JavaScript 讲义

中文版

Contents

1 入门 **3**
- 1.1 你的第一个程序 . . . 3
- 1.2 注释与代码风格 . . . 3

2 值、变量与类型 **4**
- 2.1 let 与 const . . . 4
- 2.2 数字与数学运算 . . . 4
- 2.3 类型与类型转换 . . . 5

3 字符串 **6**
- 3.1 Characters & length . . . 6
- 3.2 String methods . . . 6
- 3.3 Template literals . . . 7

4 条件判断 **8**
- 4.1 布尔值与比较 . . . 8
- 4.2 if / else if / else . . . 8
- 4.3 switch 语句 . . . 9

5 循环 **10**
- 5.1 for loops . . . 10
- 5.2 while loops . . . 11
- 5.3 for...of, break & continue . . . 11

6 函数 **13**
- 6.1 Parameters & return . . . 13
- 6.2 Arrow functions . . . 13
- 6.3 Scope . . . 14

7 数组 **15**
- 7.1 创建与修改数组 . . . 15
- 7.2 map、filter 与 reduce . . . 15
- 7.3 查找与排序 . . . 16

8 对象 **18**
- 8.1 Objects & properties . . . 18
- 8.2 Arrays of objects . . . 18
- 8.3 Looping over an object . . . 19

9 错误 **20**
- 9.1 Reading error messages . . . 20
- 9.2 try, catch & throw . . . 21

1 入门

1.1 ■ 你的第一个程序

程序 (program) 是给计算机的一串指令。在 JavaScript 中, 每条指令是一条语句 (statement), 计算机从上到下依次运行这些语句。

`console.log(...)` 把一个值打印到控制台 (console), 也就是代码下方的文字区域:

```
console.log("Hello, world!");
console.log(7 * 6);   // 42
```

- 文本要放在引号里:`"Hello"` 或 `'Hello'`。数字不用。
- 每条语句以分号 `;` 结尾。缺了分号时 JavaScript 往往能猜出来, 但写上会更清楚。
- JavaScript 区分大小写 (case-sensitive):`console.log` 能用,`Console.Log` 不能。

`console.log` 可以一次打印多个值, 它会在值之间加一个空格:

```
console.log("Total:", 3 + 4);   // Total: 7
console.log(1, 2, 3);           // 1 2 3
```

1.2 ■ 注释与代码风格

注释 (comment) 是写给人看的说明。程序运行时 JavaScript 会忽略它。

```
// A one-line comment starts with two slashes.
console.log("This line runs");   // a comment can follow code

/* A block comment
   can cover several lines. */
console.log("So does this one");
```

良好的风格 (style) 让代码容易读懂:

- 把 `{ }` 里面的行缩进 (indent) 两个或四个空格, 而且每次都用同样的方式。
- 给变量起能说明内容的名字: 用 `total`, 不要用 `t`。
- JavaScript 的名字用驼峰命名法 (camelCase):`firstName`、`scoreTotal`。

```
// Work out the area of a rectangle
const width = 4;
const height = 3;
console.log(width * height);   // 12
```

常见错误

- 文本的引号必须成对:`"Hi'` 是错误的。
- `console.log` 全部是小写:`Console.log` 和 `console.Log` 都会出错。
- 一行里 `//` 之后的内容都会被忽略, 所以 `// console.log(x)` 什么也不打印。

2 值、变量与类型

2.1 ■ let 与 const

变量 (variable) 是一个有名字、装着值的盒子。用 `let` 或 `const` 创建它, 然后用它的名字:

```
let score = 10;
score = score + 5;      // let: the value can change
console.log(score);     // 15

const name = "Mei";     // const: the value never changes
console.log(name);      // Mei
```

- `=` 的意思是**赋值** (assign): 先算出右边, 再把结果存进左边的盒子。
- 默认用 `const`。只有当值以后必须改变时才用 `let`。
- 给 `const` 赋新值是错误的:`name = "Tom"` 会让程序停下。

2.2 ■ 数字与数学运算

JavaScript 只有一种数字类型, 整数和小数都用它。算术运算符 (operator) 有:

