的基础要素之一，可用于存储各类数据，如字符串、数字、布尔值、数组等。同时，在有需求时，能够对变量中的内容进行设置、更新或读取操作。
- 在定义变量时，变量名应尽量具有明确的语义，以便自己和他人能够轻松理解。例如，可使用“name”来定义存储姓名的变量，使用“dataArr”来定义数组类型的变量。当变量名包含多个英文单词时，推荐采用驼峰命名法。

2. 语法

- 在 JavaScript 里，定义变量需使用 “var” 关键字。定义变量时，可一次性定义一个或多个变量。若要定义多个变量，需在变量名之间用逗号 “,” 分隔，示例代码如下。

```
var a;
var b, c, d;    // 同时声明多个变量
```

- 当变量被定义但未进行赋值操作时，这些变量将被赋予初始值 “undefined”（即未定义）。
- 变量定义后，可以使用等于号=来为变量赋值，等号左边的为变量的名称，等号右边为要赋予变量的值，示例代码如下。

```
var num;    // 定义一个变量 num
num = 1;    // 将变量 num 赋值为 1
```

- 此外，也可以在定义变量的同时为变量赋值，示例代码如下。

```
var num = 1;                // 定义一个变量 num 并将其赋值为 1
var a = 2, b = 3, c = 4;    // 同时定义 a、b、c 三个变量并分别赋值为 2、3、4
// var a = 2,               // 为了让代码看起来更工整，上一行代码也可以写成这样
//     b = 3,
//     c = 4;
```

2. 语法

- 在 JavaScript 的预编译阶段，会对声明的变量进行预处理。以下示例代码中，变量声明置于末尾，而赋值操作则放在前面。由于 JavaScript 会在预编译阶段对变量声明语句进行预解析，因此在第 1 行代码读取变量值时，不会抛出异常，而是返回未初始化的值 undefined。第 3 行代码在赋值操作之后进行读取，所以显示为数字 1。

```
1. document.write(str); //显示undefined
2. str = http://c.biancheng.net/js/;
3. document.write(str); //显示 http://c.biancheng.net/js/
4. var str;
```

- JavaScript 引擎的解析机制为：首先对代码进行解析，提取所有已声明的变量，随后逐行执行代码。在此过程中，所有声明的变量会被提升至代码头部，这一现象被称为变量提升（Hoisting）。

2. 语法

- 在 2015 年之前，JavaScript 仅能借助 var 关键字来声明变量。自 ECMAScript6（ES6）发布后，新增了 let 和 const 这两个关键字用于变量声明，具体情况如下：
  - 使用 let 关键字声明的变量仅在其所在的代码块内有效，类似于局部变量，且在该代码块中，禁止重复声明同名变量。
  - const 关键字的功能与 let 类似，但使用 const 关键字声明的变量具有不可修改的特性，即一旦定义，其值便不可更改，因此这类变量实际上为常量。

```
1. let name = "小明";      // 声明一个变量 name 并赋值为 “小明”
2. let age  = 11;          // 声明一个变量 age
3. let age  = 13;          // 报错：变量 age 不能重复定义
4. const PI = 3.1415       // 声明一个常量 PI，并赋值为 3.1415
5. console.log(PI)         // 在控制台打印 PI
```

2. 语法

- String 类型
  - 字符串（String）类型指的是由单引号 '' 或双引号 "" 包裹的一段文本，例如 '123'、"abc"。需特别留意的是，单引号和双引号仅是定义字符串的不同形式，并非字符串本身的组成部分。
  - 在定义字符串时，若字符串中包含引号，可运用反斜杠 \ 对字符串中的引号进行转义，或者选用与字符串中引号类型不同的引号来定义该字符串，示例代码如下。

```
1. var str = "Let's have a cup of coffee.";  // 双引号中包含单引号
2. var str = 'He said "Hello" and left.';    // 单引号中包含双引号
3. var str = 'We\'ll never give up.';        // 使用反斜杠转义字符串中的单引号
```


2. 语法

- Number 类型
  - 数值（Number）类型用于定义数值。在 JavaScript 中，不区分整数和小数（浮点数），统一采用 Number 类型表示，示例代码如下。

```
1. var num1 = 123;     // 整数
2. var num2 = 3.14;    // 浮点数
```

  - Number 类型所能定义的数值并不是无限的，JavaScript 中的 Number 类型只能表示 -(253 - 1) 到 (253 -1) 之间的数值。对于一些极大或者极小的数，也可以通过科学记数法来表示，示例代码如下。

```
1. var y=123e5;      // 123 乘以 10 的 5 次方，即 12300000
2. var z=123e-5;     // 123 乘以 10 的 -5 次方，即 0.00123
```

  - 此外，Number 类型中存在一些特殊值，分别为 Infinity、-Infinity 和 NaN，具体如下：
    - Infinity：用于表示正无穷大的数值，通常指大于 1.7976931348623157e+308 的数。
    - -Infinity：用于表示负无穷大的数值，通常指小于 5e-324 的数。
    - NaN：即非数值（Not a Number 的缩写），用于表示无效或未定义的数学运算结果，例如 0 除以 0。
  - 当某次计算的结果超出 JavaScript 中 Number 类型的取值范围时，该数值将自动转换为无穷大，其中正数转换为 Infinity，负数转换为 -Infinity。当某次计算的结果超出 JavaScript 中 Number 类型的取值范围时，该数值将自动转换为无穷大，其中正数转换为 Infinity，负数转换为 -Infinity。

2. 语法

- Boolean 类型
  - 布尔（Boolean）类型仅包含两个值，即 true（真）和 false（假），在条件判断中应用极为广泛。在使用时不仅能够直接使用 true 或 false 来定义布尔类型的变量，还可以通过某些表达式获取布尔类型的值，代码示例如下。

```
1. var a = true;   // 定义一个布尔值 true
2. var b = false;  // 定义一个布尔值 false
3. var c = 2 > 1;  // 表达式 2 > 1 成立，其结果为“真（true）”，所以 c 的值为布尔类型的 true
4. var d = 2 < 1;  // 表达式 2 < 1 不成立，其结果为“假（false）”，所以 c 的值为布尔类型的 false
```

- Null 类型
  - Null 是一种仅含单一值的特殊数据类型，它代表“空”值，意味着不存在任何实际的值，可用于定义空对象指针。运用 typeof 操作符查看 Null 的类型，会发现其类型为 Object，这表明 Null 实际上是 Object（对象）的一个特殊值。故而，可以通过将变量赋值为 Null 来创建一个空对象。

2. 语法

- Undefined 类型
  - Undefined 是一种仅具有单一值的特殊数据类型，用于表示未定义状态。当声明一个变量却未为其赋值时，该变量的默认值即为 Undefined，代码示例如下。

```
1. var num;
2. console.log(num);  // 输出 undefined
```

  - 当运用 typeof 操作符来查看未赋值变量的类型时，会发现这些变量的类型同样为 undefined。而对于未声明的变量，使用 typeof 操作符查看其类型，也会得到 undefined 的结果，示例代码如下。

```
1. var message;
2. console.log(typeof message);  // 输出 undefined
3. console.log(typeof name);     // 输出 undefined
```

2. 语法

- Object 类型
  - 在 JavaScript 里，对象（Object）类型属于由键值对构成的无序集合。若要定义对象类型，需使用花括号 {}，其语法格式如下。

```
{name1: value1, name2: value2, name3: value3, ..., nameN: valueN}
```

  - 其中，name1、name2、name3、…、nameN 为对象的键，value1、value2、value3、…、valueN 为与之对应的键值。
  - 在 JavaScript 里，对象类型的键均为字符串类型，而值可以是任意数据类型。若要获取对象中的某个值，可采用“对象名.键”的形式，其示例代码如下。

```
var person = {
    name: 'Bob',
    age: 20,
    tags: ['js', 'web', 'mobile'],
    city: 'Beijing',
    hasCar: true,
    zipcode: null
};
console.log(person.name);       // 输出 Bob
console.log(person.age);        // 输出 20
```


2. 语法

- Array 类型
  - 数组（Array）是按顺序排列的数据集合，其中的每个值被称作元素，且数组可以包含任意类型的数据。在 JavaScript 里，定义数组需使用方括号[ ]，数组中的各元素用逗号分隔，其示例代码如下。

```
[1, 2, 3, 'hello', true, null]
```

  - 另外，也可用 Array() 函数来创建数组，其示例代码如下。

```
var arr = new Array(1, 2, 3, 4);
console.log(arr);       // 输出 [1, 2, 3, 4]
```

  - 数组中的元素可通过索引进行访问。数组的索引从 0 起依次递增，即数组首个元素的索引为 0，第二个元素的索引为 1，第三个元素的索引为 2，以此类推，其示例代码如下。

```
var arr = [1, 2, 3.14, 'Hello', null, true];
console.log(arr[0]);  // 输出索引为 0 的元素，即 1
console.log(arr[5]);  // 输出索引为 5 的元素，即 true
console.log(arr[6]);  // 索引超出了范围，返回 undefined
```

2. 语法

2.3 变量与数据类型

- Function 类型
  - 函数（Function）指的是一段具备特定功能的代码块，其自身不会自动执行，需通过函数名进行调用方可运行，其示例代码如下。

```
1. function sayHello(name){
2.     return "Hello, " + name;
3. }
4. var res = sayHello("Peter");
5. console.log(res);  // 输出 Hello, Peter
```

  - 此外，函数能够存储于变量、对象和数组之中，并且可作为参数传递给其他函数，也能从其他函数返回，其示例代码如下。

```
1. var fun = function(){
2.     console.log("http://c.biancheng.net/js/");
3. }
4. function createGreeting(name){
5.     return "Hello, " + name;
6. }
7. function displayGreeting(greetingFunction, userName){
8.     return greetingFunction(userName);
9. }
10. var result = displayGreeting(createGreeting, "Peter");
11. console.log(result);  // 输出 Hello, Peter
```


2. 语法

2.4 运算符与表达式

- 算术运算符主要用于执行常见的数学运算，如加法、减法、乘法和除法等。以下表格详细列举了 JavaScript 所支持的算术运算符。

运算符	描述	示例
+	加法运算符	x + y 表示计算 x 加 y 的和
-	减法运算符	x - y 表示计算 x 减 y 的差
*	乘法运算符	x * y 表示计算 x 乘 y 的积
/	除法运算符	x / y 表示计算 x 除以 y 的商
%	取模（取余）运算符	x % y 表示计算 x 除以 y 的余数

```
1. var x = 10,
2.     y = 4;
3. console.log("x + y =", x + y);  // 输出：x + y = 14
4. console.log("x - y =", x - y);  // 输出：x - y = 6
5. console.log("x * y =", x * y);  // 输出：x * y = 40
6. console.log("x / y =", x / y);  // 输出：x / y = 2.5
7. console.log("x % y =", x % y);  // 输出：x % y = 2
```

2. 语法

- 赋值运算符用于为变量赋予值，以下表格罗列了 JavaScript 所支持的赋值运算符。

运算符	描述	示例
=	最简单的赋值运算符，将运算符右侧的值赋值给运算符左侧的变量	x = 10 表示将变量 x 赋值为 10
+=	先进行加法运算，再将结果赋值给运算符左侧的变量	x += y 等同于 x = x + y
-=	先进行减法运算，再将结果赋值给运算符左侧的变量	x -= y 等同于 x = x - y
*=	先进行乘法运算，再将结果赋值给运算符左侧的变量	x *= y 等同于 x = x * y
/=	先进行除法运算，再将结果赋值给运算符左侧的变量	x /= y 等同于 x = x / y
%=	先进行取模运算，再将结果赋值给运算符左侧的变量	x %= y 等同于 x = x % y

```
1. var x = 10;
2. x += 20;
3. console.log(x); // 输出：30
4. var x = 12,
5.     y = 7;
6. x -= y;
7. console.log(x); // 输出：5
8. x = 5;
9. x *= 25;
10. console.log(x); // 输出：125
11. x = 50;
12. x /= 10;
13. console.log(x); // 输出：5
14. x = 100;
15. x %= 15;
16. console.log(x); // 输出：10
```


2. 语法

- 在 JavaScript 中，“+”和“+=”运算符不仅可用于数学运算，还能用于字符串拼接，具体如下：
  - “+”运算符用于将其左右两侧的字符串拼接为一个整体；
  - “+=”运算符则是先对字符串进行拼接操作，随后将拼接结果赋值给该运算符左侧的变量。

```
1. var x = "Hello ";
2. var y = "World!";
3. var z = x + y;
4. console.log(z); // 输出：Hello World!
5. x += y;
6. console.log(x); // 输出：Hello World!
```

2. 语法

- 自增和自减运算符用于对变量的值执行自增（+1）和自减（-1）操作。以下表格列举了 JavaScript 所支持的自增和自减运算符。

运算符	名称	影响
++x	自增运算符	将 x 加 1，然后返回 x 的值
x++	自增运算符	返回 x 的值，然后再将 x 加 1
--x	自减运算符	将 x 减 1，然后返回 x 的值
x--	自减运算符	返回 x 的值，然后将 x 减 1

```
1. var x;
2. x = 10;
3. console.log(++x);  // 输出：11
4. console.log(x);    // 输出：11
5. x = 10;
6. console.log(x++);  // 输出：10
7. console.log(x);    // 输出：11
8. x = 10;
9. console.log(--x);  // 输出：9
10. console.log(x);   // 输出：9
11. x = 10;
12. console.log(x--);  // 输出：10
13. console.log(x);    // 输出：9
```


2. 语法

- 比较运算符用于对其左右两侧的表达式进行比较，其运算结果为布尔值，仅存在两种可能，即 true 或 false。以下表格罗列了 JavaScript 所支持的比较运算符。

