**Es6 let 与 const**

let 声明的变量只在 let 命令所在的代码块内有效

const 声明一个只读的常量，一旦声明，常量的值就不能改变

{let a = 0;

a // 0}

a // 报错 ReferenceError: a is not defined

**代码块内有效**

let 是在代码块内有效，var 是在全局范围内有效

**不能重复声明**

let 只能声明一次 var 可以声明多次

**let 不存在变量提升，var 会变量提升:**

console.log(a); //ReferenceError: a is not defined

let a = "apple";

console.log(b); //undefined

var b = "banana";

注意！！变量 b 用 var 声明存在变量提升，所以当脚本开始运行的时候，b 已经存在了，但是还没有赋值，所以会输出 undefined。

变量 a 用 let 声明不存在变量提升，在声明变量 a 之前，a 不存在，所以会报错

**const 命令**

const 声明一个只读变量，声明之后不允许改变。意味着，一旦声明必须初始化，否则会报错

**模板字符串**

* 普通字符串：

'字符串'

"字符串"

* 模板字符串：

`字符串`

**模板字符串与一般字符串的区别**

* 对于普通用法没有区别

const name1 = 'zjr';

const name2 = `zjr`;

console.log(name1, name2, name1 === name2);

// zjr zjr true

* 字符串拼接的巨大区别

const person = {

name: 'zjr',

age: 18,

sex: '男'

};

const info = '我的名字是：' + person.name + '，性别是：' + person.sex + '，今年：' + person.age + '岁';

console.log(info);

// 我的名字是：zjr，性别是：男，今年：18岁

const person = {

name: `zjr`,

age: 18,

sex: `男`

};

const info = `我的名字是：${person.name}，性别是：${person.sex}，今年：${person.age}岁`;

console.log(info);

// 我的名字是：zjr，性别是：男，今年：18岁

模板字符串最大的优势：方便注入！

当输出多行模板字符串时，所有的空格、换行或缩进都会被保存在输出中！

// 模板字符串

const info = `第一行

第二行`; // 注意不能有缩进

console.log(info);

/\*

第一行

第二行

\*/

模板字符串第二大优势：可以兼容表达式和函数！

模板字符串的 ${} 注入可以兼容几乎所有的值！

模板字符串、字符串、数值、布尔值、表达式、函数……（只要结果是个 “值” 即可）

**箭头函数**

普通函数：

* function 函数名() {}
* const 常量名 = function() {};

箭头函数（也可以叫匿名函数）：

* 参数 => 函数体
* () => {}

注意：由于箭头函数是匿名函数，所以我们通常把它赋给一个变量

当只有单个参数时可以省略括号（）

// 单个参数可以省略 ()

const add = x => {

return x + 1;

};

注意！！箭头函数里没有this

**ES6 解构赋值**

解构赋值是对赋值运算符的扩展。

他是一种针对数组或者对象进行模式匹配，然后对其中的变量进行赋值。

在代码书写上简洁且易读，语义更加清晰明了；也方便了复杂对象中数据字段获取。

**数组模型的解构（Array）**

基本

let [a, b, c] = [1, 2, 3];

// a = 1

// b = 2

// c = 3

可嵌套

let [a, [[b], c]] = [1, [[2], 3]];

// a = 1

// b = 2

// c = 3

可忽略

let [a, , b] = [1, 2, 3];

// a = 1

// b = 3

不完全解构

let [a = 1, b] = []; // a = 1, b = undefined

剩余运算符

let [a, ...b] = [1, 2, 3];

//a = 1

//b = [2, 3]

字符串等

在数组的解构中，解构的目标若为可遍历对象，皆可进行解构赋值。可遍历对象即实现 Iterator 接口的数据。

let [a, b, c, d, e] = 'hello';

// a = 'h'

// b = 'e'

// c = 'l'

// d = 'l'

// e = 'o'