运算符	含义	例子	结果
`+ -`	加、减	`7 - 2`	`5`
`* /`	乘、除	`7 / 2`	`3.5`
`%`	余数 (remainder)	`7 % 2`	`1`
`**`	乘方	`2 ** 3`	`8`

```
console.log(7 / 2);               // 3.5
console.log(Math.floor(7 / 2));   // 3   (whole-number division)
console.log(7 % 2);               // 1   (odd numbers leave 1)
console.log(2 + 3 * 4);           // 14  (* before +)
console.log((2 + 3) * 4);         // 20
```

`Math` 里有很多有用的函数:

```
console.log(Math.round(2.5));     // 3
console.log(Math.max(4, 9, 2));   // 9
console.log(Math.sqrt(49));       // 7
```

- `Math.random()` 给出一个从 0 到 1(不含 1) 的随机 (random) 数。`Math.floor(Math.random() * 6) + 1` 就是掷一次骰子, 得到 1 到 6。
- 小数以二进制存储, 所以 `0.1 + 0.2` 打印出 `0.30000000000000004`。显示结果之前先取整:`(0.1 + 0.2).toFixed(2)` 是 `"0.30"`。

2.3 ■ 类型与类型转换

每个值都有一个类型 (type)。`typeof` 告诉你是哪一种:

```
console.log(typeof 42);          // number
console.log(typeof "42");        // string
console.log(typeof true);        // boolean
console.log(typeof undefined);   // undefined
```

- 用 `let x;` 创建、没有给值的变量是 `undefined`。
- `null` 表示" 没有值", 是有意设置的。

`+` 遇到字符串时会**拼接**而不是相加, 所以要先把文本转换 (convert) 成数字:

```
console.log("5" + 1);            // 51  (joined)
console.log(Number("5") + 1);    // 6   (added)
console.log(parseInt("12px"));   // 12  (reads the number at the start)
console.log(String(9) + "!");    // 9!
```

- 不是数字的文本会得到 `NaN`(" 不是一个数 "):`Number("abc")` 是 `NaN`。
- `-`、`*` 和 `/` 会自动把文本转成数字 (`"6" * 2` 是 12), 但不要依赖这一点: 要有意识地转换。

常见错误

- `=` 是赋值, 不是比较。比较要用 `===`(第 4 章)。
- `"5" + 1` 是 `"51"`。用输入做数学运算之前, 先用 `Number(...)` 转换。
- `Math.round(2.5)` 是 3, 但 `Math.round(-2.5)` 是 -2: 它把.5 向较大的数取整。

3 字符串

3.1 ■ Characters & length

字符串是用引号包裹的文本。对于简单字母，每个字符有一个位置索引；首字符索引为 0。JavaScript 按 UTF-16 码点计数：一个表情符号可能占两个位置。

```
const word = "Script";
console.log(word.length);      // 6
console.log(word[0]);          // S
console.log(word[2]);          // r
console.log(word.at(-1));      // t  (counts back from the end)
console.log(word[word.length - 1]);   // t
```

- 末位索引为 `length - 1`。`word[6]` 访问越界时返回 `undefined`，而非报错。
- 字符串不可变：`word[0] = "s"` 无法修改原字符串，在严格模式下会抛出 TypeError。应构建新字符串替代。

使用 `+` 拼接字符串，该操作称为拼接：

```
const first = "Ada";
const last = "Lovelace";
console.log(first + " " + last);   // Ada Lovelace
```

3.2 ■ String methods

方法是属于某个值的函数，用点号调用。字符串方法返回**新值**，原值不变。

```
const city = "  Hong Kong ";
console.log(city.trim());                  // Hong Kong
console.log(city.trim().toUpperCase());    // HONG KONG
console.log(city.length);                  // 12  (the spaces are still
↪  there)
```

方法	含义	例子	结果
`.toUpperCase()` / `.toLowerCase()`	改变大小写	`"Hi".toLowerCase()`	`"hi"`
`.trim()`	去除两端空格	`" hi ".trim()`	`"hi"`
`.includes(t)`	是否包含 t?	`"cat".includes("a")`	`true`
`.indexOf(t)`	t 首次出现的位置，或 -1	`"cat".indexOf("t")`	`2`
`.slice(a, b)`	从 a 到（不含）b 的子串	`"Script".slice(0, 3)`	`"Scr"`
`.split(sep)`	拆分为数组	`"a,b,c".split(",")`	`["a","b","c"]`
`.replace(a, b)`	将第一个 a 替换为 b	`"cat".replace("c", "b")`	`"bat"`

```
const email = "mei@school.org";
const at = email.indexOf("@");
console.log(email.slice(0, at));          // mei
console.log(email.slice(at + 1));         // school.org
console.log(email.split("@"));            // ["mei","school.org"]
```