运算符	名称	示例
==	等于	x == y 表示如果 x 等于 y，则为真
===	全等	x === y 表示如果 x 等于 y，并且 x 和 y 的类型也相同，则为真
!=	不相等	x != y 表示如果 x 不等于 y，则为真
!==	不全等	x !== y 表示如果 x 不等于 y，或者 x 和 y 的类型不同，则为真
<	小于	x < y 表示如果 x 小于 y，则为真
>	大于	x > y 表示如果 x 大于 y，则为真
>=	大于或等于	x >= y 表示如果 x 大于或等于 y，则为真
<=	小于或等于	x <= y 表示如果 x 小于或等于 y，则为真

```
1. var x = 25;
2. var y = 35;
3. var z = "25";
4. console.log(x == z);  // 输出： true
5. console.log(x === z); // 输出： false
6. console.log(x != y);  // 输出： true
7. console.log(x !== z); // 输出： true
8. console.log(x < y);   // 输出： true
9. console.log(x > y);   // 输出： false
10. console.log(x <= y);  // 输出： true
11. console.log(x >= y);  // 输出： false
```

2. 语法

- 逻辑运算符常用于组合多个表达式，其运算结果为布尔值，仅有两种可能，即 true 或 false。以下表格列举了 JavaScript 支持的逻辑运算符。

运算符	名称	示例
&&	逻辑与	x && y 表示如果 x 和 y 都为真，则为真
||	逻辑或	x || y 表示如果 x 或 y 有一个为真，则为真
!	逻辑非	!x 表示如果 x 不为真，则为真

```
1. var year = 2021;
2. // 闰年可以被 400 整除，也可以被 4 整除，但不能被 100 整除
3. if((year % 400 == 0) || ((year % 100 != 0) && (year % 4 == 0))){
4.     console.log(year + " 年是闰年。");
5. } else{
6.     console.log(year + " 年是平年。");
7. }
```


2. 语法

- 三元运算符（亦称条件运算符）由一个问号和一个冒号构成，其语法格式如下。

```
1. 条件表达式 ? 表达式1 : 表达式2 ;
```

- 若“条件表达式”的结果为真（true），则执行“表达式1”内的代码；反之，则执行“表达式2”内的代码，其示例代码如下。

```
1. var x = 11,
2.     y = 20;
3. x > y ? console.log("x 大于 y") : console.log("x 小于 y");  // 输出：x 小于 y
```

2. 语法

2.4 运算符与表达式

- 位运算符用于对二进制位进行操作，JavaScript 所支持的位运算符如下表所示。

运算符	描述	示例
&	按位与：如果对应的二进制位都为 1，则该二进制位为 1	5 & 1 等同于 0101 & 0001 结果为 0001，十进制结果为 1
|	按位或：如果对应的二进制位有一个为 1，则该二进制位为 1	5 | 1 等同于 0101 | 0001 结果为 0101，十进制结果为 5
^	按位异或：如果对应的二进制位只有一个为 1，则该二进制位为 1	5 ^ 1 等同于 0101 ^ 0001 结果为 0100，十进制结果为 4
~	按位非：反转所有二进制位，即 1 转换为 0，0 转换为 1	~5 等同于 ~0101 结果为 1010，十进制结果为 -6
<<	按位左移：将所有二进制位统一向左移动指定的位数，并在最右侧补 0	5 << 1 等同于 0101 << 1 结果为 1010，十进制结果为 10
>>	按位右移（有符号右移）：将所有二进制位统一向右移动指定的位数，并拷贝最左侧的位来填充左侧	5 >> 1 等同于 0101 >> 1 结果为 0010，十进制结果为 2
>>>	按位右移零（无符号右移）：将所有二进制位统一向右移动指定的位数，并在最左侧补 0	5 >>> 1 等同于 0101 >>> 1 结果为 0010，十进制结果为 2

2. 语法

2.5 流程控制

- 条件判断语句是程序开发中常用的语句形式。与大多数编程语言一样，JavaScript 也具备条件判断语句。条件判断是指程序依据不同条件执行不同操作，例如依据年龄展示不同内容，根据布尔值 true 或 false 判断操作成功与否等。
- if-else
  - 在 JavaScript 里，if 语句是最为基础的条件判断语句，其语法格式如下：

```
if(条件表达式){
    // 要执行的代码;
}
```

  - 当条件表达式的计算结果为布尔值 true，也就是该条件表达式成立时，将执行{}内的代码，其示例代码如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <script type="text/javascript">
        var age = 20;
        if(age >= 18){      // 如果 age >= 18 的结果为 true，则执行下面 { } 中的代码
            alert("adult");
        }
    </script>
</body>
</html>
```

2. 语法

- if-else
  - if-else 语句作为 if 语句的升级版，不仅能够指定表达式成立时需执行的代码，还能明确表达式不成立时应执行的代码，其语法格式如下。

```
if(条件表达式){
    // 当表达式成立时要执行的代码
}else{
    // 当表达式不成立时要执行的代码
}
```

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <script type="text/javascript">
        var now = new Date();           // 获取当前的完整日期
        var dayOfWeek = now.getDay();   // 获取一个 0-6 之间的数字，用来表示当前是星期几，0 表示星期日、1 表示星期一、以此类推
        if (dayOfWeek > 0 && dayOfWeek < 6) {       // 判断：如果当前是星期一到星期五中的一天，则输出“Have a nice day!”，若不是则输出“Have a nice weekend!”
            alert("Have a nice day!");
        } else {
            alert("Have a nice weekend!");
        }
    </script>
</body>
</html>
```


2. 语法

- if-else
  - if 和 if-else 语句均仅包含一个条件表达式，而 if-else-if-else 语句作为它们的高级形式，允许定义多个条件表达式，并依据表达式的结果执行相应代码，其语法格式如下。

```
if (条件表达式 1) {
    // 条件表达式 1 为真实执行的代码
} else if (条件表达式 2) {
    // 条件表达式 2 为真实执行的代码
}
...
  else if (条件表达式N) {
    // 条件表达式 N 为真实执行的代码
} else {
    // 所有条件表达式都为假时要执行的代码
}
```

2. 语法

- if-else
  - 在执行 if-else if-else 语句时，一旦遇到条件表达式为真的情况，程序会即刻执行其后 {} 内的代码，随后退出整个 if-else if-else 语句结构。即便后续代码中存在其他为真的条件表达式，也不会再执行，其代码示例如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <script type="text/javascript">
        var now = new Date();          // 获取当前的完整日期
        var dayOfWeek = now.getDay();  // 获取一个 0-6 之间的数字，用来表示当前是星期几，0 表示星期日、1 表示星期一、以此类推
        if(dayOfWeek == 0) {           // 判断当前是星期几
            alert("星期日")
        } else if(dayOfWeek == 1) {
            alert("星期一")
        } else if(dayOfWeek == 2) {
            alert("星期二")
        } else if(dayOfWeek == 3) {
            alert("星期三")
        } else if(dayOfWeek == 4) {
            alert("星期四")
        } else if(dayOfWeek == 5) {
            alert("星期五")
        } else {
            alert("星期六")
        }
    </script>
</body>
</html>
```


2. 语法

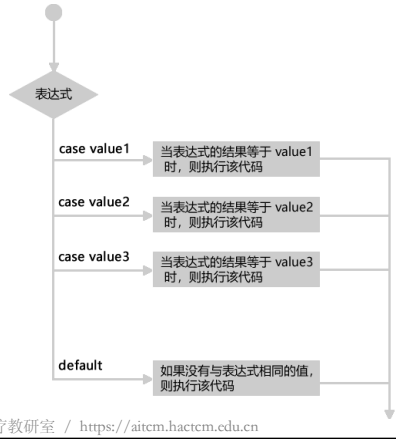
- switch-case
  - switch-case 语句与 if-else 语句的多分支结构相仿，二者均能够依据不同条件执行不同代码。不过，相较于 if-else 多分支结构，switch-case 语句更为简洁紧凑，执行效率也更高，其语法格式如下。

```
switch (表达式){
    case value1:
        statements1  // 当表达式的结果等于 value1 时，则执行该代码
        break;
    case value2:
        statements2  // 当表达式的结果等于 value2 时，则执行该代码
        break;
    ......
    case valueN:
        statementsN  // 当表达式的结果等于 valueN 时，则执行该代码
        break;
    default :
        statements  // 如果没有与表达式相同的值，则执行该代码
}
```

  - switch 语句会依据表达式的值，逐个与各 case 子句中的值进行比对：
    - 若两者相等，便会执行该 case 子句后续的语句段；一旦遇到 break 关键字，程序将跳出整个 switch 语句。
    - 若不相等，程序会继续对下一个 case 子句进行匹配。
    - switch 语句包含一个可选的 default 关键字，若在之前的所有 case 子句中均未找到符合条件的匹配项，程序将执行 default 关键字后面的语句段。
  - 在 switch 语句里，表达式会运用全等运算符（===），与各个 case 子句中的值进行匹配。鉴于采用的是全等运算符，各值的类型不会被自动转换。

2. 语法

- switch-case
  - switch 语句的执行流程如图所示。switch 语句按行依次执行，当它找到匹配的 case 子句时，不仅会执行该子句对应的代码，还会继续向后执行，直至 switch 语句结束。为避免这种情况，需在每个 case 子句末尾使用 break 语句跳出 switch 结构。



2. 语法

- switch-case

```
var id = 1;
switch (id) {
    case 1 :
        console.log("普通会员");
        break;  //停止执行，跳出switch
    case 2 :
        console.log("VIP会员");
        break;  //停止执行，跳出switch
    case 3 :
        console.log("管理员");
        break;  //停止执行，跳出switch
    default :  //上述条件都不满足时，默认执行的代码
        console.log("游客");
}
```

2. 语法

- switch-case
  - case 子句可省略语句。在此情况下，一旦匹配成功，无论下一个 case 条件是否满足，程序都将继续执行下一个 case 子句中的语句。

```
1. var id = 1;
2. switch (id) {
3.     case 1 :
4.     case 2 :
5.         console.log("VIP会员");
6.         break;
7.     case 3 :
8.         console.log("管理员");
9.         break;
10.    default :
11.        console.log("游客");
```


2. 语法

- switch-case
  - 在 switch 语句里，case 子句仅明确了执行的起始点，却未指明执行的终点。若 case 子句中未包含 break 语句，就会出现连续执行的现象，进而忽略后续 case 子句的条件限制，这极易破坏 switch 结构的逻辑。因此，当在函数中运用 switch 语句时，可借助 return 语句终止该 switch 语句，以避免代码继续执行。另外，default 作为 switch 子句，可置于 switch 内部的任意位置，且不会对其他 case 子句的正常执行造成影响，其代码示例如下。

```
1. var id = 1;
2. switch (id) {
3.     default :
4.         console.log("游客");
5.         break;
6.     case 1 :
7.         console.log("普通会员");
8.         break;
9.     case 2 :
10.        console.log("VIP会员");
11.        break;
12.    case 3 :
13.        console.log("管理员");
14.        break;
15.}
```

2. 语法

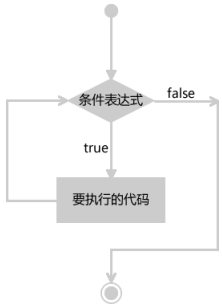
- switch-case
  - 以下是 default 语句与 case 语句的比较：
    - 语义差异：default 代表默认项，而 case 表示判例。
    - 功能扩展性：default 选项具有唯一性，不可进行扩展；相反，case 选项具有可扩展性，不受限制。
    - 异常处理方面：default 与 case 所扮演的角色不同，case 主要用于枚举，而 default 则用于异常处理。

2. 语法

- 循环是指重复执行某一操作。在编写代码时，常常会碰到需要反复执行的任务，比如遍历数据、重复输出特定字符串等。若逐行编写此类操作，不仅烦琐，还会降低效率。因此，对于这类重复性操作，应优先选择使用循环来实现。循环的核心目的在于反复执行特定代码段。合理运用循环，能够有效减轻编程负担，避免代码冗余，提升开发效率，同时也便于后续的代码维护工作。
- while
  - while 循环在每次循环之前，会先对条件表达式进行求值，如果条件表达式的结果为 true，则执行{}中的代码，如果条件表达式的结果为 false，则退出 while 循环，执行 while 循环之后的代码，其语法格式如下。

```
while（条件表达式）{
    // 要执行的代码
}
```

2. 语法




```
1. var i = 1;
2. while( i <= 5) {
3.     document.write(i+", ");
4.     i++;
5. }
```

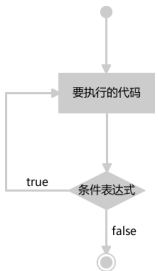
- 在编写循环语句时，务必保证条件表达式的结果能够为假（即布尔值 false）。因为一旦表达式的结果为 true，循环将持续执行，不会自动停止。通常把这种无法自动停止的循环称为“无限循环”或“死循环”。
- 若不慎造成无限循环，可能会致使浏览器或计算机出现卡死现象。

2. 语法

- do-while
  - do while 循环与 while 循环极为相似，区别在于，do while 循环会先执行循环内的代码，再对条件表达式进行判定。所以，无论条件表达式为真或为假，do while 循环至少会执行一次；而 while 循环则不然，若条件表达式为假，它会直接退出循环，其语法示例如下。

```
1. do {
2.     // 需要执行的代码
3. } while (条件表达式);
```

  - 与 while 循环相比，do while 循环还有一处差异，即 do while 循环需在末尾使用分号 “;” 结尾，而 while 循环则无需如此。



2. 语法


```
1. var i = 1;
2. do{
3.     document.write(i + " ");
4.     i++;
5. }while (i > 5);
```

- for
  - 当明确知晓循环次数时，使用 for 循环较为适宜，其语法格式如下。