解构默认值

let [a = 2] = [undefined]; // a = 2

当解构模式有匹配结果，且匹配结果是 undefined 时，会触发默认值作为返回结果。

let [a = 3, b = a] = []; // a = 3, b = 3

let [a = 3, b = a] = [1]; // a = 1, b = 1

let [a = 3, b = a] = [1, 2]; // a = 1, b = 2

a 与 b 匹配结果为 undefined ，触发默认值：a = 3; b = a =3

a 正常解构赋值，匹配结果：a = 1，b 匹配结果 undefined ，触发默认值：b = a =1

a 与 b 正常解构赋值，匹配结果：a = 1，b = 2

**对象模型的解构（Object）**

基本

let { foo, bar } = { foo: 'aaa', bar: 'bbb' };

// foo = 'aaa'

// bar = 'bbb'

let { baz : foo } = { baz : 'ddd' };

// foo = 'ddd'

可嵌套可忽略

let obj = {p: ['hello', {y: 'world'}] };

let {p: [x, { y }] } = obj;

// x = 'hello'

// y = 'world'

let obj = {p: ['hello', {y: 'world'}] };

let {p: [x, { }] } = obj;

// x = 'hello'

不完全解构

let obj = {p: [{y: 'world'}] };

let {p: [{ y }, x ] } = obj;

// x = undefined

// y = 'world'

剩余运算符（传递参数范围：0到无限）

let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40};

// a = 10

// b = 20

// rest = {c: 30, d: 40}

解构默认值

let {a = 10, b = 5} = {a: 3};

// a = 3; b = 5;

let {a: aa = 10, b: bb = 5} = {a: 3};

// aa = 3; bb = 5;

**对象字面量**

所谓的对象字面量就是对象的一种新的写法！

之前我们需要用 new 构造函数来生成一个对象，现在我们也可以直接用对象字面量的写法来生成一个对象！

// 实例化构造函数生成对象

const person = new Object();

person.age = 18;

person.speak = function() {};

console.log(person); // { age: 18, speak: [Function (anonymous)] }

// 对象字面量

const person = {

age: 18,

speak: function() {}

};

console.log(person); // { age: 18, speak: [Function (anonymous)] }

**函数默认值**

调用函数的时候传参了，就用传递的参数；如果没传参，就用默认值。

// ES6 默认值实现方式

const multiply = (x, y = 3) => {

return x \* y;

};

console.log(multiply(2, 2)); // 4

console.log(multiply(2)); // 6

注意！函数默认值通常给最后一个变量！

如果默认变量在前面，则传参时要用undefined占位！

const multiply = (x = 3, y) => {

return x \* y;

};

console.log(multiply(2, 2)); // 4

console.log(multiply(undefined，2)); // 6

**剩余参数**

认识剩余参数

const add = (x, y, z, ...args) => {};

//这里的...args就是剩余参数，用来接收x，y，z以外的所有参数

剩余参数的本质

剩余参数永远是个数组，即使没有值，也是个空数组。

const add = (x, y, ...args) => {

console.log(x, y, args);

};

add(); // undefined undefined []

add(1); // 1 undefined []

add(1, 2); // 1 2 []

add(1, 2, 3, 4, 5); // 1 2 [ 3, 4, 5 ]

注意！！箭头函数的参数部分即使只有一个剩余参数，也不能省略圆括号。

const add = (...args) => {};

注意！！剩余参数只能是最后一个参数，之后不能再有其他参数，否则会报错。

**数组展开运算符**

* 剩余参数：参数列表 ——> 参数数组
* 数组展开运算符：参数数组 ——> 参数列表

如果我们需要求 [3, 1, 2] 中的最大值，那么我们是不能直接通过 Math.max() 来求的，因为 Math.max() 接受的不能是数组，而是一个一个的参数，比如：Math.max(3, 1, 2)，所以把参数数组转化为参数列表才是关键。

**展开运算符的基本用法与剩余参数类似：**

console.log(Math.min(...[3, 1, 2]));