3.3 ■ Template literals

模板字面量用反引号（多数键盘上位于 1 键左侧）包裹。可在 `${ }` 内嵌入任意表达式，其值将直接插入文本中：

```
const name = "Mei";
const age = 17;
console.log(`${name} is ${age} years old.`);   // Mei is 17 years old.
console.log(`Next year: ${age + 1}`);          // Next year: 18
```

- 反引号也可以容纳多行文本。
- 这比许多 `+` 符号更易读：请将 `"Hi " + name + "!"` 与 `` `Hi ${name}!` `` 进行比较。

```
const price = 4.5;
console.log(`Two cost $${(price * 2).toFixed(2)}`);   // Two cost $9.00
```

常见错误

- `${ }` 仅在反引号内有效。在 `"Hi ${name}"` 中，文本按原样输出。
- 方法不会改变字符串本身：单独一行的 `city.trim()` 不会执行任何操作。请存储结果：`const clean = city.trim();`。
- `slice(a, b)` 在 `b` **之前**停止，就像最后一个索引是 `length - 1` 一样。

4 条件判断

4.1 ■ 布尔值与比较

布尔值 (Boolean) 只能是 `true` 或 `false`。比较 (comparison) 会得出一个布尔值:

运算符	含义	例子	结果
`===`	相等 (值相同且类型相同)	`5 === 5`	`true`
`!==`	不相等	`5 !== 3`	`true`
`< >`	小于、大于	`2 > 7`	`false`
`<= >=`	不大于、不小于	`7 >= 7`	`true`

```
console.log(5 === 5);      // true
console.log(5 === "5");    // false  (a number is not a string)
console.log(5 == "5");     // true   (== converts first: avoid it)
```

- 始终用 `===` 和 `!==`。旧的 `==` 会先转换类型, 结果常常出人意料。

用逻辑运算符 (logical operator) 把条件连起来:

- `&&`(并且): 只有**两边**都为真时才为真。
- `||`(或者): **至少一边**为真时就为真。
- `!`(非): 把 `true` 变成 `false`, 反之亦然。

```
const age = 15;
console.log(age >= 13 && age <= 19);   // true   (a teenager)
console.log(age < 5 || age > 65);      // false
console.log(!(age > 18));              // true
```

4.2 ■ if / else if / else

`if` 语句只在条件 (condition) 为真时运行一个代码块 (block)。`else if` 检查另一个条件;`else` 接住剩下的所有情况:

```
const mark = 72;
if (mark >= 80) {
  console.log("A");
} else if (mark >= 60) {
  console.log("B");
} else {
  console.log("C");
}
// prints B
```

- JavaScript 从上往下检查条件, **只运行第一个**为真的分支 (branch)。
- 条件写在圆括号里; 要运行的代码写在花括号 `{ }` 里。

所以顺序很重要。把最严格的条件放在前面:

```
const temp = 35;
if (temp > 30) {
  console.log("hot");
} else if (temp > 20) {
  console.log("warm");   // skipped: the first branch already ran
}
// prints hot
```

4.3 ■ switch 语句

`switch` 把一个值和几个固定的情况 (case) 逐一比较。当每个判断都是" 它是否等于……?" 时, 它比一长串 `else if` 更整齐:

```
const day = "Sat";
switch (day) {
  case "Sat":
  case "Sun":
    console.log("weekend");
    break;
  case "Fri":
    console.log("nearly there");
    break;
  default:
    console.log("school day");
}
// prints weekend
```