```
1. for(initialization; condition; increment) {
2.     // 要执行的代码
3. }
```

  - for 循环中包含三个可选的表达式 initialization、condition 和 increment，其中：
    - initialization：为一个表达式或者变量声明，通常将该步骤称为“初始化计数器变量”，在循环过程中只会执行一次；
    - condition：为一个条件表达式，与 while 循环中的条件表达式功能相同，通常用来与计数器的值进行比较，以确定是否进行循环，通过该表达式可以设置循环的次数；
    - increment：为一个表达式，用来在每次循环结束后更新（递增或递减）计数器的值。

```
1. for (var i = 1; i <= 10; i++) {
2.     document.write(i + " ");
3. }
```


2. 语法

- for-in
  - for-in 循环是一种特殊的循环类型，作为普通 for 循环的变体，它主要用于遍历对象。借助 for-in 循环，能够依次迭代出对象的各个属性，其语法格式如下。

```
1. for (variable in object) {
2.     // 要执行的代码
3. }
```

  - 其中，variable 作为一个变量，在每次循环时会被赋予不同的值。可在{}内利用该变量开展一系列操作；object 是待遍历的对象，在每次循环中，object 对象的一个属性的键会被赋值给变量 variable，直至对象的所有属性遍历完毕，其代码示例如下。

```
1. // 定义一个对象
2. var person = {"name": "Clark", "surname": "Kent", "age": "36"};
3. // 遍历对象中的所有属性
4. for(var prop in person) {
5.     document.write("<p>" + prop + " = " + person[prop] + "</p>");
6. }
```

2. 语法

2.5 流程控制

- for-of
  - for-of 循环是 ECMAScript 6 新增的一种循环方式，它与 for-in 循环相似，同样是普通 for 循环的变体。借助 for-of 循环，能够便捷地遍历数组或其他可迭代对象，如字符串、对象等，其语法格式如下。

```
for (variable of iterable) {
    // 要执行的代码
}
```

  - 其中，variable 作为一个变量，在每次循环时会被赋予不同的值。可在后续的{}中利用该变量执行一系列操作；iterable 则是待遍历的内容，在每次循环里，iterable 中的一个值会被赋值给变量 variable，直至 iterable 中的所有值均被遍历完毕，其代码示例如下。

```
// 定义一个数组
var arr = ['a', 'b', 'c', 'd', 'e', 'f'];
// 使用 for of 循环遍历数组中的每个元素
for (var value of arr) {
    document.write(value + ", ");
}
document.write("<br>");
// 定义一个字符串
var str = "Hello World!";
// 使用 for of 循环遍历字符串中的每个字符
for (var value of str) {
    document.write(value + ", ");
}
document.write("<br>");
// 定义一个对象
var obj = {"name": "Clark", "surname": "Kent", "age": "36"};
// 使用 for of 循环遍历对象中的所有属性
for(var value in obj) {
    document.write(value + ", ");
}
```


2. 语法

2.6 函数

- 函数是一组可重复使用的代码块，用于执行特定任务、实现特定功能。在前面使用的 alert()、write() 即为 JavaScript 中的内置函数。
- 除内置函数，开发者还能够自行创建函数，并在需要之处调用该函数。如此操作，不仅能避免编写重复代码，还有益于代码的后期维护。
- 函数声明需以 “function” 关键字起始，其后紧跟待创建的函数名称，“function” 关键字与函数名称之间用空格分隔。函数名之后是一对括号 “( )”，用于定义函数所需的参数，若有多个参数，则用逗号 “,” 分隔，单个函数最多可包含 255 个参数。最后是一对花括号 “{ }”，用于定义函数体，即实现函数功能的代码，其语法格式如下。

```
function functionName(parameter_list) {
    // 函数中的代码
}
```

- 定义一个名为 sayHello() 的函数，该函数需接收一个名为 name 的参数。调用此函数时，会在页面中输出 “Hello ...”，其代码示例如下。

```
function sayHello(name){
    document.write("Hello " + name);
}
```

2. 语法

- 当成功定义一个函数后，便能够在当前文档的任意位置对其进行调用。函数的调用操作十分简便，仅需在函数名之后添加一对括号即可，例如 alert()、write()。需注意的是，若在定义函数时，函数名后的括号内指定了参数，那么在调用该函数时，也必须在括号中提供相应的参数，其代码示例如下。

```
1. function sayHello(name){
2.     document.write("Hello " + name);
3. }
4. // 调用 sayHello() 函数
5. sayHello('Demo');
```

- JavaScript 对于大小写敏感，所以在定义函数时 function 关键字一定要使用小写，而且调用函数时必须使用与声明时相同的大小写来调用函数。

2. 语法

- 定义函数时，可给函数的参数设定默认值。当调用该函数时，若未提供参数，便会将此默认值用作参数值，其代码示例如下。

```
1. function sayHello(name = "World"){
2.     document.write("Hello " + name);
3. }
4. sayHello();                // 输出：Hello World
5. sayHello('c.biancheng.net');     // 输出：Hello c.biancheng.net
```

- 在函数里，可运用 return 语句将一个值（即函数的运行结果）返回给调用该函数的程序。此返回值可以为任意类型，诸如数组、对象、字符串等。针对具有返回值的函数，通常会使用一个变量来接收其返回值，其代码示例如下。

```
1. function getSum(num1, num2){
2.     return num1 + num2;
3. }
4. var sum1 = getSum(7, 12);      // 函数返回值为：19
5. var sum2 = getSum(-5, 33);     // 函数返回值为：28
```

2. 语法

- return 语句通常定义在函数末尾。当函数执行至 return 语句时，会即刻停止运行，并返回到调用该函数的位置继续执行后续代码。此外，一个函数仅能有一个返回值。若需返回多个值，可将这些值置于一个数组中，然后返回该数组，其代码示例如下。

```
1. function division(dividend, divisor){
2.     var quotient = dividend / divisor;
3.     var arr = [dividend, divisor, quotient]
4.     return arr;
5. }
6. var res = division(100, 4)
7. document.write(res[0]);          // 输出：100
8. document.write(res[1]);          // 输出：4
9. document.write(res[2]);          // 输出：25
```


2. 语法

- 函数表达式与变量声明极为相似，它是另一种声明函数的形式，其语法格式如下。

```
1. var myfunction = function name(parameter_list){
2.     // 函数中的代码
3. };
```

- 参数说明如下：
  - myfunction：变量名，可通过该变量名调用等号之后的函数。
  - name：函数名，此参数可省略（通常情况下建议省略），若省略该参数，函数将成为匿名函数。
  - parameter_list：参数列表，单个函数最多可包含 255 个参数。

2. 语法


```
// 函数声明
function getSum(num1, num2) {
    var total = num1 + num2;
    return total;
}
// 函数表达式
var getSum = function(num1, num2) {
    var total = num1 + num2;
    return total;
};
```

- 上述示例中的两个函数是等价的，其功能、返回值和调用方法均相同。需要注意，在函数声明时，右花括号后无需加分号；若采用函数表达式，则应在表达式末尾加分号。

2. 语法

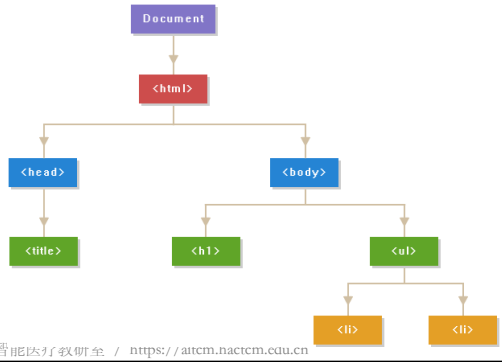
- 尽管函数声明与函数表达式在外观上极为相似，但其运行机制却存在差异，其代码示例如下。

```
declaration();          // 输出: function declaration
function declaration() {
    document.write("function declaration");
}
expression();           // 报错：Uncaught TypeError: undefined is not a function
var expression = function() {
    document.write("function expression");
};
```

- 如上述示例所示，若在函数表达式定义之前对其进行调用，程序将会抛出异常；而函数声明则能够正常运行。这是由于在程序执行前，JavaScript 会预先解析函数声明，所以无论在函数声明之前还是之后调用该函数，都是可行的。与之不同的是，函数表达式是把一个匿名函数赋值给一个变量，因此在程序尚未执行到该表达式时，函数实际上并未被定义，故而无法对其进行调用。

3. DOM

- 文档对象模型（Document Object Model，简称 DOM）是一种独立于平台和语言的模型，用于表示 HTML 或 XML 文档。该模型明确了文档的逻辑结构，同时规定了程序访问和操作文档的方式。
- 网页加载时，浏览器会自动构建当前页面的文档对象模型（DOM）。在 DOM 里，文档的各个部分（如元素、属性、文本等）会被组织成类似族谱的逻辑树结构。该树中每个分支的末端称作节点，且每个节点均为一个对象，具体如下图所示。



3. DOM

- 借助 DOM（文档对象模型），JavaScript 能够对 HTML 文档中的任意内容执行访问、修改、删除或添加操作。当浏览器加载 HTML 文档时，会自动创建一个 Document 对象，此对象作为 DOM 树中所有节点的根节点，可用于访问 HTML 文档内的所有元素。鉴于 Document 对象是 Window 对象的一部分，故而可以通过 window.document 来访问 Document 对象。下表列举了 Document 对象提供的属性及其描述。

属性	描述
document.activeElement	返回当前获取焦点的元素
document.anchors	返回对文档中所有 Anchor 对象的引用
document.applets	返回对文档中所有 Applet 对象的引用。注意: HTML5 已不支持 <applet> 元素
document.baseURI	返回文档的基础 URI
document.body	返回文档的 body 元素
document.cookie	设置或返回与当前文档有关的所有 cookie
document.doctype	返回与文档相关的文档类型声明 (DTD)
document.documentElement	返回文档的根节点
document.documentMode	返回浏览器渲染文档的模式
document.documentURI	设置或返回文档的位置

3. DOM

属性	描述
document.domain	返回当前文档的域名
document.domConfig	已废弃，返回 normalizeDocument() 被调用时所使用的配置
document.embeds	返回文档中所有嵌入内容（embed）的集合
document.forms	返回文档中所有 Form 对象的引用
document.images	返回文档中所有 Image 对象的引用
document.implementation	返回处理该文档的 DOMImplementation 对象
document.inputEncoding	返回文档的编码方式
document.lastModified	返回文档的最后修改日期
document.links	返回对文档中所有 Area 和 Link 对象的引用
document.readyState	返回文档状态（载入中）
document.referrer	返回载入当前文档的 URL
document.scripts	返回页面中所有脚本的集合
document.strictErrorChecking	设置或返回是否强制进行错误检查
document.title	返回当前文档的标题
document.URL	返回文档的完整 URL

3. DOM

- DOM操作的核心在于对“节点”进行管理，涵盖创建、插入、删除、替换、查询等操作。下表列举了 Document 对象所提供的方法及其描述。

方法	描述
document.addEventListener()	向文档中添加事件
document.adoptNode(node)	从另外一个文档返回 adapded 节点到当前文档
document.close()	关闭使用 document.open() 方法打开的输出流，并显示选定的数据
document.createAttribute()	为指定标签添加一个属性节点
document.createComment()	创建一个注释节点
document.createDocumentFragment()	创建空的 DocumentFragment 对象，并返回此对象
document.createElement()	创建一个元素节点
document.createTextNode()	创建一个文本节点
document.getElementsByClassName()	返回文档中所有具有指定类名的元素集合
document.getElementById()	返回文档中具有指定 id 属性的元素
document.getElementsByName()	返回具有指定 name 属性的对象集合
document.getElementsByTagName()	返回具有指定标签名的对象集合

3. DOM

- DOM操作的核心在于对“节点”进行管理，涵盖创建、插入、删除、替换、查询等操作。下表列举了 Document 对象所提供的方法及其描述。

方法	描述
document.importNode()	把一个节点从另一个文档复制到该文档以便应用
document.normalize()	删除空文本节点，并合并相邻的文本节点
document.normalizeDocument()	删除空文本节点，并合并相邻的节点
document.open()	打开一个流，以收集来自 document.write() 或 document.writeln() 方法的输出
document.querySelector()	返回文档中具有指定 CSS 选择器的第一个元素
document.querySelectorAll()	返回文档中具有指定 CSS 选择器的所有元素
document.removeEventListener()	移除文档中的事件句柄
document.renameNode()	重命名元素或者属性节点
document.write()	向文档中写入某些内容
document.writeln()	等同于 write() 方法，不同的是 writeln() 方法会在末尾输出一个换行符

3.


