1. 常量与变量

- **变量**：用于存储可改变的数据。
  - `let`：块级作用域，可重新赋值，但不可重复声明。
  - `var`：函数级作用域，可重复声明，存在变量提升问题（不推荐新代码使用）。
- **常量**：存储不可改变的数据，用 `const` 声明，声明时必须初始化。

```
// let 变量
let age = 18;
age = 20;     // 合法，age 由 18 变为 20

// var 变量（不推荐）
var name = 'Alice';
var name = 'Bob';  // 虽然合法，但容易导致混淆

// const 常量
const PI = 3.14159;
// PI = 3.14;      // 错误！无法修改常量
```

解释

- `let` 和 `const` 都是 ES6 引入的，避免了 `var` 的提升和全局污染问题。
- `const` 保证了变量值不会被意外改写，适合存储不会变的配置或常量。

- 合法字符规则（字母、数字、_、$）
- 不能以数字开头

列一张表：JS 关键字（如 `let`，`function`，`if`）不能作为变量

- 驼峰命名法
- 规范性建议 语义清晰、不要用拼音

✅ JavaScript 常见关键字/保留字对照表

分类	关键字	用途说明
变量声明	`var` `let` `const`	声明变量或常量
函数与类	`function` `return` `class` `constructor` `extends` `super`	定义函数/类/继承等
条件判断	`if` `else` `switch` `case` `default`	条件控制语句
循环控制	`for` `while` `do` `break` `continue`	循环与跳出语句
异步操作	`async` `await`	异步编程控制
逻辑操作	`true` `false` `null` `undefined`	逻辑值 / 空值常量
运算判断	`typeof` `instanceof`	判断数据类型或原型关系
对象操作	`new` `this` `delete` `in`	创建对象、访问/删除属性等
模块系统	`import` `export`	模块导入与导出（ES6）
异常处理	`try` `catch` `finally` `throw`	异常捕获与抛出
作用域控制	`with` `yield`	执行上下文控制（不推荐使用）
未来		

保留字	`enum` `implements` `interface` `package` `private` `protected` `public``static`	

2. 在 HTML 中引入 JavaScript 的方式

1. **内联脚本**
   直接将 JavaScript 写在 `<script>` 标签中。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>内联脚本示例</title>
</head>
<body>
  <script>
    alert('Hello, JavaScript!');
  </script>
</body>
</html>
```

2. **外部脚本**
   将 JS 写在独立文件，通过 `src` 引入。

```
<!-- index.html -->
<script src="main.js"></script>
```

```
// main.js
console.log('外部脚本已加载');
```

3. **事件属性方式**
   在 HTML 元素上直接绑定事件。

```
<button onclick="sayHi()">点击我</button>
<script>
  function sayHi() {
    alert('Hi!');
  }
</script>
```

3. 数据类型

JavaScript 中的数据类型分为 **原始类型** 和 **引用类型**。

类型	示例	说明
Number	`123`, `3.14`	数字（整数、浮点数）
String	`'hello'`, `"JS"`	字符串
Boolean	`true`, `false`	布尔值
undefined	`undefined`	未定义
null	`null`	空值
Symbol	`Symbol('id')`	唯一标识
BigInt	`123n`	大整数
Object	`{}`, `[]`, `function(){}`	引用类型

```
let a;              // undefined
let b = null;      // null
let c = 100n;      // BigInt
let obj = { name: 'Jett' }; // Object
```

4. `typeof` 运算符

用于检测值的数据类型，返回一个字符串。

```
console.log(typeof 123);          // "number"
console.log(typeof 'hello');      // "string"
console.log(typeof true);         // "boolean"
console.log(typeof undefined);    // "undefined"
console.log(typeof null);         // "object"  // 历史遗留问题
console.log(typeof Symbol());     // "symbol"
console.log(typeof 123n);         // "bigint"
console.log(typeof {});           // "object"
console.log(typeof []);           // "object"
console.log(typeof function(){});// "function"
```

> 注意：`typeof null` 返回 `"object"`，这是语言早期设计遗留的坑。

5. 算术运算符

运算符	含义	示例
`+`	加	`1 + 2 // 3`
`-`	减	`5 - 3 // 2`
`*`	乘	`2 * 3 // 6`
`/`	除	`6 / 2 // 3`
`%`	取模（余数）	`5 % 2 // 1`
`**`	幂运算	`2 ** 3 // 8`