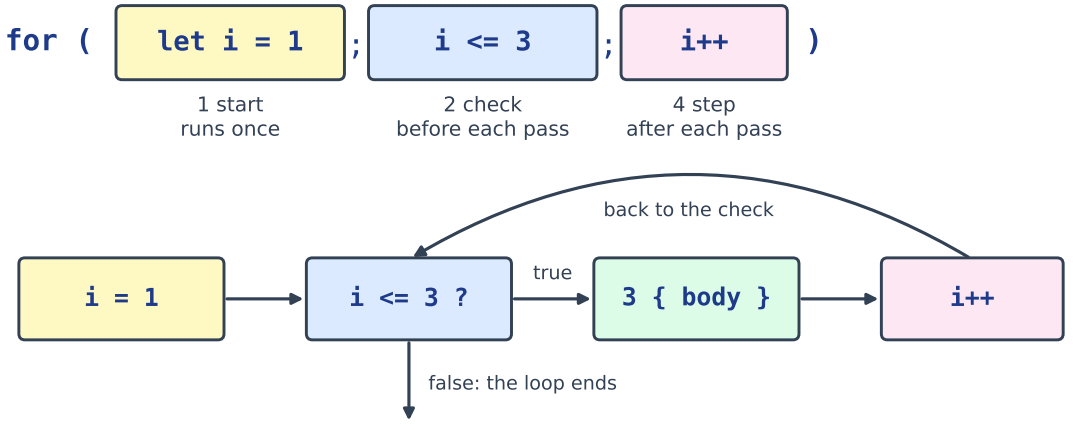
- `switch` 用 `===` 比较, 所以 `case 1:` 匹配不到字符串 `"1"`。
- `break` 结束 `switch`。没有它, 代码会**贯穿** (fall through) 到下一个 case 里继续运行。
- 没有任何 case 匹配时运行 `default`, 就像最后的 `else`。
- 两个 case 之间没有代码时, 它们共用同一段代码, 就像这里的 `"Sat"` 和 `"Sun"`。

常见错误

- `if (x = 5)` 是赋值而不是比较, 这个判断永远为真。要写 `if (x === 5)`。
- `if (age > 13 && < 19)` 是错误的: 要把变量再写一遍,`age > 13 && age < 19`。
- `switch` 里漏掉 `break`, 会把下一个 case 也一起运行。

5 循环

5.1 ■ for loops



循环循环用于重复执行一段代码块。当你确切知道需要重复多少次时，最适合使用 `for` 循环。它的括号内包含三个部分，用分号分隔：

```
for (let i = 1; i <= 3; i++) {
  console.log(i);
}
// 1
// 2
// 3
```

1. **start**: `let i = 1` 在循环开始前仅执行一次。
2. **condition**: `i <= 3` 在每次迭代前都会检查；当其变为假时，循环停止。
3. **step**: `i++`（将 1 加到 `i` 上）在每次迭代后执行。

循环的每一次通过称为一次迭代迭代。可以更改步长以其他方式计数：

```
let line = "";
for (let i = 10; i > 0; i -= 3) {
  line += i + " ";
}
console.log(line);   // 10 7 4 1
```

累加器累加器模式是在循环**之前**声明一个变量，并在每次迭代中更新它：

```
let total = 0;
for (let n = 1; n <= 5; n++) {
  total += n;           // the same as total = total + n
}
console.log(total);     // 15
```

5.2 ■ while loops

`while` 循环在其条件为真时重复执行。适用于无法预先确定需要多少次迭代的场景：

```
let money = 100;
let years = 0;
while (money < 200) {
  money = money * 1.1;    // 10% interest each year
  years++;
}
console.log(years);       // 8
```

- 条件在每次迭代**之前**检查，因此循环体可能一次都不执行。
- 如果条件持续为真且未遇到 `break` 或其他退出机制，将形成无限循环无限循环。（若发生此情况请按下**停止**。）

5.3 ■ for...of, break & continue

`for...of` 依次访问数组中的每一项（或字符串中的每个字符），无需管理索引：

```
const fruits = ["apple", "kiwi", "plum"];
for (const fruit of fruits) {
  console.log(fruit.toUpperCase());
}
// APPLE
// KIWI
// PLUM
```

`break` 立即跳出循环；而 `continue` 则跳过本次迭代的剩余部分并进入下一次迭代：

```
const marks = [55, 80, -1, 90];
let sum = 0;
for (const m of marks) {
  if (m < 0) {
    break;             // -1 means "end of the data"
  }
  if (m < 60) {
    continue;          // skip a fail, keep looping
  }
  sum += m;
}
console.log(sum);      // 80
```