```
1. // 给整个文档添加「点击事件监听」——无论点击页面哪个位置，都会触发下方函数
2. document.addEventListener("click", function(){
3.     //  将当前「活动元素」（即被点击的元素）的HTML内容，替换掉<body>的全部内容
4.     //  document.activeElement：获取当前获得焦点或被点击的元素
5.     document.body.innerHTML = document.activeElement;
6.
7.     // 创建一个新的<div>元素（用于后续操作）
8.     var box = document.createElement('div');
9.     // 将新创建的<div>添加到<body>的末尾
10.    document.body.appendChild(box);
11.
12.    // 创建一个「id属性节点」，值为"myDiv"
13.    var att = document.createAttribute('id');
14.    att.value = "myDiv";
15.    // 将id属性绑定到<body>中的第一个<div>元素（即刚创建的box）
16.    document.getElementsByTagName('div')[0].setAttributeNode(att);
17.
18.    // 通过id获取这个<div>，并将其内容设置为「随机数」（Math.random()生成0-1的随机数）
19.    document.getElementById("myDiv").innerHTML = Math.random();
20.
21.    // 创建一个新的<button>元素（按钮）
22.    var btn = document.createElement("button");
23.    // 创建一个「文本节点」，内容为"按钮"
24.    var t = document.createTextNode("按钮");
25.    // 将文本节点放入按钮内部（否则按钮没有显示内容）
26.    btn.appendChild(t);
27.    // 将按钮添加到<body>的末尾
28.    document.body.appendChild(btn);
29.
30.    // 创建一个「onclick属性节点」，值为调用myfunction()的字符串
31.    var att = document.createAttribute('onclick');
32.    att.value = "myfunction()";
33.    // 将onclick属性绑定到<body>中的第一个<button>元素（即刚创建的按钮）
34.    //     点击按钮时，会触发myfunction函数
35.    document.getElementsByTagName('button')[0].setAttributeNode(att);
36.});

37.// 定义全局函数myfunction——点击按钮时会执行此函数
38.function myfunction(){
39.    // 弹出警告框，显示当前文档的标题（document.title）
40.    alert(document.title);
41.}
```

3. DOM

- 在DOM（文档对象模型）中，有三种方法可用于获取元素节点，分别是通过元素ID、标签名称和类名称进行获取。
- getElementById
  - DOM提供了名为getElementById的方法，该方法会返回与给定id属性值对应的元素节点对象。
  - getElementById是document对象独有的函数。在脚本代码中，函数名后必须跟随一对圆括号，括号内包含函数的参数。getElementById方法仅需一个参数，即想要获取的元素的id属性值，此id属性值须置于单引号或双引号中，其语法格式如下。

```
1. document.getElementById("purchases");
```

  - 该调用将返回一个对象，此对象对应于 document 对象中的一个唯一元素，即元素 id 属性值为 purchases 的节点对象。可使用 typeof 操作符对此进行验证，typeof 操作符可判定操作数是字符串、数值、函数、布尔值还是对象。
  - 实际上，文档中的每个元素均为一个对象。借助 DOM 提供的方法，能够获取任何对象。通常情况下，无需为文档中的每个元素都定义唯一的 id 值，因为 DOM 提供了另一种方法来获取那些没有 id 属性的对象。

3. DOM

- getElementsByTagName
  - getElementsByTagName方法返回一个对象数组，每个对象分别对应着文档里有着给定标签的一个元素。类似于getElementById，这个方法只有一个参数的函数，它的参数是标签的名字，其语法格式如下。

```
1. element.getElementsByTagName(tag);
```

  - 它与getElementById方法存在诸多相似之处，不过它返回的是一个数组。在编写脚本时，需留意将其与getElementById方法区分开来。该数组中的每个元素均为对象，可运用getElementsByTagName方法，结合循环语句与typeof操作符进行验证，其示例代码如下。

```
1. for(var i=0;i<document.getElementsByTagName("li").length;i++){
2.     alert(typeof document.getElementsByTagName("li")[i]);
3. }
```

  - 需要注意的是，即便在整个文档里该标签仅存在一个元素，getElementsByTagName方法返回的依旧是一个数组，只不过此数组的长度为1。

3. DOM

- getElementsByTagName
  - 在实际开发过程中，反复输入 “getElementsByTagName” 方法不仅烦琐，还会降低代码的可读性。为解决这一问题，可采用一个简便的方法，即将 “getElementsByTagName” 方法赋值给一个变量，其示例代码如下。

```
1. var items=document.getElementsByTagName("li");
2. for(var i=0;i<items.length;i++){
3.     alert(typeof items[i]);
4. }
```

  - getElementsByTagName该方法允许将通配符作为其参数，这意味着文档中的每个元素都将在该函数返回的数组中占有一个位置。通配符（“*”）必须置于引号内，以将其与乘法运算符区分开来。以下代码可用于获取文档中元素节点的总数，其示例代码如下。

```
1. alert(document.getElementsByTagName("*").length);
```

  - 当然，还可将 getElementsByTagName 与 getElementById 结合使用。例如，若要获取某个 id 属性值为“purchases”的元素所包含的列表项数量，实现代码如下。

```
1. var shop=document.getElementById("purchases");
2. var items=shop.getElementsByTagName("*");
3. alert(items.length);
```


3. DOM

- getElementsByClassName
  - 在 HTML 5 DOM 中，新增了 getElementsByClassName 方法。该方法可依据元素 class 属性中的类名来访问元素。getElementsByClassName 方法与 getElementsByTagName 方法类似，同样仅接受一个参数，即类名，其语法格式如下。

```
1. element.getElementsByTagName(class);
```

  - 返回值与 getElementsByTagName 方法的返回值相似，均为包含具有相同类名元素的数组。以下代码将返回类名为 sale 的所有元素，其代码示例如下。

```
1. document.getElementsByClassName("sale");
```

  - getElementsByClassName方法同样能够查找具有多个类名的元素。若要指定多个类名，仅需在字符串参数中使用空格分隔各个类名，其代码示例如下。

```
1. alert(document.getElementsByClassName("import sale").length)
```

  - 当执行上述代码时，能够清晰地知晓有多少个元素同时具备“import”和“sale”这两个类名。需要明确的是，即便在元素的 class 属性中，类名的顺序并非“import sale”，甚至元素还包含更多的类名，也不会对匹配该元素造成影响。

3. DOM

3.1 DOM概念与属性

- getElementsByClassName
  - getElementsByClassName方法可与 getElementsByTagName 方法、getElementById 方法结合使用。例如，若要确定 id 属性值为“purchases”的元素中，类名包含“sale”的元素数量，其代码示例如下。

```
1. var shop=document.getElementById("purchases");
2. var items=shop.getElementsByClassName ("sale");
3. alert(items.length);
```


3. DOM

3.2 DOM操作

- 前文介绍了 HTML 元素的获取方法，尽管这在查找和获取元素方面颇为实用，但 DOM 的真正强大之处在于能够借助 JavaScript 对文档进行动态修改。本节将着重阐述元素的添加、修改和删除操作。
- 增加元素
  - 若需向 HTML 中添加新元素，首先要创建该元素，随后将其追加到已有的元素中。document.createElement() 方法和 document.createTextNode() 方法分别用于创建新的 Element 节点和 Text 节点，而 node.appendChild()、node.insertBefore() 和 node.replaceChild() 方法则可将创建好的元素添加到文档中，具体实现步骤如下。
  - ① 首先，需创建一个新元素，例如 <p>，其代码如下所示。

```
1. var para=document.createElement("p");
```

  - ② 若需向 <p> 元素添加文本内容，需先创建一个文本节点，其代码如下所示。

```
1. var node=document.createTextNode("这是创建的新段落。");
```

3. DOM

- 增加元素
  - ③ 随后，将该文本节点追加至最初创建的 <p> 元素内，其代码如下所示。

```
1. para.appendChild(node);
```

  - ④ 最后，需向一个现有元素追加此新建元素，其代码示例如下。

```
1. var element=document.getElementById("div1");
2. element.appendChild(para);
```


3. DOM

- 修改元素
  - 元素修改涵盖两方面内容：一是元素内容的修改，二是元素属性的修改。
  - ① 修改元素内容
  - 修改元素内容最简单的方法是使用 innerHTML 属性。借助该属性，能够对元素内容进行重新赋值，进而实现修改元素内容的目的，其语法格式如下。

```
1. document.getElementById(id).innerHTML=new HTML;
```

  - 例如，若要修改某段落的内容可使用innerHTML属性，其代码示例如下。

```
1. <!DOCTYPE html>
2. <html>
3.     <head>
4.     <meta charset="utf-8" />
5.     <title>改变元素内容</title>
6.     <script>
7.         document.getElementById("p1").innerHTML="New text!";
8.     </script>
9. </head>
10. <body>
11.     <p id="p1">Hello World!</p>
12. </body>
13. </html>
```

3. DOM

- 修改元素
  - ② 改变元素属性
  - 获取所需元素后，即可着手获取其各项属性。其中，getAttribute函数专门用于获取元素属性；相应地，也可使用setAttribute函数更改元素节点的属性值。
  - getAttribute是一个仅接收一个参数的函数，该参数为待查询属性的名称，其语法格式如下。

```
1. object.getAttribute(attribute);
```

  - getAttribute方法并不属于document对象，仅能通过元素节点对象进行调用。例如，可将其与getElementsByTagName方法结合使用，以获取每个 <p> 元素的title属性,其代码示例如下。

```
1. var para=document.createElement("p");
2.     for(var i=0;i<para.length;i++){
3.     alert(para[i].getAttribute("title"));
4. }
```


3. DOM

- 修改元素
  - setAttribute() 方法用于设置属性，它允许修改属性节点的值。与 getAttribute 方法相同，该方法仅适用于元素节点。其语法格式如下。

```
1. object.setAttribute(attribute,value);
```

  - 以下示例将展示 setAttribute() 方法如何更改元素的 title 属性，其代码示例如下。

```
1. var shop=document.getElementById("purchases");
2. alert(shop.getAttribute("title"));
3. shop.setAttribute("title","a list");
4. alert(shop.getAttribute("title"));
```

  - 执行上述代码后，将弹出两个 alert 对话框。第一个对话框显示的内容为空白或“null”；第二个对话框将显示“a list”消息。在设置节点的 title 属性时，该属性原本并不存在。这意味着 setAttribute 方法实际上执行了两项操作：首先创建该属性，然后为其赋值。若元素节点本身已存在待修改的属性，setAttribute 方法将直接覆盖该属性。

3. DOM

3.2 DOM操作

- 删除元素
  - 若需在HTML中删除元素，首先要获取该元素，接着获取其父元素，最后使用removeChild方法将该元素删除，具体实现步骤如下。
  - ① 获取指定元素，例如，若要获取 id 属性值为 div1 的元素，其代码示例如下。

```
1. object.setAttribute(attribute,value);
```

  - ② 获取该元素的父元素，其代码示例如下。

```
1. var parent=document.getElementById("div1");
```

  - ③ 从父元素中移除该元素，其代码示例如下。

```
1. parent.removeChild(child);
```

  - 通过上述操作，成功将 id 属性值为 p1 的元素从 HTML 中移除。若能在不引用父元素的情况下直接删除该元素，便可大幅精简代码。然而，DOM 必须先获取待删除元素及其父元素，才会执行元素删除操作。因此，可对上述代码进行优化以实现元素删除，其代码示例如下。

```
1. var child=document.getElementById("p1");
2. child.parentNode.removeChild(child);
```


4. BOM

4.1 BOM概念与属性

- 浏览器对象模型（Browser Object Model，简称 BOM）作为 JavaScript 的重要组成部分，赋予了 JavaScript 程序与浏览器进行交互的能力。window 对象作为 BOM 的核心，代表着当前浏览器窗口，它提供了一系列用于操作或访问浏览器的方法和属性。此外，JavaScript 中的所有全局对象、函数以及变量均隶属于 window 对象，甚至前文所介绍的 document 对象同样属于 window 对象，下表罗列了 window 对象所提供的属性及其描述。

属性	描述
closed	返回窗口是否已被关闭
defaultStatus	设置或返回窗口状态栏中的默认文本
document	对 Document 对象的只读引用
frames	返回窗口中所有已经命名的框架集合，集合由 Window 对象组成，每个 Window 对象在窗口中含有一个 <frame> 或 <iframe> 标签
history	对 History 对象的只读引用，该对象中包含了用户在浏览器中访问过的 URL
innerHeight	返回浏览器窗口的高度，不包含工具栏与滚动条
innerWidth	返回浏览器窗口的宽度，不包含工具栏与滚动条
localStorage	在浏览器中以键值对的形式保存某些数据，保存的数据没有过期时间，会永久保存在浏览器中，直至手动删除
length	返回当前窗口中 <iframe> 框架的数量
location	引用窗口或框架的 Location 对象，该对象中包含当前 URL 的有关信息
name	设置或返回窗口的名称

4. BOM

- 下表罗列了 window 对象所提供的属性及其描述。

属性	描述
navigator	对 Navigator 对象的只读引用，该对象中包含当前浏览器的有关信息
opener	返回对创建此窗口的 window 对象的引用
outerHeight	返回浏览器窗口的完整高度，包含工具栏与滚动条
outerWidth	返回浏览器窗口的完整宽度，包含工具栏与滚动条
pageXOffset	设置或返回当前页面相对于浏览器窗口左上角沿水平方向滚动的距离
pageYOffset	设置或返回当前页面相对于浏览器窗口左上角沿垂直方向滚动的距离
parent	返回父窗口
screen	对 Screen 对象的只读引用，该对象中包含计算机屏幕的相关信息
screenLeft	返回浏览器窗口相对于计算机屏幕的 X 坐标
screenTop	返回浏览器窗口相对于计算机屏幕的 Y 坐标
screenX	返回浏览器窗口相对于计算机屏幕的 X 坐标
sessionStorage	在浏览器中以键值对的形式存储一些数据，数据会在关闭浏览器窗口或标签页之后删除
screenY	返回浏览器窗口相对于计算机屏幕的 Y 坐标
self	返回对 window 对象的引用
status	设置窗口状态栏的文本
top	返回最顶层的父窗口

4. BOM