相当于

console.log(Math.min(3, 1, 2));

**区分剩余参数和展开运算符**

const add = (...args) => {

console.log(args); // [ 1, 2, 3 ]

console.log(...args); // 1 2 3

};

add(1, 2, 3);

也就是说：剩余参数args是一个数组，在args前面再加...展开运算符才能将这个数组展开成为数组列表，也就是一个一个的数字

**数组展开运算符的应用**

1，复制数组

const a = [1, 2, 3];

const b = [...a];

a[1] = 5;

console.log(a); // [ 1, 5, 3 ]

console.log(b); // [ 1, 2, 3 ]

2，合并数组

const a = [1, 2];

const b = [3];

const c = [4, 5];

console.log([0, ...a, 99, ...c, ...b]);

// [0, 1, 2, 99, 4, 5, 3]

1. 字符串转为数组

字符串可以按照数组的形式展开。

console.log(...'alex'); // a l e x

console.log('a', 'l', 'e', 'x'); // a l e x

console.log([...'alex']); // [ 'a', 'l', 'e', 'x' ]

// ES6 之前字符串转数组是通过：'alex'.split('');

**对象展开运算符**

**可以复制对象**

const a = {x: 1, y: 2};

const c = {...a};

console.log(c, c === a);

// { x: 1, y: 2 } false

注意！！此时如果a是多维的，也就是嵌套对象，...a就是浅拷贝

**Set 和 Map**

**Set 是一系列无序、没有重复值的数据集合**

Set 没有下标去标识每一个值，所以 Set 是无序的，也不能像数组那样通过下标去访问 Set 的成员。

**Set 实例的方法和属性**

1，add 方法（添加值）

const s = new Set();

s.add(0);

// 可以连写

s.add(1).add(2).add(2).add(3); //添加重复值的时候，set会自动去重

console.log(s); // Set(4) { 0, 1, 2, 3 }

2，has 方法（判断值是否存在）

const s = new Set();

s.add(0);

s.add(1).add(2).add(2).add(3);

console.log(s.has(1)); // true

console.log(s.has(4)); // false

3，delete 方法（删除值）

const s = new Set();

s.add(0);

s.add(1).add(2).add(2).add(3);

s.delete(2);

// 使用 delete 删除不存在的成员，什么都不会发生，也不会报错

s.delete(4);

console.log(s); // Set(3) { 0, 1, 3 }

4，clear 方法（清空set）

const s = new Set();

s.add(0);

s.add(1).add(2).add(2).add(3);

s.clear();

console.log(s); // Set(0) {}

const s = new Set();

s.add(0);

s.add(1).add(2).add(2).add(3);

5，forEach 方法（遍历）

forEach 方法可以接受两个参数，第一个是：回调函数，第二个是：指定回调函数的 this 指向。

s.forEach(function (value, key, set) {

// Set 中 value = key，原因：好多数据结构都有 forEach 方法，为了方便统一，所以参数是统一的，但是参数的意义各有不同

// set 就是 s 本身

console.log(value, key, set === s);

console.log(this);

}, document); //这里的document如果不指定，就是window

/\*

0 0 true

#document

1 1 true

#document

2 2 true

#document

3 3 true

#document

\*/

1. size 属性（类似于数组的长度length）

const s = new Set();

s.add(0);

s.add(1).add(2).add(2).add(3);

console.log(s.size); // 4

**Set 如何判断重复**

* Set 对重复值的判断基本遵循严格相等（===），也就是说，两个严格相等的值，无法被添加进一个Set
* 但是对于 NaN 的判断与 === 不同，Set 中 NaN 等于 NaN

**什么时候使用 Set**

* 数组或字符串需要去重时
* 不需要通过下标访问，只需要遍历时
* 为了使用 Set 提供的方法和属性时

**Map是什么**

Map 可以理解为：“映射”。

Map 和 对象 都是键值对的集合。

// 键 ——> 值，key ——> value

// 对象：

const person = {

name: 'alex',

age: 18

};

// Map：

const m = new Map();

m.set('name', 'alex');

m.set('age', 18);

console.log(m); // Map(2) { 'name' => 'alex', 'age' => 18 }

// Map 和 对象 的区别：

// 对象一般用字符串当作 “键”（当然在书写时字符串键的引号可以去掉）.