```
let x = 10, y = 3;
console.log(x + y);   // 13
console.log(x - y);   // 7
console.log(x * y);   // 30
console.log(x / y);   // 3.333...
console.log(x % y);   // 1
console.log(x ** y);  // 1000
```

6. 赋值运算符

除了最基础的 `=`，还有复合赋值运算符，可在运算后同时赋值。

运算符	含义	等价于
`=`	直接赋值	
`+=`	加后赋值	`a += b` 等价于 `a = a + b`
`-=`	减后赋值	`a -= b` 等价于 `a = a - b`
`*=`	乘后赋值	`a *= b` 等价于 `a = a * b`
`/=`	除后赋值	`a /= b` 等价于 `a = a / b`
`%=`	取模后赋值	`a %= b` 等价于 `a = a % b`
`**=`	幂运算后赋值	`a **= b` 等价于 `a = a ** b`

```
let n = 5;
n += 3;   // n = 8
n *= 2;   // n = 16
n **= 2;  // n = 256
```

7. 比较运算符

运算符	含义	示例
`==`	相等（类型转换后比较）	`'5' == 5` → `true`
`===`	严格相等（不类型转换）	`'5' === 5` → `false`
`!=`	不等（类型转换后比较）	`'5' != 5` → `false`
`!==`	严格不等	`'5' !== 5` → `true`
`>`	大于	`3 > 2` → `true`
`<`	小于	`3 < 2` → `false`
`>=`	大于等于	`3 >= 3` → `true`
`<=`	小于等于	`2 <= 3` → `true`

```
console.log( '5' == 5 );   // true （值相等，类型转换后比较）
console.log( '5' === 5 );  // false（类型不同，不严格相等）
```

8. 布尔运算符

1. 基本概念

1. **逻辑与** `&&`
   - 运算过程中，若遇到「假值」（falsy），立即返回该假值，不再计算后面的表达式。
   - 否则，全部计算完毕后，返回最后一个值。
2. **逻辑或** `||`

- 运算过程中，若遇到「真值」（truthy），立即返回该真值，不再计算后面的表达式。
- 否则，全部计算完毕后，返回最后一个值。

3. **逻辑非 `!`**
   - 先将值转换为布尔类型（`true` / `false`），然后取反。

> 在 JavaScript 中，以下 6 种值被视为 **假值**（falsy），其余均为 **真值**（truthy）：
> `false`、`0`、`""`（空字符串）、`null`、`undefined`、`NaN`。

---

2. 真值表

A	B	A && B	A || B	!A
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

- 注意：上表中的 “true”/“false” 都指 Booleans，但在 JS 中 `&&` / `||` 返回的却是参与运算的**原始值**，并非全都严格返回 `true` 或 `false`。

---

3. 短路特性（Short-circuit Evaluation）

JavaScript 的 `&&` 和 `||` 都具有“短路”行为，会提前返回，不会强制把所有操作数都转换成布尔再计算。

3.1 逻辑与 `&&`

```
console.log(0 && alert('不会执行'));        // 输出 0，alert 永不执行
console.log('foo' && 42 && null && true); // 输出 null，第一次 falsy 时就返回
console.log('foo' && 42 && 'bar');        // 输出 'bar'，所有都是真值，返回最后一个
```

- 第一个例子中，`0` 属于假值，`&&` 立即返回 `0`，后续 `alert` 不会被调用。
- 第二个例子中，依次遇到 `null`（falsy），立刻返回 `null`。

3.2 逻辑或 `||`

```
console.log('' || alert('执行了') || 123); // 弹窗“执行了”，并返回 undefined（alert 返回 undefined）
console.log(false || 0 || '' || 'hello'); // 输出 'hello'，最后一个真值
console.log(false || null || 0);          // 输出 0，最后一个 falsy
```

- 第一个例子中，`''` 是假值，接着执行 `alert`，因为 `alert` 返回 `undefined`（也是假值），再继续 `|| 123`，最终返回 `123`。
- 第二个例子一路假值后，最后遇到 `'hello'`（truthy），立即返回。

3.3 逻辑非 `!`

```js
复制编辑
console.log(!0);        // true，因为 0 → false，再取反 → true
console.log(!'abc');    // false，因为 'abc' → true → 取反 → false
console.log(!!123);     // true，!123 → false，再 !false → true
```

- 常用 `!!` 将任意值强制转换为布尔。

---

9. 类型转换

9.1 隐式转换

JS 会根据上下文自动转换类型：

```
console.log('5' + 3);   // '53'     （字符串拼接）
console.log('5' - 3);   // 2        （数字运算，'5' 隐式转为 5）
console.log([] + {});   // '[object Object]'   （toString 转换）
```

9.2 显式转换

使用全局函数强制转换：

- `String(…)` 或 `.toString()` → 字符串
- `Number(…)` → 数字
- `Boolean(…)` → 布尔

```
console.log( String(123) );     // '123'
console.log( Number('3.14') ); // 3.14
console.log( Boolean(0) );      // false
```