```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <script type="text/javascript">
        window.defaultStatus = "JavaScript"
        document.write(window.defaultStatus + "<br>");    // 输出：JavaScript
        document.write(window.innerHeight + "<br>");      // 输出：314
        document.write(window.innerWidth + "<br>");       // 输出：539
        document.write(window.length + "<br>");           // 输出：0
        document.write(window.location + "<br>");         // 输出：file:///F:/code/index.html
        document.write(window.opener + "<br>");           // 输出：null
        document.write(window.outerHeight + "<br>");      // 输出：558
        document.write(window.outerWidth + "<br>");       // 输出：555
        document.write(window.parent + "<br>");           // 输出：[object Window]
        document.write(window.screenLeft + "<br>");       // 输出：2263
        document.write(window.screenTop + "<br>");        // 输出：401
        document.write(window.screenX + "<br>");          // 输出：2263
        document.write(window.screenY + "<br>");          // 输出：401
    </script>
</body>
</html>
```

4. BOM

- window 作为 BOM（浏览器对象模型）的顶层对象，代表着当前的浏览器窗口，它涵盖了其他所有的 BOM 对象。以下表格详细列举了 window 对象所提供的方法及其具体描述。

方法	描述
alert()	在浏览器窗口中弹出一个提示框，提示框中有一个确认按钮
atob()	解码一个 base-64 编码的字符串
btoa()	创建一个 base-64 编码的字符串
blur()	把键盘焦点从顶层窗口移开
clearInterval()	取消由 setInterval() 方法设置的定时器
clearTimeout()	取消由 setTimeout() 方法设置的定时器
close()	关闭某个浏览器窗口
confirm()	在浏览器中弹出一个对话框，对话框带有一个确认按钮和一个取消按钮
createPopup()	创建一个弹出窗口，注意：只有 IE 浏览器支持该方法
focus()	使一个窗口获得焦点
getSelection()	返回一个 Selection 对象，对象中包含用户选中的文本或光标当前的位置
getComputedStyle()	获取指定元素的 CSS 样式

4. BOM

4.1 BOM概念与属性

- window 作为 BOM（浏览器对象模型）的顶层对象，代表着当前的浏览器窗口，它涵盖了其他所有的 BOM 对象。以下表格详细列举了 window 对象所提供的方法及其具体描述。

方法	描述
matchMedia()	返回一个 MediaQueryList 对象，表示指定的媒体查询解析后的结果
moveBy()	将浏览器窗口移动指定的像素
moveTo()	将浏览器窗口移动到一个指定的坐标
open()	打开一个新的浏览器窗口或查找一个已命名的窗口
print()	打印当前窗口的内容
prompt()	显示一个可供用户输入的对话框
resizeBy()	按照指定的像素调整窗口的大小，即将窗口的尺寸增加或减少指定的像素
resizeTo()	将窗口的大小调整到指定的宽度和高度
scroll()	已废弃。可以使用 scrollTo() 方法来替代
scrollBy()	将窗口的内容滚动指定的像素
scrollTo()	将窗口的内容滚动到指定的坐标
setInterval()	创建一个定时器，按照指定的时长（以毫秒计）来不断调用指定的函数或表达式
setTimeout()	创建一个定时器，在经过指定的时长（以毫秒计）后调用指定函数或表达式，只执行一次
stop()	停止页面载入
postMessage()	安全地实现跨源通信

4.

属性

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <p id="show_tag">此处显示点击按钮的效果</p>
    <button onclick="myBtoa()">btoa()</button>
    <button onclick="myAtob()">atob()</button>
    <button onclick="myAlert()">alert()</button>
    <button onclick="myConfirm()">confirm()</button>
    <button onclick="myOpen()">open()</button>
    <button onclick="myMoveBy()">moveBy()</button>
    <button onclick="myMoveTo()">moveTo()</button>
    <button onclick="myPrint()">print()</button>
    <button onclick="myPrompt()">prompt()</button>
    <button onclick="myResizeBy()">resizeBy()</button>
    <button onclick="myClose()">close()</button>
    <script type="text/javascript">
        var ptag = document.getElementById('show_tag');
        var str;
        function myBtoa(){
            str = btoa("JavaScript");
            ptag.innerHTML = str;
        }
        function myAtob(){
            ptag.innerHTML = typeof str;
            if(str == undefined){
                ptag.innerHTML = "请先点击 btoa() 按钮";
                return;
            }
            ptag.innerHTML = atob(str);
        }
        function myAlert(){
            alert("这是一个提示框！");
        }
        function myConfirm(){
            var x;
            var r = confirm("按下按钮!");
            if (r == true){
                x = "你按下了\"确定\"按钮!";
            }
```


4.

属性

```
            else{
                x = "你按下了\"取消\"按钮!";
            }
            ptag.innerHTML = x;
        }
        var myWin;
        function myOpen(){
            if(myWin == undefined || (myWin != undefined && myWin.closed == true)){
                myWin = window.open('', '', 'width=200,height=100');
            }else{
                return;
            }
        }
        function myMoveBy(){
            if(myWin == undefined || myWin.closed == true) myOpen();
            myWin.moveBy(200, 200);
        }
        function myMoveTo(){
            if(myWin == undefined || myWin.closed == true) myOpen();
            myWin.moveTo(0, 0);
        }
        function myPrint(){
            print();
        }
        function myPrompt(){
            var name = prompt("请输入你的名字。")
            if (name != null && name != ""){
                ptag.innerHTML = "你好 " + name  + "! 今天感觉如何?";
            } else {
                ptag.innerHTML = "你没有输入任何内容";
            }
        }
        function myResizeBy(){
            if(myWin == undefined || myWin.closed == true) myOpen();
            myWin.resizeBy(100, 100);
        }
        function myClose(){
            if(myWin == undefined) return;
            if(myWin != undefined && myWin.closed == false) myWin.close();
        }
    </script>
</body>
</html>
```

5. 常用内置对象

- Date 对象作为 JavaScript 的内置对象，可访问计算机系统的时间。此外，该对象还提供了多种用于管理、操作和格式化时间或日期的方法。
- 在处理时间和日期之前，需先创建一个 Date 对象。与其他对象（如数组对象、字符串对象等）不同，Date 对象无法直接声明，必须借助 Date() 函数来定义。JavaScript 提供了四种不同的方法来创建 Date 对象，其语法格式如下。

```
1. var time = new Date();
2. var time = new Date(milliseconds);
3. var time = new Date(datestring);
4. var time = new Date(year, month, date[, hour, minute, second, millisecond]);
```

- 参数说明如下：
  - 不提供参数：若调用 Date() 函数时不提供参数，则创建一个包含当前时间和日期的 Date 对象；
  - milliseconds（毫秒）：若提供一个数值作为参数，则会将这个参数视为一个以毫秒为单位的时间值，并返回自 1970-01-01 00:00:00 起，经过指定毫秒数的时间，例如 new Date(5000) 会返回一个 1970-01-01 00:00:00 经过 5000 毫秒之后的时间；
  - datestring（日期字符串）：若提供一个字符串形式的日期作为参数，则会将其转换为具体的时间，日期的字符串形式有两种，如下所示：

5. 常用内置对象

- 参数说明如下：
  - YYYY/MM/dd HH:mm:ss（推荐）：若省略时间部分，则返回的 Date 对象的时间为 00:00:00；
  - YYYY-MM-dd HH:mm:ss：若省略时间部分，则返回的 Date 对象的时间为 08:00:00（加上本地时区），若不省略，在 IE 浏览器中会转换失败。
- 将具体的年月日、时分秒转换为 Date 对象，其中：
  - year：表示年，为了避免错误的产生，推荐使用四位的数字来表示年份；
  - month：表示月，0 代表 1 月，1 代表 2 月，以此类推；
  - date：表示月份中的某一天，1 代表 1 号，2 代表 2 号，以此类推；
  - hour：表示时，以 24 小时制表示，取值范围为 0 ~ 23；
  - minute：表示分，取值范围为 0 ~ 59；
  - second：表示秒，取值范围为 0 ~ 59；
  - millisecond：表示毫秒，取值范围为 0 ~ 999。

5. 常用内置对象

5.1 Date对象

- 其代码示例如下：

```
1. var time1 = new Date();
2. var time2 = new Date(1517356800000);
3. var time3 = new Date("2018/12/25 12:13:14");
4. var time4 = new Date(2020, 9, 12, 15, 16, 17);
5. document.write(time1 + "<br>");     // 输出：Fri Jul 23 2021 13:41:39 GMT+0800 (中国标准时间)
6. document.write(time2 + "<br>");     // 输出：Wed Jan 31 2018 08:00:00 GMT+0800 (中国标准时间)
7. document.write(time3 + "<br>");     // 输出：Tue Dec 25 2018 12:13:14 GMT+0800 (中国标准时间)
8. document.write(time4 + "<br>");     // 输出：Mon Oct 12 2020 15:16:17 GMT+0800 (中国标准时间)
```


5. 常用内置对象

5.1 Date对象

- 下表中列举了 Date 属性中提供的方法及其描述。

方法	描述	方法	描述
getDate()	从 Date 对象返回一个月中的某一天 (1 ~ 31)	getUTCFullYear()	根据通用时间从 Date 对象返回四位数的年份
getDay()	从 Date 对象返回一周中的某一天 (0 ~ 6)	getUTCHours()	根据通用时间返回 Date 对象的小时 (0 ~ 23)
getMonth()	从 Date 对象返回月份 (0 ~ 11)	getUTCMinutes()	根据通用时间返回 Date 对象的分钟 (0 ~ 59)
getFullYear()	从 Date 对象返回四位数字的年份	getUTCSeconds()	根据通用时间返回 Date 对象的秒钟 (0 ~ 59)
getYear()	已废弃，请使用 getFullYear() 方法代替	getUTCMilliseconds()	根据通用时间返回 Date 对象的毫秒(0 ~ 999)
getHours()	返回 Date 对象的小时 (0 ~ 23)	parse()	返回1970年1月1日午夜到指定日期（字符串）的毫秒数
getMinutes()	返回 Date 对象的分钟 (0 ~ 59)	setDate()	设置 Date 对象中月的某一天 (1 ~ 31)
getSeconds()	返回 Date 对象的秒数 (0 ~ 59)	setMonth()	设置 Date 对象中月份 (0 ~ 11)
getMilliseconds()	返回 Date 对象的毫秒(0 ~ 999)	setFullYear()	设置 Date 对象中的年份（四位数字）
getTime()	返回 1970 年 1 月 1 日至今的毫秒数	setYear()	已废弃，请使用 setFullYear() 方法代替
getTimezoneOffset()	返回本地时间与格林威治标准时间 (GMT) 的分钟差	setHours()	设置 Date 对象中的小时 (0 ~ 23)
getUTCDate()	根据通用时间从 Date 对象返回月中的一天 (1 ~ 31)	setMinutes()	设置 Date 对象中的分钟 (0 ~ 59)
getUTCDay()	根据通用时间从 Date 对象返回周中的一天 (0 ~ 6)	setSeconds()	设置 Date 对象中的秒钟 (0 ~ 59)
getUTCMonth()	根据通用时间从 Date 对象返回月份 (0 ~ 11)	setMilliseconds()	设置 Date 对象中的毫秒 (0 ~ 999)

5. 常用内置对象

5.1 Date对象

- 下表中列举了 Date 属性中提供的方法及其描述。

方法	描述
setTime()	以毫秒设置 Date 对象
setUTCDate()	根据通用时间设置 Date 对象中月份的一天 (1 ~ 31)
setUTCMonth()	根据通用时间设置 Date 对象中的月份 (0 ~ 11)
setUTCFullYear()	根据通用时间设置 Date 对象中的年份（四位数字）
setUTCHours()	根据通用时间设置 Date 对象中的小时 (0 ~ 23)
setUTCMinutes()	根据通用时间设置 Date 对象中的分钟 (0 ~ 59)
setUTCSeconds()	根据通用时间设置 Date 对象中的秒钟 (0 ~ 59)
setUTCMilliseconds()	根据通用时间设置 Date 对象中的毫秒 (0 ~ 999)
toSource()	返回该对象的源代码
toString()	把 Date 对象转换为字符串
toTimeString()	把 Date 对象的时间部分转换为字符串
toDateString()	把 Date 对象的日期部分转换为字符串
toGMTString()	已废弃，请使用 toUTCString() 方法代替
toUTCString()	根据通用时间，把 Date 对象转换为字符串

方法	描述
toLocaleString()	根据本地时间格式，把 Date 对象转换为字符串
toLocaleTimeString()	根据本地时间格式，把 Date 对象的时间部分转换为字符串
toLocaleDateString()	根据本地时间格式，把 Date 对象的日期部分转换为字符串
UTC()	根据通用时间返回 1970 年 1 月 1 日 到指定日期的毫秒数
valueOf()	返回 Date 对象的原始值

5. 常用内置对象

5.1 Date对象

- 其代码示例如下：

```
1. var time = new Date();
2. document.write(time.getDate() + "<br>");             // 输出：23
3. document.write(time.getDay() + "<br>");              // 输出：5
4. document.write(time.getFullYear() + "<br>");         // 输出：2021
5. document.write(time.getHours() + "<br>");            // 输出：16
6. document.write(time.getMonth() + "<br>");            // 输出：6
7. document.write(time.getTime() + "<br>");             // 输出：1627028869285
8. document.write(time.getUTCDate() + "<br>");          // 输出：23
9. document.write(time.toDateString() + "<br>");        // 输出：Fri Jul 23 2021
10. document.write(time.toString() + "<br>");           // 输出：Fri Jul 23 2021 16:29:57 GMT+0800 （中国标准时间）
11. document.write(time.toLocaleDateString() + "<br>"); // 输出：2021/7/23
12. document.write(time.toLocaleTimeString() + "<br>"); // 输出：下午4:31:00
13. document.write(time.toLocaleString() + "<br>");     // 输出：2021/7/23下午4:31:00
```

5. 常用内置对象

5.2 Math对象

- Math 作为 JavaScript 的内置对象，提供了一系列数学领域常用的常量值与函数，可用于执行常见的数学计算，如计算平均数、求绝对值以及四舍五入等操作。
- 调用 Math 对象的属性和方法时，无需预先借助 new 运算符创建该对象，直接将 Math 作为对象进行调用即可，其代码示例如下。

```
1. var pi_val = Math.PI;                // 数学中 π 的值：3.141592653589793
2. var abs_val = Math.sin(-5.35);       // -5.35 的绝对值：5.35
```