常见错误

- 差一错误：`i < 3` 对 0、1、2 执行操作，而 `i <= 3` 多执行了一次。请仔细计算迭代次数。
- 在循环内部初始化总数：`let total = 0;` 必须在循环**之前**，否则每次迭代都会被重置。
- 如果条件始终为真且未到达退出点，`while` 循环将无限运行。

6 函数

6.1 ■ Parameters & return

函数是一种可重复使用的代码块。函数声明为其命名，您可以使用它来调用该函数：

```
function greet(name) {
  return `Hello, ${name}!`;
}

console.log(greet("Mei"));   // Hello, Mei!
console.log(greet("Tom"));   // Hello, Tom!
```

- name 是参数：一个变量，在函数被调用时接收值。
- "Mei" 是实参：传入的实际值。
- return 向调用者返回值并立即结束函数。

普通函数如果没有返回值而执行到末尾，则返回 undefined。因此请**返回**答案，让调用者决定是否打印它：

```
function area(width, height) {
  return width * height;
}

const a = area(4, 3);
console.log(a + 1);          // 13  (a returned value can be used again)
```

6.2 ■ Arrow functions

箭头函数是编写函数的简写方式。将其存储在 const 中：

```
const square = (n) => n * n;
const add = (a, b) => a + b;

console.log(square(5));      // 25
console.log(add(2, 3));      // 5
```

- 当 => 后面只有一个表达式时，其值会被直接返回，无需书写 return。
- 使用花括号时，需要再次使用 return：

```
const grade = (mark) => {
  if (mark >= 50) {
    return "pass";
  }
  return "fail";
};
console.log(grade(64));      // pass
```

箭头函数通常作为参数传递给其他函数，这将在主题 7 中的 `map` 和 `filter` 中看到。

6.3 ■ Scope

变量的作用域是指能够访问该变量的代码部分。在 `{ }` 中使用 `let` 或 `const` 声明的变量可在该块及其嵌套作用域中访问。即使外层函数调用结束后，嵌套函数仍可保持对该变量的访问权：

```
const level = "global";

function show() {
  const inside = "local";
  console.log(level);        // global  (outer variables are visible)
  console.log(inside);       // local
}

show();
console.log(typeof inside);  // undefined  (inside does not exist out
↪  here)
```

- 在函数内部声明的变量是局部变量；每次调用都会获得一个新的实例。
- 通过参数**传入**值，并通过 `return` **返回**结果，而不是在函数内部修改外部变量。

常见错误

- 忘记 `return`：函数计算出答案后仍会返回 `undefined`。
- 不带括号调用：`greet` 表示函数本身；`greet("Mei")` 用于执行它。
- 在函数内部打印**并**打印其返回值会导致打印两次，或者仅打印 `undefined`。

7 数组

7.1 ■ 创建与修改数组

数组 (array) 按顺序存放一列值, 写在方括号里。每一项是一个元素 (element), 第一个索引是 0:

```
const scores = [72, 88, 95];
console.log(scores[0]);        // 72
console.log(scores.length);    // 3
scores[1] = 90;                // change one element
console.log(scores);           // [72,90,95]
```

- const 不允许给 scores 换一个全新的数组, 但仍然可以修改数组里面的内容。
- 读取超出末尾的位置会得到 undefined:scores[3] 是 undefined。

用 push 和 pop 在末尾添加和删除:

```
const queue = ["Ann", "Ben"];
queue.push("Cai");             // add to the end
console.log(queue);            // ["Ann","Ben","Cai"]
const last = queue.pop();      // remove from the end and return it
console.log(last);             // Cai
console.log(queue.length);     // 2
```

数组里还可以放数组, 这样就构成一个网格, 用两个索引读取, 先写行:

```
const grid = [
  [1, 2, 3],
  [4, 5, 6],
];
console.log(grid[1][2]);       // 6  (row 1, column 2)
```

7.2 ■ map、filter 与 reduce

这几个方法接收一个函数, 并把它作用到每个元素上。它们返回一个**新**数组 (或一个值), 原数组不变:

- map 把每个元素都改一下, 组成一个新数组。
- filter 只保留让函数返回 true 的元素。
- reduce 把所有元素合成一个值, 比如总和。

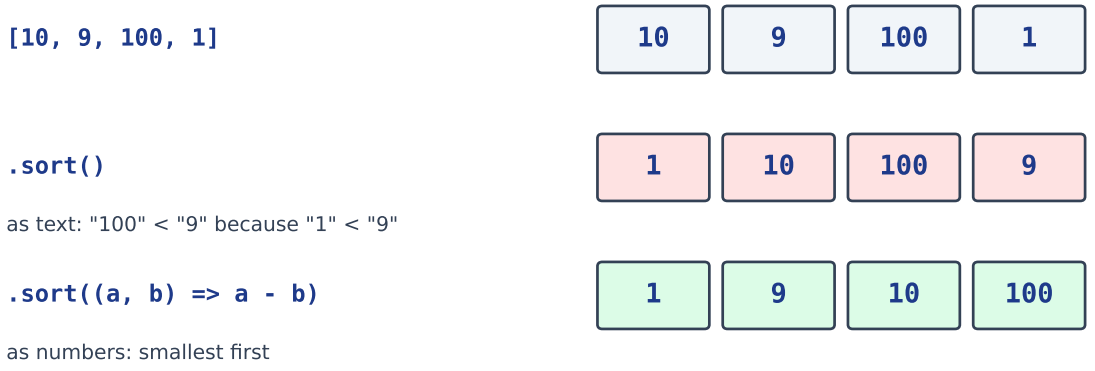
```
const prices = [4, 10, 25];
const withTax = prices.map((p) => p * 1.1);
console.log(withTax.map((p) => p.toFixed(2)));   //
↪  ["4.40","11.00","27.50"]

const cheap = prices.filter((p) => p < 20);
console.log(cheap);                               // [4,10]

const total = prices.reduce((sum, p) => sum + p, 0);
console.log(total);                               // 39
```

在 reduce 里,sum 是累加器。它从最后一个实参 (0) 开始, 每处理一个元素就换成函数的结果。

7.3 ■ 查找与排序



方法	返回	对 [5, 8, 2] 的例子	结果
.includes(x)	true 或 false	.includes(8)	true
.indexOf(x)	x 第一次出现的索引,没有则为 -1	.indexOf(2)	2
.find(f)	第一个让 f 为真的元素	.find((n) => n > 6)	8
.findIndex(f)	这个元素的索引,没有则为 -1	.findIndex((n) => n > 6)	1

`sort` 把元素排好顺序。不给函数时，它把元素**当作文本**排序，所以数字的顺序会出错：

```
const nums = [10, 9, 100, 1];
console.log([...nums].sort());                    // [1,10,100,9]   (as
↪  text!)
console.log([...nums].sort((a, b) => a - b));     // [1,9,10,100]
console.log([...nums].sort((a, b) => b - a));     // [100,10,9,1]
```

- 比较函数在 `a` 应该排在前面时返回负数，在 `b` 应该排在前面时返回正数。
- `sort` 会改变数组本身。`[...nums]` 先复制一份，所以 `nums` 保持原来的顺序。

常见错误

- 最后一个元素是 `arr[arr.length - 1]`，不是 `arr[arr.length]`。
- `map` 和 `filter` 不会改变数组：要把它们的结果存起来。
- 对数字用 `nums.sort()` 会按文本排序。要传入 `(a, b) => a - b`。

8 对象

8.1 ■ Objects & properties

对象对象将相关的值按名称分组在一起。每个名称-值对称为属性属性。该名称被称为键键：

```
const pupil = {
  name: "Mei",
  age: 17,
  form: "11A",
};
console.log(pupil.name);        // Mei
console.log(pupil["form"]);     // 11A  (bracket form, with the key as a
↪  string)
```

- 使用点号（pupil.age）或方括号（pupil["age"]）读取属性。当键名存储在变量中时，必须使用方括号：pupil[field]。
- 访问不存在的键会返回 undefined。

通过赋值来修改现有属性或添加新属性：

```
const pupil = { name: "Mei", age: 17 };
pupil.age = 18;                 // change
pupil.house = "Red";            // add
console.log(pupil);             // {"name":"Mei","age":18,"house":"Red"}
```