// Map 中的 “键” 可以是一切类型。函数，未定义等等

const m = new Map();

m.set(true, 'true');

m.set({}, 'object');

m.set(new Set([1, 2]), 'set');

m.set(undefined, 'undefined');

console.log(m);

/\*

Map(4) {

true => 'true',

{} => 'object',

Set(2) { 1, 2 } => 'set',

undefined => 'undefined'

}

\*/

**Map 实例的方法和属性**

1，set 方法（插入映射）

const m = new Map();

m.set('age', 18).set(true, 'true').set('age', 22);

console.log(m); // Map(2) { 'age' => 22, true => 'true' }

2，get 方法（取得映射的“值”）

const m = new Map();

m.set('age', 18).set(true, 'true').set('age', 22);

console.log(m.get('age')); // 22

console.log(m.get(true)); // true

3，has 方法（判断Map里是否存在某个“键”）

const m = new Map();

m.set('age', 18).set(true, 'true').set('age', 22);

console.log(m.has('age')); // true

console.log(m.has('true')); // false

4，delete 方法（删除）

m.delete('age');

// 使用 delete 删除不存在的成员，什么都不会发生，也不会报错

m.delete('name');

5，clear 方法（清空）

m.clear();

6，forEach 方法（遍历，类似于set）

m.forEach(function (value, key, map) {

console.log(this);

}, document);

7，size 属性（类似于长度）

// 对象没有类似的属性，对象没有size或length这种属性！

console.log(m.size);

注意！！

【Map 如何判断键名是否相同】

在 Set 中遇到重复的值直接去掉后者，而 Map 中遇到重复的键值则是后面的覆盖前面的。

* 基本遵循严格相等（===）
* Map 中 NaN 也是等于 NaN

【什么时候使用 Map】

* 如果只是需要键值对结构
* 需要字符串以外的值做键
* 对象一般用在模拟实体上

**Map vs Set 的核心区别：**

| **特性** | **Map** | **Set** |
| --- | --- | --- |
| 数据结构 | 键值对（key-value） | 值的集合（value only） |
| 是否有键 | ✅ 有，任意类型（对象、函数都可以） | ❌ 没有键，只有值 |
| 值是否唯一 | ✅ 键唯一 | ✅ 值唯一 |
| 取值方式 | map.get(key) | 无法按键取值，只能遍历 |
| 设置数据 | map.set(key, value) | set.add(value) |
| 获取长度 | map.size | set.size |
| 应用场景 | 快速查找、映射关系（比如缓存） | 去重、集合运算、唯一性判断 |

**.includes()方法**

数组和字符串都适用

**ES6 Symbol**

ES6 引入了一种新的原始数据类型 Symbol ，表示独一无二的值，最大的用法是用来定义对象的唯一属性名。

ES6 数据类型除了 Number 、 String 、 Boolean 、 Object、 null 和 undefined ，还新增了 Symbol 。

基本用法

Symbol 函数栈不能用 new 命令，因为 Symbol 是原始数据类型，不是对象。可以接受一个字符串作为参数，为新创建的 Symbol 提供描述，用来显示在控制台或者作为字符串的时候使用，便于区分。

let sy = Symbol("kk");

console.log(sy); // Symbol(KK)

typeof(sy); // "symbol"

// 相同参数 Symbol() 返回的值不相等

let sy1 = Symbol("kk");

sy === sy1; // false

**使用场景：作为属性名**

由于每一个 Symbol 的值都是不相等的，所以 Symbol 作为对象的属性名，可以保证属性不重名。

Symbol.for()

Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。

let yellow = Symbol("Yellow");

let yellow1 = Symbol.for("Yellow");

yellow === yellow1; // false

let yellow2 = Symbol.for("Yellow");

yellow1 === yellow2; // true

Symbol.keyFor()

Symbol.keyFor() 返回一个已登记的 Symbol 类型值的 key ，用来检测该字符串参数作为名称的 Symbol 值是否已被登记。

let yellow1 = Symbol.for("Yellow");

Symbol.keyFor(yellow1); // "Yellow"

**原生可遍历与非原生可遍历**

什么是可遍历

只要有 Symbol.iterator 方法，并且这个方法可以生成可遍历对象，就是可遍历的。

只要可遍历，就可以使用 for...of 循环来统一遍历。

**原生可遍历的有哪些？**

数组

字符串

Set

Map

arguments

NodeList

**非原生可遍历的有哪些？**

对象

（手动添加一个 Symbol.iterator 方法，或者是将数组的 Symbol.iterator 方法赋给对象）

**ES6 字符串**

子串的识别

ES6 之前判断字符串是否包含子串，用 indexOf 方法，ES6 新增了子串的识别方法。

includes()：返回布尔值，判断是否找到参数字符串。

startsWith()：返回布尔值，判断参数字符串是否在原字符串的头部。

endsWith()：返回布尔值，判断参数字符串是否在原字符串的尾部。

以上三个方法都可以接受两个参数，需要搜索的字符串，和可选的搜索起始位置索引。

let string = "apple,banana,orange";

string.includes("banana"); // true

string.startsWith("apple"); // true

string.endsWith("apple"); // false

string.startsWith("banana",6) // true

注意点：

这三个方法只返回布尔值，如果需要知道子串的位置，还是得用 indexOf 和 lastIndexOf 。

这三个方法如果传入了正则表达式而不是字符串，会抛出错误。而 indexOf 和 lastIndexOf 这两个方法，它们会将正则表达式转换为字符串并搜索它。

**字符串重复**

repeat()：返回新的字符串，表示将字符串重复指定次数返回。

console.log("Hello,".repeat(2)); // "Hello,Hello,"

如果参数是小数，向下取整

console.log("Hello,".repeat(3.2)); // "Hello,Hello,Hello,"

如果参数是 0 至 -1 之间的小数，会进行取整运算，0 至 -1 之间的小数取整得到 -0 ，等同于 repeat 零次

console.log("Hello,".repeat(-0.5)); // ""

如果参数是 NaN，等同于 repeat 零次

console.log("Hello,".repeat(NaN)); // ""

如果参数是负数或者 Infinity ，会报错:

console.log("Hello,".repeat(-1));

// RangeError: Invalid count value

console.log("Hello,".repeat(Infinity));

// RangeError: Invalid count value

如果传入的参数是字符串，则会先将字符串转化为数字

console.log("Hello,".repeat("hh")); // ""

console.log("Hello,".repeat("2")); // "Hello,Hello,"

字符串补全

padStart：返回新的字符串，表示用参数字符串从头部（左侧）补全原字符串。

padEnd：返回新的字符串，表示用参数字符串从尾部（右侧）补全原字符串。

**ES6异步编程**

众所周知，JavaScript 的脚本是单线程执行的，JS 执行环境中负责执行代码的线程只有一个，即：JavaScript 脚本只在一个线程上运行。也就是说，JavaScript 同时只能执行一个任务，其他任务都必须在后面排队等待。

**一句话总结异步：同步代码 > 微任务（Promise） > 宏任务（setTimeout, setInterval 等）**

console.log('1');

console.log('2');

console.log('3');