5. 常用内置对象

5.2 Math对象

- Math对象
  - Math对象的属性均为只读常数，下表中列举了 Math 对象中提供的属性及其描述信息。

属性	描述
E	返回算术常量 e，即自然对数的底数（约等于 2.718）
LN2	返回 2 的自然对数（约等于 0.693）
LN10	返回 10 的自然对数（约等于 2.302）
LOG2E	返回以 2 为底的 e 的对数（约等于 1.443）
LOG10E	返回以 10 为底的 e 的对数（约等于 0.434）
PI	返回圆周率 π（约等于 3.14159）
SQRT1_2	返回返回 2 的平方根的倒数（约等于 0.707）
SQRT2	返回 2 的平方根（约等于 1.414）

```
1. document.write(Math.E + "<br>");          // 输出：2.718281828459045
2. document.write(Math.LN2 + "<br>");        // 输出：0.6931471805599453
3. document.write(Math.LN10 + "<br>");       // 输出：2.302585092994046
4. document.write(Math.LOG2E + "<br>");      // 输出：1.4426950408889634
5. document.write(Math.LOG10E + "<br>");     // 输出：0.4342944819032518
6. document.write(Math.PI + "<br>");         // 输出：3.141592653589793
7. document.write(Math.SQRT1_2 + "<br>");    // 输出：0.7071067811865476
8. document.write(Math.SQRT2 + "<br>");      // 输出：1.4142135623730951
```

5. 常用内置对象

5.2 Math对象

- 下表中列举了 Math 对象中提供的方法及其描述信息。

方法	描述	方法	描述
abs(x)	返回 x 的绝对值	expm1(x)	返回 exp(x) - 1 的值
acos(x)	返回 x 的反余弦值	floor(x)	对 x 进行向下取整，即返回小于 x 的最大整数
acosh(x)	返回 x 的反双曲余弦值	fround(x)	返回最接近 x 的单精度浮点数
asin(x)	返回 x 的反正弦值	hypot([x, [y, [...]]])	返回所有参数平方和的平方根
asinh(x)	返回 x 的反双曲正弦值	imul(x, y)	将参数 x、y 分别转换位 32 位整数，并返回它们相乘后的结果
atan(x)	返回 x 的反正切值	log(x)	返回 x 的自然对数
atanh(x)	返回 x 的反双曲正切值	log1p(x)	返回 x 加 1 后的自然对数
atan2(y,x)	返回 y/x 的反正切值	log10(x)	返回 x 以 10 为底的对数
cbrt(x)	返回 x 的立方根	log2(x)	返回 x 以 2 为底的对数
ceil(x)	对 x 进行向上取整，即返回大于 x 的最小整数	max([x, [y, [...]]])	返回多个参数中的最大值
clz32(x)	返回将 x 转换成 32 无符号整形数字的二进制形式后，开头 0 的个数	min([x, [y, [...]]])	返回多个参数中的最小值
cos(x)	返回 x 的余弦值	pow(x,y)	返回 x 的 y 次幂
cosh(x)	返回 x 的双曲余弦值	random()	返回一个 0 到 1 之间的随机数
exp(x)	返回算术常量 e 的 x 次方，即 E^x	round(x)	返回 x 四舍五入后的整数

5. 常用内置对象

5.2 Math对象

- 下表中列举了 Math 对象中提供的方法及其描述信息。

方法	描述
sign(x)	返回 x 的符号，即一个数是正数、负数还是 0
sin(x)	返回 x 的正弦值
sinh(x)	返回 x 的双曲正弦值
sqrt(x)	返回 x 的平方根
tan(x)	返回 x 的正切值
tanh(x)	返回 x 的双曲正切值
toSource()	返回字符串"Math"
trunc(x)	返回 x 的整数部分
valueOf()	返回 Math 对象的原始值

5. 常用内置对象

5.2 Math对象

- 其代码示例如下：

```
1. document.write(Math.abs(-3.1415) + "<br>");          // 输出：3.1415
2. document.write(Math.acos(0.5) + "<br>");             // 输出：1.0471975511965979
3. document.write(Math.ceil(1.45) + "<br>");            // 输出：2
4. document.write(Math.exp(1) + "<br>");                // 输出：2.718281828459045
5. document.write(Math.floor(5.99) + "<br>");           // 输出：5
6. document.write(Math.log(6) + "<br>");                // 输出：1.791759469228055
7. document.write(Math.max(4, 8, 1, 9) + "<br>");       // 输出：9
8. document.write(Math.min(4, 8, 1, 9) + "<br>");       // 输出：1
9. document.write(Math.random() + "<br>");              // 输出：0.9172594288928195
10. document.write(Math.pow(2, 3) + "<br>");            // 输出：8
11. document.write(Math.sign(-123) + "<br>");           // 输出：-1
12. document.write(Math.sqrt(125) + "<br>");            // 输出：11.180339887498949
```


5. 常用内置对象

5.3 RegExp对象

- JavaScript 字符串作为编程中使用最为频繁的数据类型之一，在诸多场景下都需对其进行操作，例如判断字符串是否为合法的 E-mail 地址、从字符串中截取指定部分等。正则表达式是用于匹配字符串或特殊字符的逻辑公式，它由特定字符组合而成，是一种用以表示某些规则的特殊字符串，能够表达对字符串数据的过滤逻辑。在 JavaScript 中，需借助 RegExp 对象来运用正则表达式。创建 RegExp 对象有以下两种方法，其语法结构如下。

```
1. var patt = new RegExp(pattern, modifiers);
2. var patt = /pattern/modifiers;
```

- pattern：正则表达式，按照正则表达式的语法定义的正则表达式
- modifiers：修饰符，用来设置字符串的匹配模式，可选值如下表所示

修饰符	描述
i	执行对大小写不敏感的匹配
g	执行全局匹配（查找所有的匹配项，而非在找到第一个匹配项后停止）
m	执行多行匹配
s	允许使用.匹配换行符
u	使用 Unicode 码的模式进行匹配
y	执行“粘性”搜索，匹配从目标字符串的当前位置开始

5. 常用内置对象

- 正则表达式由字母、数字、标点及一些特殊字符构成，例如/abc/、/(\d+)\.\d*/ 。可在正则表达式中使用的特殊字符如下表所示。

特殊字符	含义
\	转义字符，在非特殊字符之前使用反斜杠表示下一个字符是特殊字符，不能按照字面理解，例如\b表示一个字符边界；在特殊字符之前使用反斜杠则表示下一个字符不是特殊字符，应该按照字面理解。例如反斜杠本身，若要在正则表达式中定义一个反斜杠，则需要在反斜杠前再添加一个反斜杠\\。
^	匹配字符串的开头，如果设置了修饰符 m，则也可以匹配换行符后紧跟的位置。 例如“/^A/”并不会匹配“an A”中的“A”，但是会匹配“An E”中的“A”。
$	匹配字符串的末尾，如果设置了修饰符 m，则也可以匹配换行符之前的位置。 例如“/t$/”并不会匹配“eater”中的“t”，但是会匹配“eat”中的“t”。
*	匹配前一个表达式 0 次或多次，等价于 {0,}。例如“/bo*/”能够匹配“A ghost booooored”中的“boooo”和“A bird warbled”中的“b”，但是在“A goat grunted”中不会匹配任何内容。
+	匹配前面一个表达式 1 次或者多次，等价于 {1,}。例如“/a+/”能够匹配“candy”中的“a”和“caaaaaaandy”中所有的“a”，但是在“cndy”中不会匹配任何内容。
?	匹配前面一个表达式 0 次或者 1 次，等价于 {0,1}。例如“/e?le?/”能够匹配“angel”中的“el”，“angle”中的“le”以及“oslo”中的“l”。
.	匹配除换行符之外的任何单个字符。例如“/.n/”将会匹配“nay, an apple is on the tree”中的“an”和“on”。
(x)	匹配“x”并记住这一匹配项，这里的括号被称为捕获括号。
(?:x)	匹配“x”但是不记住匹配项，这里的括号被称为非捕获括号。
x(?=y)	当“x”后面跟着“y”时，匹配其中的“x”。例如“/Jack(?=Sprat)/”会匹配后面跟着“Sprat”的“Jack”，“/Jack(?=Sprat|Frost)/”会匹配后面跟着“Sprat”或者是“Frost”的“Jack”注意：无论是“Sprat”还是“Frost”都不是匹配结果的一部分。
(?<=y)x	当“x”前面是“y”时，匹配其中的“x”。例如“/(?<=Jack)Sprat/”会匹配前面未“Sprat”的“Jack”，“/(?<=Jack|Tom)Sprat/”会匹配前面为“Jack”或者“Tom”的“Sprat”。注意：无论是“Jack”和“Tom”都不是匹配结果的一部分。
x(?!y)	当“x”后面不是“y”时，匹配其中的“x”。 例如“/\d+(?!\.)/”会匹配“3.141”中的“141”，而不是“3.141”。
(?<!y)x	当“x”前面不是“y”时，匹配其中的“x”。
x|y	匹配“x”或者“y”。 例如“/green|red/”能够匹配“green apple”中的“green”和“red apple”中的“red”。

特殊字符	含义
{n}	n 是一个正整数，表示匹配前一个字符 n 次。例如“/a{2}/”不会匹配“candy”中的“a”，但是能够匹配“caandy”中所有的“a”，以及“caaandy”中的前两个“a”。
{n,}	n 是一个正整数，表示匹配前一个字符至少 n 次。例如“/a{2,}/”能够匹配“aa”、“aaaa”或“aaaaa”，但不会匹配“a”。
{n,m}	n 和 m 都是整数，表示匹配前一个字符至少 n 次，最多 m 次，如果 n 或 m 等于 0，则表示忽略这个值。例如“/a{1, 3}/”能够匹配“candy”中的“a”，“caandy”中的前两个“a”，“caaaaaaandy”中的前三个“a”。
[xyz]	转义序列，匹配 x、y 或 z，也可以使用破折号-来指定一个字符范围。例如“[abcd]”和“[a-d]”是一样的，它们都能匹配“brisket”中的“b”，“city”中的“c”。
[^xyz]	反向字符集，匹配除 x、y、z 以外的任何字符，通用也可以使用破折号-来指定一个字符范围。例如“[^abc]”和“[^a-c]”是一样的，它们都能匹配“brisket”中的“r”，“chop”中的“h”。
[\b]	匹配一个退格符，注意：不要和 \b 混淆。
\b	匹配一个单词的边界，即单词的开始或末尾。例如“/\bm/”能够匹配“moon”中的“m”，但不会匹配“imoon”中的“m”。
\B	匹配一个非单词边界。例如“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。
\cX	当 X 是 A 到 Z 之间的字符时，匹配字符串中的一个控制符。例如“/\cM/”能够匹配字符串中的“control-M(U+000D)”。
\d	匹配一个数字，等价于“[0-9]”。例如“/\d/”或者“/[0-9]/”能够匹配“B2 is the suite number.”中的“2”
\D	匹配一个非数字字符，等价于“[^0-9]”。 例如“/\D/”或者“/[^0-9]/”能够匹配“B2 is the suite number.”中的“B”。
\f	匹配一个换页符 (U+000C)。
\n	匹配一个换行符 (U+000A)。

5. 常用内置对象

5.3 RegExp对象

- 正则表达式由字母、数字、标点及一些特殊字符构成，例如/abc/、/(\d+)\.\d*/ 。可在正则表达式中使用的特殊字符如下表所示。

特殊字符	含义
\r	匹配一个回车符 (U+000D)。
\s	匹配一个空白字符，包括空格、制表符、换页符和换行符，等价于“[\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]”。例如“/\s\w*/”能够匹配“foo bar.”中的“bar”。
\S	匹配一个非空白字符，等价于“[^\f\n\r\t\v\u00a0\u1680\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]”。例如“/\S\w*/”能够匹配“foo bar.”中的“foo”。
\t	匹配一个水平制表符 (U+0009)。
\v	匹配一个垂直制表符 (U+000B)。
\w	匹配一个单字字符（字母、数字或者下划线），等价于“[A-Za-z0-9_]”。例如“/\w/”能够匹配“apple,”中的“a”，“$5.28,”中的“5”和“3D.”中的“3”。
\W	匹配一个非单字字符，等价于“[^A-Za-z0-9_]”。例如“/\W/”或者“/[^A-Za-z0-9_]/”能够匹配“50%.”中的“%”。
\n	获取最后的第 n 个匹配的值。比如“/apple(,)\sorange\1/”能够匹配“apple, orange, cherry, peach.”中的“apple, orange,”。
\0	匹配 NULL（U+0000）字符，不要在这后面跟其它小数，因为 \0<digits> 是一个八进制转义序列。
\xhh	匹配一个两位十六进制数（\x00-\xFF）表示的字符。
\uhhhh	匹配一个四位十六进制数表示的 UTF-16 代码单元。
\u{hhhh}或\u{hhhhh}	（仅在设置了修饰符 u 时）匹配一个十六进制数表示的 Unicode 字符。

5. 常用内置对象

- 在 JavaScript 的 RegExp 对象里，提供了一系列用于执行正则表达式的方法，具体如下表所示。

方法	描述
compile()	在 1.5 版本中已废弃，编译正则表达式
exec()	在字符串搜索匹配项，并返回一个数组，若没有匹配项则返回 null
test()	测试字符串是否与正则表达式匹配，匹配则返回 true，不匹配则返回 false
toString()	返回表示指定对象的字符串

- 此外，String 对象的一些方法也支持正则表达式，具体如下表所示。

方法	描述
search()	在字符串中搜索匹配项，并返回第一个匹配的结果，若没有找到匹配项则返回 -1
match()	在字符串搜索匹配项，并返回一个数组，若没有匹配项则返回 null
matchAll()	在字符串搜索所有匹配项，并返回一个迭代器（iterator）
replace()	替换字符串中与正则表达式相匹配的部分
split()	按照正则表达式将字符串拆分为一个字符串数组