8.2 ■ Arrays of objects

记录列表本质上是一个对象数组。可结合主题 7 中的数组方法进行操作：

```
const pupils = [
  { name: "Mei", mark: 88 },
  { name: "Tom", mark: 64 },
  { name: "Sara", mark: 95 },
];

const passed = pupils.filter((p) => p.mark >= 70);
console.log(passed.map((p) => p.name));            // ["Mei","Sara"]

const best = [...pupils].sort((a, b) => b.mark - a.mark)[0];
console.log(`${best.name} has the top mark`);       // Sara has the top
↪  mark
```

对某属性求和，可使用 reduce 或循环遍历：

```
const pupils = [{ mark: 88 }, { mark: 64 }, { mark: 95 }];
let sum = 0;
for (const p of pupils) {
  sum += p.mark;
}
console.log((sum / pupils.length).toFixed(1));      // 82.3
```

8.3 ■ Looping over an object

Object.keys 返回键的数组，而 Object.entries 返回 [key, value] 对，因此可以遍历它们：

```
const stock = { pens: 12, rulers: 0, books: 5 };

console.log(Object.keys(stock));              // ["pens","rulers","books"]

for (const [item, count] of Object.entries(stock)) {
  if (count === 0) {
    console.log(`${item}: sold out`);
  }
}
// rulers: sold out
```

对象是理想的计数器：将每一个被统计的对象作为键使用。

```
const words = ["red", "blue", "red", "red"];
const counts = {};
for (const w of words) {
  counts[w] = (counts[w] || 0) + 1;   // 0 the first time the key is seen
}
console.log(counts);                  // {"red":3,"blue":1}
```

常见错误

- pupil.field 用于查找名为 field 的键。当键名存储在变量中时，应写成 pupil[field]。
- 属性之间用逗号分隔，而非分号：{ name: "Mei", age: 17 }。
- for...of 无法直接遍历对象。请遍历 Object.keys(...) 或 Object.entries(...)。

9 错误

9.1 ■ Reading error messages

当 JavaScript 无法继续执行时，程序会停止并显示错误提示。提示的第一个词表示错误类型，其余部分说明出错原因及位置。阅读该信息后，请定位到所指出的代码行进行检查。

```
SyntaxError: missing ) after argument list
ReferenceError: totl is not defined
TypeError: Cannot read properties of undefined (reading 'length')
```

错误类型	含义	常见原因
`SyntaxError`	代码书写有误，导致程序无法运行	缺少括号或引号
`ReferenceError`	引用了未定义的标识符	拼写错误：例如将 `totl` 误写为 `total`
`TypeError`	值的类型与预期操作不符	在 `.length` 上使用了 `undefined` 操作

不同浏览器的具体报错措辞可能略有差异，但错误名称保持一致。以下程序捕获每个异常，以便打印其名称：

```
try {
  const total = 5;
  console.log(totl);
} catch (e) {
  console.log(e.name);           // ReferenceError
}

try {
  const pupil = undefined;
  console.log(pupil.length);
} catch (e) {
  console.log(e.name);           // TypeError
}
```

某些错误不会触发报错，仅产生错误结果，这称为逻辑错误。排查此类问题可在流程中插入 `console.log` 打印中间值，并人工核对。

9.2 ■ try, catch & throw

可能发生异常的代码应置于 try 块中。若其中发生错误，JavaScript 将跳转至 catch 而非终止程序，且 e 变量将保存该错误对象：

```
try {
  JSON.parse("{not valid}");
  console.log("parsed");         // skipped
} catch (e) {
  console.log("Could not read the data");
}
console.log("The program carries on");
```

可通过 throw 主动抛出错误。这使得函数能够明确拒绝无效输入，而非返回错误结果：

```
function average(list) {
  if (list.length === 0) {
    throw new Error("the list is empty");
  }
  return list.reduce((s, x) => s + x, 0) / list.length;
}

console.log(average([4, 6]));    // 5
try {
  average([]);
} catch (e) {
  console.log(`Error: ${e.message}`);   // Error: the list is empty
}
```

- new Error("…") 会创建一个异常，其 message 属性值为自定义文本。
- 仅在代码可能真正出错（如读取数据）时使用 try。不要用它来掩盖自己代码中的错误。

常见错误

- 跳过该消息。其 **第一**行指出了错误类型及具体原因；请从此处开始排查。
- 不执行任何操作的 catch（即 catch (e) {}）会掩盖问题：至少应打印 e.message。
- throw "text" 可以工作，但 throw new Error("text") 还会记录错误发生的位置。