// 由于是单线程，所以三个函数的执行必然是遵循 “由前到后，依次执行”，所以执行结果必然是 1 2 3 的顺序

**同步模式**

同步模式指的就是代码会依次执行，后一个任务必须等待前一个任务执行完成之后才可以执行，程序的执行顺序和代码的编写顺序完全一致，这种模式的特点是：简单和安全。

**异步模式**

不同于同步模式，异步模式的 API 不会等待这个任务的结束才开始下一个任务，对于耗时操作都是开启过后就立即执行下一个任务，而针对耗时操作结束后的逻辑一般会通过回调函数的方式定义，当耗时操作完成后会自动调用相应的回调函数来进行后续的处理。

**异步编程一图流**

console.log('1');

setTimeout(() => {

console.log('2');

}, 1800)

setTimeout(() => {

console.log('3');

setTimeout(() => {

console.log('4');

}, 600)

}, 1000)

console.log('5');

**第一步：同步代码先执行**

**第二步：异步开始计时（不会立刻执行）**

**//第一个计时器：设定1800ms后输出2**

**//第二个计时器：一秒后输出3，并设定一个600ms后输出4的定时器**

注意！！虽然第二个计时器嵌套了另一个计时器，但根据倒计时来看，4还是比2先输出，因为600ms小于800ms

| **时间** | **发生的事情** | **输出** |
| --- | --- | --- |
| 0 | 输出 '1'、'5' | ✅ 1, 5 |
| 1000 | 输出 '3'，并设定 600ms 后输出 '4' | ✅ 3 |
| 1600 | 输出 '4' | ✅ 4 |
| 1800 | 输出 '2' | ✅ 2 |

**回调函数**

所有异步编程方案的根基！

1. 我要做一件事
2. 我知道做这件事的步骤
3. 但做这件事之前依赖他人的一个任务，并且我不知道他什么时候能弄完
4. 我还有其他事，我不想干等着他
5. 于是，我干脆把这件事怎么做的步骤提前教给他，让他在把任务完成后帮我随带做了
6. 我开开心心的去做其他事了

**回调函数的用法：**把函数作为参数传递

const love = function () {

console.log('Node.js YYDS!');

}

setTimeout(love, 3000);

// 3s 后，setTimeout 会自动调用（回调）love 函数

// 你也可以直接把回调函数写到形参中

setTimeout(() => {

console.log('Node.js YYDS!');

}, 3000);

**回调地狱：**

用传统回调函数去处理较为复杂的异步问题时，很容易形成深层次的回调函数嵌套问题

setTimeout(() => {

console.log("云南省");

const str01 = "云南省";

setTimeout(() => {

console.log(str01 + "玉溪市");

const str02 = "云南省玉溪市";

setTimeout(() => {

console.log(str02 + "峨山县");

}, 1000, str02);

}, 1000, str01);

}, 1000);

/\*

云南省

云南省玉溪市

云南省玉溪市峨山县

\*/

**什么是 Promise？**

Promise 是一个表示异步操作最终完成（或失败）及其结果值的对象。

它有 3 种状态：

* pending（进行中）
* fulfilled（已成功）
* rejected（已失败）

基本语法：

const p = new Promise((resolve, reject) => {

// 异步操作

let success = true;

if (success) {

resolve("操作成功！");

} else {

reject("操作失败！");

}

});

p.then(result => {

console.log("成功：", result);

}).catch(error => {

console.error("失败：", error);

});

上述代码的解释：

const p = new Promise((resolve, reject) => {

// 这里模拟异步任务，比如 setTimeout、请求接口

// 如果成功，就调用 resolve("操作成功！")

// 如果失败，就调用 reject("操作失败！")

});

处理成功和失败：

p.then(result => {

// 如果成功了，就会进到这里

}).catch(error => {

// 如果失败了，就会进到这里

});

小总结：

* Promise 是用来处理异步任务的。
* 它有 .then() 处理成功、.catch() 处理失败。
* 它能让你的代码更清晰、更容易维护。

**.then的链式调用**

**简单来说：**

每一个 .then() 都会 返回一个新的 Promise对象，这个新的 Promise 会根据你 .then() 里的返回值来决定后续的行为。

经典例子：

new Promise((resolve) => {

resolve(1);