5. 常用内置对象

- 除方法外，RegExp 对象还提供了一些属性，具体如下。

属性	描述
constructor	返回一个函数，该函数是一个创建 RegExp 对象的原型
global	判断是否设置了修饰符 "g"
ignoreCase	判断是否设置了修饰符 "i"
lastIndex	用于规定下次匹配的起始位置
multiline	判断是否设置了修饰符 "m"
source	返回正则表达式的匹配模式

```
1. var str = "Hello World!";
2. var reg = /[a-g]/g;
3. document.write(reg.exec(str) + "<br>");            // 输出：e
4. document.write(reg.test(str) + "<br>");            // 输出：true
5. document.write(reg.toString() + "<br>");           // 输出：/[a-g]/g
6. document.write(str.search(reg) + "<br>");          // 输出：1
7. document.write(str.match(reg) + "<br>");           // 输出：e,d
8. document.write(str.matchAll(reg) + "<br>");        // 输出：[object RegExp String Iterator]
9. document.write(str.replace(reg, "+") + "<br>");    // 输出：H+llo Worl+!
10. document.write(str.split(reg) + "<br>");          // 输出：H,llo Worl,!
```

5. 常用内置对象

- 当网页开始加载时，浏览器会自动构建当前页面的文档对象模型，并将文档的各个部分，如元素、属性、文本等，组织成类似族谱的逻辑树结构。逻辑树每个分支的末端被称作节点，每个节点都包含一个对象，即 Element 对象。
- 借助 Document 对象提供的方法，如 getElementsByTagName()、getElementById()、getElementsByClassName() 等，能够获取 Element 对象。而 Element 对象自身也提供了一系列方法和属性，用于操作文档中的元素及其属性。

5. 常用内置对象

- 下表罗列了 JavaScript Element 对象所提供的属性及其相关描述。

属性	描述	属性	描述
element.accessKey	设置或返回一个访问单选按钮的快捷键	element.lang	设置或者返回一个元素的语言
element.attributes	返回一个元素的属性数组	element.lastChild	返回元素的最后一个子元素
element.childNodes	返回元素的一个子节点的数组	element.namespaceURI	返回命名空间的 URI
element.children	返回元素中子元素的集合	element.nextSibling	返回指定元素之后的兄弟元素，两个元素在 DOM 树中位于同一层级（包括文本节点、注释节点）
element.classList	返回元素中类名组成的对象	element.nextElementSibling	返回指定元素之后的兄弟元素，两个元素在 DOM 树中位于同一层级（不包括文本节点、注释节点
element.className	设置或返回元素的 class 属性	element.nodeName	返回元素名称（大写）
element.clientHeight	返回内容的可视高度（不包括边框，边距或滚动条）	element.nodeType	返回元素的节点类型
element.clientWidth	返回内容的可视宽度（不包括边框，边距或滚动条）	element.nodeValue	返回元素的节点值
element.contentEditable	设置或返回元素的内容是否可编辑	element.offsetHeight	返回元素的高度，包括边框和内边距，但不包括外边距
element.dir	设置或返回一个元素中的文本方向	element.offsetWidth	返回元素的宽度，包括边框和内边距，但不包括外边距
element.firstChild	返回元素中的第一个子元素	element.offsetLeft	返回元素在水平方向的偏移量
element.id	设置或者返回元素的 id	element.offsetParent	返回距离该元素最近的进行过定位的父元素
element.innerHTML	设置或者返回元素的内容	element.offsetTop	返回元素在垂直方向的偏移量
element.isContentEditable	返回元素内容是否可编辑，如果可编辑则返回 true，否则返回 false	element.ownerDocument	返回元素的根元素（文档对象）

5. 常用内置对象

5.4 Element对象

- 下表罗列了 JavaScript Element 对象所提供的属性及其相关描述。

属性	描述
element.parentNode	返回元素的父节点
element.previousSibling	返回元素之前的兄弟元素，两个元素在 DOM 树中位于同一层级（包括文本节点、注释节点）
element.previousElementSibling	返回元素之前的兄弟元素，两个元素在 DOM 树中位于同一层级（不包括文本节点、注释节点）
element.scrollHeight	返回元素的完整高度（包括被滚动条隐蔽的部分）
element.scrollLeft	设置或返回元素滚动条距离元素左侧的距离
element.scrollTop	设置或返回元素滚动条距离元素上方的距离
element.scrollWidth	返回元素的完整宽度（包括被滚动条隐蔽的部分）
element.style	设置或返回元素的样式属性
element.tabIndex	设置或返回元素的标签顺序
element.tagName	以字符的形式返回元素的名称（大写）
element.textContent	设置或返回某个元素以及其中的文本内容
element.title	设置或返回元素的 title 属性
element.length	返回对象的长度

5. 常用内置对象

5.4 Element对象

- 下表罗列了 JavaScript Element 对象所提供的方法及其详细描述。

方法	描述
element.addEventListener()	为指定元素定义事件
element.appendChild()	为元素添加一个新的子元素
element.cloneNode()	克隆某个元素
element.compareDocumentPosition()	比较当前元素与指定元素在文档中的位置，返回值如下： • 1：表示两个元素没有关系，不属于同一文档； • 2：表示当前元素在指定元素之后； • 4：当前元素在指定元素之前； • 8：当前元素在指定元素之内； • 16：指定元素在当前元素之内； • 32：两个元素没有关系，或者它们是同一元素的两个属性。
element.focus()	使元素获得焦点
element.getAttribute()	通过属性名称获取指定元素的属性值
element.getAttributeNode()	通过属性名称获取指定元素得属性节点
element.getElementsByTagName()	通过标签名获取当前元素下的所有子元素的集合
element.getElementsByClassName()	通过类名获取当前元素下的子元素的集合
element.hasAttribute()	判断元素是否具有指定的属性，若存在则返回 true，不存在则返回 false
element.hasAttributes()	判断元素是否存在任何属性，若存在则返回 true，不存在则返回 false
element.hasChildNodes()	判断一个元素是否具有子元素，有则返回 true，没有则返回 false
element.hasFocus()	判断元素是否获得了焦点

方法	描述
element.hasFocus()	判断元素是否获得了焦点
element.insertBefore()	在已有子元素之前插入一个新的子元素
element.isDefaultNamespace()	如果指定 namespaceURI 是默认的则返回 true，否则返回 false。
element.isEqualNode()	检查两个元素是否相等
element.isSameNode()	检查当前元素与指定元素是否为同一元素
element.isSupported()	判断当前元素是否支持某个特性
element.normalize()	合并相邻的文本节点，并删除空的文本节点
element.querySelector()	根据 CSS 选择器，返回第一个匹配的元素
document.querySelectorAll()	根据 CSS 选择器，返回所有匹配的元素
element.removeAttribute()	从元素中删除指定的属性
element.removeAttributeNode()	从元素中删除指定的属性节点
element.removeChild()	删除一个子元素
element.removeEventListener()	移除由 addEventListener() 方法添加的事件
element.replaceChild()	替换一个子元素
element.setAttribute()	设置或者修改指定属性的值
element.setAttributeNode()	设置或者修改指定的属性节点
element.setUserData()	在元素中为指定键值关联对象
element.toString()	将元素转换成字符串
nodelist.item()	返回某个元素基于文档树的索引

5. 常用内置对象

5.4 Element对象

- 其代码示例如下：

```
1. <!DOCTYPE html>
2. <html lang="zh-CN">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>JavaScript</title>
6. </head>
7. <body onload="accesskey()">
8.     <a id="r" class="aaa bbb ccc" href="javascript:;">使用 Alt + r 访问该元素</a><br>
9.     <a id="g" href="javascript:;">使用 Alt + g 访问该元素</a>
10.    <script type="text/javascript">
11.        function accesskey(){
12.            document.getElementById('r').accessKey="r"
13.            document.getElementById('g').accessKey="g"
14.        }
15.        var ele = document.getElementById('r');
16.        console.log(ele.attributes);                          // 输出：NamedNodeMap {0: id, 1: href, id: id, href: href, length: 2}
17.        console.log(document.body.childNodes);                // 输出：NodeList(7) [text, a#r, br, text, a#g, text, script]
18.        console.log(ele.classList);                           // 输出：DOMTokenList(3) ["aaa", "bbb", "ccc", value: "aaa bbb ccc"]
19.        console.log(ele.className);                           // 输出：aaa bbb ccc
20.        console.log(ele.clientHeight);                        // 输出：DOMTokenList(3) ["aaa", "bbb", "ccc", value: "aaa bbb ccc"]
21.        console.log(ele.tagName);                             // 输出：A
22.        console.log(ele.compareDocumentPosition(document.getElementById('g')));     // 输出：4
23.        console.log(ele.getAttribute('href'));                // 输出：javascript:;
24.        console.log(ele.getAttributeNode('href'));            // 输出：href="javascript:;"
25.    </script>
26. </body>
27. </html>
```


5. 常用内置对象

5.5 History对象

- JavaScript 的 history 对象包含了用户在浏览器中的历史访问记录，这些记录不仅包含用户直接通过浏览器浏览的页面，还包含当前页面中通过 <iframe> 标签加载的页面。在实际运用中，可以通过 window 对象的 history 属性来获取 history 对象。由于 window 对象是全局对象，在使用 window.history 时，可省略 window 前缀。例如，window.history.go() 可简化为 history.go()。
- 下表罗列了 JavaScript history 对象里的常用属性及其详细描述。

属性	说明
length	返回浏览历史的数目，包含当前已经加载的页面。
scrollRestoration	利用浏览器特性，使在返回上一页或者下一页时，将页面滚动到之前浏览的位置，该属性有两个值，分别是 auto（表示滚动）与 manual（表示不滚动）。
state	返回浏览器在当前 URL 下的状态信息，如果没有调用过 pushState() 或 replaceState() 方法，则返回默认值 null。

```
1. <!DOCTYPE html>
2. <html lang="zh-CN">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>JavaScript</title>
6. </head>
7. <body>
8.     <script type="text/javascript">
9.         document.write(history.length + "<br>");
10.        document.write(history.scrollRestoration + "<br>");
11.        document.write(history.state + "<br>");
12.    </script>
13. </body>
14. </html>
```

5. 常用内置对象

5.5 History对象

- 下表罗列了 JavaScript history 对象的常用方法及其详细描述。

方法	说明
back()	参照当前页面，返回历史记录中的上一条记录（即返回上一页），也可以通过点击浏览器工具栏中的←按钮来实现同样的效果。
forward()	参照当前页面，前往历史记录中的下一条记录（即前进到下一页），也可以通过点击浏览器工具栏中的→按钮来实现同样的效果。
go()	参照当前页面，根据给定参数，打开指定的历史记录，例如 -1 表示返回上一页，1 表示返回下一页。
pushState()	向浏览器的历史记录中插入一条新的历史记录。
replaceState()	使用指定的数据、名称和 URL 来替换当前历史记录。

5. 常用内置对象

5.5 History对象

- 其代码示例如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <button onclick="myBack()">back()</button>
    <button onclick="myForward()">forward()</button>
    <button onclick="myGo()">go()</button>
    <button onclick="myPushState()">pushState()</button>
    <button onclick="myReplaceState()">replaceState()</button>
    <script type="text/javascript">
        function myBack(){
            history.back();
        }
        function myForward(){
            history.forward();
        }
        function myGo(){
            var num = prompt('请输入一个整数', '1');
            history.go(num);
        }
        function myPushState(){
            var state = { 'page_id': 1, 'user_id': 5 }
            var title = 'JavaScript'
            var url = 'index.html'
            history.pushState(state, title, url)
            console.log(history.state);
        }
        function myReplaceState(){
            var state = { 'page_id': 3, 'user_id': 5 }
            var title = 'history'
            var url = 'index.html'
            history.replaceState(state, title, url)
            console.log(history.state);
        }
    </script>
</body>
</html>
```

5. 事件系统

- 事件（event）指的是用户与网页交互时所触发的情况，诸如点击某个链接或按钮、在文本框内输入文本、按下键盘上的特定按键、移动鼠标等。当事件发生时，可借助 JavaScript 中的事件处理程序（也称事件监听器）来检测该事件，并执行特定程序。
- 通常而言，事件名称一般以单词“on”开头，例如点击事件 onclick、页面加载事件 onload 等。以下表格列举了 JavaScript 中的一些常用事件。

事件		描述
鼠标、键盘事件	onclick	点击鼠标时触发此事件
	ondblclick	双击鼠标时触发此事件
	onmousedown	按下鼠标时触发此事件
	onmouseup	鼠标按下后又松开时触发此事件
	onmouseover	当鼠标移动到某个元素上方时触发此事件
	onmousemove	移动鼠标时触发此事件
	onmouseout	当鼠标离开某个元素范围时触发此事件
	onkeypress	当按下并松开键盘上的某个键时触发此事件
	onkeydown	当按下键盘上的某个按键时触发此事件
	onkeyup	当放开键盘上的某个按键时触发此事件

5. 事件系统

- 通常而言，事件名称一般以单词“on”开头，例如点击事件 onclick、页面加载事件 onload 等。以下表格列举了 JavaScript 中的一些常用事件。

事件		描述
窗口事件	onabort	图片在下载过程中被用户中断时触发此事件
	onbeforeunload	当前页面的内容将要被改变时触发此事件
	onerror	出现错误时触发此事件
	onload	页面内容加载完成时触发此事件
	onmove	当移动浏览器的窗口时触发此事件
	onresize	当改变浏览器的窗口大小时触发此事件
	onscroll	当滚动浏览器的滚动条时触发此事件
	onstop	当按下浏览器的停止按钮或者正在下载的文件被中断时触发此事件
	oncontextmenu	当弹出右键上下文菜单时触发此事件
	onunload	改变当前页面时触发此事件

事件		描述
表单事件	onblur	当前元素失去焦点时触发此事件
	onchange	当前元素失去焦点并且元素的内容发生改变时触发此事件
	onfocus	当某个元素获得焦点时触发此事件
	onreset	当点击表单中的重置按钮时触发此事件
	onsubmit	当提交表单时触发此事件

5. 事件系统

- 事件需与 HTML 元素绑定后才可触发。为 HTML 元素绑定事件处理程序的方法众多，其中最为简便的方式是借助 HTML 事件属性直接绑定，诸如 onclick、onmouseover、onmouseout 等属性。
- 以 onclick 属性为例，借助该属性能够为指定的 HTML 元素定义鼠标点击事件（即当在该元素上单击鼠标左键时触发的事件），以下为其代码示例。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <button type="button" onclick="myBtn()">按钮</button>
    <script type="text/javascript">
        function myBtn(){
            alert("Hello World!");
        }
    </script>
</body>
</html>
```


5. 事件系统

- 除上述方法外，还可直接借助 JavaScript 提供的内置函数，为指定元素绑定事件处理程序。以下为代码示例。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <button type="button" id="myBtn">按钮</button>
    <script>
        function sayHello() {
            alert('Hello World!');
        }
        document.getElementById("myBtn").onclick = sayHello;
    </script>
</body>
</html>
```

5. 事件系统

- 通过调用对象的 addEventListener() 方法，同样能够实现事件注册。这是当前推荐采用的使用方式，其语法格式如下。

```
element.addEventListener(String type, Function listener, boolean useCaptrue);
```

- 参数说明如下：
  - type：注册事件的类型名称。需要注意，事件类型与事件属性有所区别，事件类型名称无前缀“on”。例如，事件属性“onclick”对应的事件类型为“click”。
  - listener：监听函数，也就是事件处理函数。当指定类型的事件发生时，该函数将被调用。调用此函数时，默认会向其传递唯一参数——event 对象。
  - useCaptrue：此为布尔值。若该值为 true，则指定的事件处理函数将在事件传播的捕获阶段触发；若为 false，则事件处理函数将在冒泡阶段触发。

5. 事件系统

- 使用 addEventListener() 为所有按钮注册 click 事件，具体实现步骤如下。
  - ① 调用 document 的 getElementsByTagName() 方法捕获所有按钮对象
  - ② 使用 for 语句遍历按钮集（btn），并使用 addEventListener() 方法分别为每一个按钮注册事件函数，获取当前对象所显示的文本。
- 其代码示例如下。

```
<button id="btn1" onclick="btn1();">按 钮 1</button>
<button id="btn2" onclick="btn2(event);">按 钮 2</button>
<script>
    var btn = document.getElementsByTagName("button"); //捕获所有按钮
    for(var i in btn){                    //遍历按钮集合
      btn[i].addEventListener("click", function(){
        alert(this.innerHTML);
    }, true);   //为每个按钮对象注册一个事件处理函数，定义在捕获阶段进行响应
    }
</script>
```

5. 事件系统

6.2 事件绑定

- 借助 addEventListener() 方法，既能为多个对象注册相同的事件处理函数，也能为同一个对象注册多个事件处理函数。为同一对象注册多个事件处理函数在模块化开发中颇具实用价值。例如，为段落文本注册 mouseover 和 mouseout 两个事件。当光标移至段落文本上方时，其背景将显示为蓝色；而当光标移出段落文本时，背景会自动变为红色。如此一来，无需破坏文档结构，为段落文本添加多个事件属性。其代码示例如下。

```
1. <p id="p1">为对象注册多个事件</p>
2. <script>
3.     var p1 = document.getElementById("p1");  //捕获段落元素的句柄
4.     p1.addEventListener("mouseover", function () {
5.         this.style.background = 'blue';
6.     }, true);  //为段落元素注册第1个事件处理函数
7.     p1.addEventListener("mouseout", function () {
8.         this.style.background = 'blue';
9.     }, true);  //为段落元素注册第2个事件处理函数
10.</script>
```


5. 事件系统

6.3 事件处理

- 通常而言，事件可划分为四大类别，即鼠标事件、键盘事件、表单事件和窗口事件，此外还有一些其他类型的事件。
- （1）onmouseover 事件
- onmouseover 事件是指当用户的鼠标指针移至元素上方时所触发的事件，其示例代码如下。

```
1. <!DOCTYPE html>
2. <html lang="zh-CN">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>JavaScript</title>
6. </head>
7. <body>
8.     <button type="button" onmouseover="alert('的鼠标已经移动到了该按钮上');">请将鼠标移动至此处</button><br>
9.     <a href="#" onmouseover="myEvent()">请将鼠标移动至此处</a>
10.    <script>
11.        function myEvent() {
12.            alert('的鼠标已经移动到了该链接上');
13.        }
14.    </script>
15.</body>
16.</html>
```

5. 事件系统

- （2） onmouseout 事件
- onmouseout 事件与 onmouseover 事件截然相反，该事件会在鼠标移出元素范围时触发，其示例代码如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <div style="width: 350px; height: 200px; border:1px solid black" id="myBox"></div>
    <script>
        function myEvent() {
            alert('的鼠标已经离开指定元素');
        }
        document.getElementById("myBox").onmouseout = myEvent;
    </script>
</body>
</html>
```


5. 事件系统

- （3） onkeydown 事件
- onkeydown 事件是指在用户按下键盘上的任意按键时所触发的事件，其示例代码如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <input type="text" onkeydown="myEvent()">
    <script>
        function myEvent() {
            alert("按下了键盘上的某个按钮");
        }
    </script>
</body>
</html>
```

5. 事件系统

- （4）onkeyup 事件
- onkeyup 事件是指当用户按下键盘上的某个按键，随后释放该按键（即完成一次按键的按下与松开动作）时所触发的事件。其示例代码如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <input type="text" onkeyup="myEvent()">
    <script>
        function myEvent() {
            alert("按下了键盘上的某个按钮，并将其释放了");
        }
    </script>
</body>
</html>
```

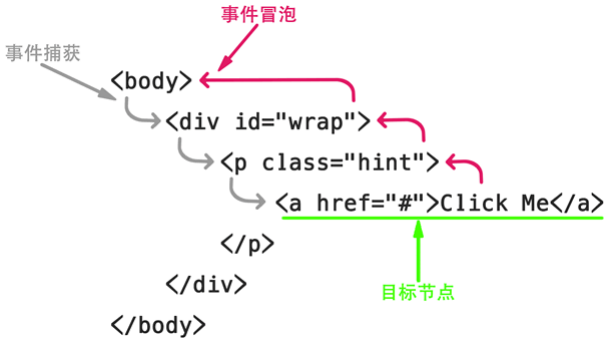

5. 事件系统

- 在 JavaScript 中，事件发生的顺序被称为“事件流”。当某个事件被触发时，会引发一系列的连锁反应。例如，在以下代码中，若为每个标签都定义了事件，当点击其中的 <a> 标签时，会发现绑定在 和 <p> 标签上的事件也被触发了。为了解决这一问题，微软和网景两家公司提出了两个不同的概念，即事件捕获与事件冒泡。
  - 事件捕获：由微软公司提出，此机制下事件从文档根节点（Document 对象）开始，朝着目标节点流动。在这一过程中，事件会依次经过目标节点的各个父级节点，并在这些节点上触发捕获事件，直至抵达事件的目标节点。
  - 事件冒泡：由网景公司提出，与事件捕获机制相反，事件会从目标节点起始，流向文档根节点。其间，事件会逐个经过目标节点的各个父级节点，并在这些节点上触发捕获事件，直至到达文档的根节点。整个过程宛如水中的气泡，自水底向上运动。

```
<body>
    <div id="wrap">
        <p class="hint">
            <a href="#">Click Me</a>
        </p>
    </div>
</body>
```

5. 事件系统

- 随后，为实现标准的统一，W3C 采取了一种折中的方案，将事件捕获与事件冒泡进行合并，即如今所采用的“先捕获后冒泡”机制，由此JavaScript事件分为三个阶段。
  - 捕获阶段：事件从父元素开始向目标元素传播，从 Window 对象开始传播。
  - 目标阶段：该事件到达目标元素或开始该事件的元素。
  - 冒泡阶段：这时与捕获阶段相反，事件向父元素传播，直到 Window 对象。



5. 事件系统

- 为什么要“先捕获，后冒泡”？
  - 这种顺序的设计，本质是为了满足浏览器的渲染逻辑和开发者的事件控制需求，具体原因如下：
  - 符合DOM树的“层级依赖”关系，DOM树是树形结构，父元素包含子元素。事件捕获的“向下”顺序，本质是从“全局”到“局部”的渗透。父元素先“感知”到事件（比如验证权限、修改全局状态），再让子元素处理具体逻辑，例如：点击一个按钮（子元素），父容器可以先检查“用户是否登录”——如果未登录，直接阻止事件继续传播（stopPropagation()），子元素根本不会收到事件。
  - 支持“事件委托”（Event Delegation），事件冒泡的“向上”顺序，是事件委托的核心机制，开发者只需在父元素上绑定一个事件监听器，就能处理所有子元素的事件（比如列表项的点击）。
  - 给开发者“分层控制”的能力，开发者可以选择在捕获阶段或冒泡阶段处理事件。
    - 捕获阶段：适合做“前置处理”，如权限验证、数据预处理。
    - 冒泡阶段：适合做“后置处理”，如更新UI、触发业务逻辑。

5. 事件系统

- 在事件捕获阶段，事件会从 DOM 树的最外层起始，逐个遍历目标节点的各个父节点，并触发父节点上的事件，直至抵达事件的目标节点。以上图中的代码为例，若单击其中的 <a> 标签，该事件将按照 document -> div -> p -> a 的顺序传递至 <a> 标签，其代码示例如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
    <style type="text/css">
        div, p, a {
            padding: 15px 30px;
            display: block;
            border: 2px solid #000;
            background: #fff;
        }
    </style>
</head>
<body>
    <div id="wrap">DIV
        <p class="hint">P
            <a href="#">A</a>
        </p>
    </div>
    <script>
        function showTagName() {
            alert("事件捕获: " + this.tagName);
        }
        var elems = document.querySelectorAll("div, p, a");
        for (let elem of elems) {
            elem.addEventListener("click", showTagName, true);
        }
    </script>
</body>
</html>
```


5. 事件系统

- 事件冒泡与事件捕获恰好相反。它起始于目标节点，随后沿着父节点逐层向上推进，依次触发各父节点上的事件，直至抵达文档根节点，这一过程犹如水底的气泡持续上浮，其代码示例如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
    <style type="text/css">
        div, p, a {
            padding: 15px 30px;
            display: block;
            border: 2px solid #000;
            background: #fff;
        }
    </style>
</head>
<body>
    <div onclick="alert('事件冒泡: ' + this.tagName)">DIV
        <p onclick="alert('事件冒泡: ' + this.tagName)">P
            <a href="#" onclick="alert('事件冒泡: ' + this.tagName)">A</a>
        </p>
    </div>
</body>
</html>
```

5. 事件系统

- 在了解事件捕获和事件冒泡之后，会发现这一特性并不十分友好。举例来说，当在某个节点上绑定事件时，本意是在点击该节点时触发此事件，但由于事件冒泡机制，该节点的事件却被其子元素触发了。此时，可使用 stopPropagation() 方法来阻止事件捕获和事件冒泡的发生，其语法格式如下。

```
1. event.stopPropagation();
```


5. 事件系统

- 需要注意的是，stopPropagation() 会阻止事件捕获和事件冒泡，但是无法阻止标签的默认行为，例如点击链接仍然可以打开对应网页。

```
1. <!DOCTYPE html>
2. <html lang="zh-CN">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>JavaScript</title>
6.     <style type="text/css">
7.         div, p, a {
8.             padding: 15px 30px;
9.             display: block;
10.            border: 2px solid #000;
11.            background: #fff;
12.        }
13.    </style>
14. </head>
15. <body>
16.    <div id="wrap">DIV
17.        <p class="hint">P
18.            <a href="#">A</a>
19.        </p>
20.    </div>
21.    <script>
22.        function showAlert(event) {
23.            alert("点击了 "+ this.tagName + " 标签");
24.            event.stopPropagation();
25.        }
26.        var elems = document.querySelectorAll("div, p, a");
27.        for(let elem of elems) {
28.            elem.addEventListener("click", showAlert);
29.        }
30.    </script>
31. </body>
32. </html>
```

5. 事件系统

- 此外，还可运用 stopImmediatePropagation() 方法来阻止同一节点同一事件的其他事件处理程序。例如，当为某个节点定义了多个点击事件时，事件触发后，这些事件会按照定义顺序依次执行。若其中一个事件处理程序使用了 stopImmediatePropagation() 方法，那么其余的事件处理程序将不再执行，其语法格式如下。

```
event.stopImmediatePropagation();
```


5. 事件系统

- 其代码示例如下。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
    <style type="text/css">
        div, p, a {
            padding: 15px 30px;
            display: block;
            border: 2px solid #000;
            background: #fff;
        }
    </style>
</head>
<body>
    <div onclick="alert('点击了 ' + this.tagName + ' 标签')">DIV
        <p onclick="alert('点击了 ' + this.tagName + ' 标签')">P
            <a href="#" id="link">A</a>
        </p>
    </div>
    <script>
        function sayHi() {
            alert("事件处理程序 1");
            event.stopImmediatePropagation();
        }
        function sayHello() {
            alert("事件处理程序 2");
        }
        // 为 id 为 link 的标签定义多个点击事件
        var link = document.getElementById("link");
        link.addEventListener("click", sayHi);
        link.addEventListener("click", sayHello);
    </script>
</body>
</html>
```

5. 事件系统

- 部分事件存在与之关联的默认操作。例如，单击某个链接时，页面会自动跳转至指定页面；单击提交按钮时，数据会被提交至服务器。若要阻止此类默认操作，可使用 preventDefault() 方法。其语法格式如下。

```
event.preventDefault();
```

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <title>JavaScript</title>
</head>
<body>
    <a href="http://c.biancheng.net/" id="link">链接</a>
    <script>
        var link = document.getElementById("link");
        link.addEventListener('click', function(event){
            event.preventDefault(); // 阻止链接跳转
        });
    </script>
</body>
</html>
```


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