})

.then((res) => {

console.log("第一步：", res); // 输出 1

return res + 1;

})

.then((res) => {

console.log("第二步：", res); // 输出 2

return new Promise((resolve) => {

setTimeout(() => {

resolve(res + 1);

}, 1000);

});

})

.then((res) => {

console.log("第三步：", res); // 输出 3（1秒后）

throw new Error("出错啦！");

})

.then((res) => {

console.log("不会执行到这");

})

.catch((err) => {

console.error("错误处理：", err.message); // 捕获上面 throw 的错误

});

执行流程图（简化版）：

resolve(1)

↓

.then(res => res + 1) // 返回 2

↓

.then(res => Promise) // 返回一个 Promise，等待 1秒后 resolve(3)

↓

.then(res => throw) // 抛出错误

↓

.catch(err => ...) // 处理错误

.then() 链式调用的实质：

每一个 .then()：

* 都是 在等待上一个 Promise 完成之后再执行。
* 会 把上一步 return 的结果传给下一步。
* 可以中间返回 Promise，实现异步串行流程。

**.finally方法**

当 Promise 状态发生变化时，不论如何变化都会执行，不变化不执行。

主要是用来处理一些必做操作，比如在操作数据库之后（无论成功与否）都要关闭数据库连接。

* finally() 不能接收参数
* finally 也是 then 的特例
* finally 也会返回全新的 Promise

**async/await语法糖**

async/await 是以更舒适的方式使用 Promise 的一种特殊语法！

可以避免.then的链式调用

function delay() {

return new Promise(resolve => {

setTimeout(() => {

resolve("1秒后完成！");

}, 1000);

});

}

async function run() {

console.log("开始");

const result = await delay(); // 等待 delay 执行完成

console.log(result); // 打印 "1秒后完成！"

console.log("结束");

}

run();

解释：

* async function 声明一个异步函数
* await 表示 “等待 Promise 执行完成，再继续往下执行”
* await 只能在 async 函数中使用

**异常处理(try...catch):**

async function fetchData() {

try {

const res = await fetch("https://api.example.com/data");

const data = await res.json();

console.log("数据：", data);

} catch (err) {

console.error("出错了：", err);

}

}

**ES6 数值**

常量

Number.EPSILON

Number.EPSILON 属性表示 1 与大于 1 的最小浮点数之间的差。

它的值接近于 2.2204460492503130808472633361816E-16，或者 2-52。

最大/最小安全整数

安全整数

安全整数表示在 JavaScript 中能够精确表示的整数，安全整数的范围在 2 的 -53 次方到 2 的 53 次方之间（不包括两个端点），超过这个范围的整数无法精确表示。

最大安全整数

安全整数范围的上限，即 2 的 53 次方减 1 。

最小安全整数

安全整数范围的下限，即 2 的 53 次方减 1 的负数。

方法

Number 对象新方法

Number.isFinite()

用于检查一个数值是否为有限的（ finite ），即不是 Infinity

console.log( Number.isFinite(1)); // true

console.log( Number.isFinite(0.1)); // true

// NaN 不是有限的

console.log( Number.isFinite(NaN)); // false

console.log( Number.isFinite(Infinity)); // false

console.log( Number.isFinite(-Infinity)); // false

// Number.isFinate 没有隐式的 Number() 类型转换，所有非数值都返回 false

console.log( Number.isFinite('foo')); // false

console.log( Number.isFinite('15')); // false

console.log( Number.isFinite(true)); // false

Number.isNaN()

用于检查一个值是否为 NaN 。

console.log(Number.isNaN(NaN)); // true

console.log(Number.isNaN('true'/0)); // true

// 在全局的 isNaN() 中，以下皆返回 true，因为在判断前会将非数值向数值转换

// 而 Number.isNaN() 不存在隐式的 Number() 类型转换，非 NaN 全部返回 false

Number.isNaN("NaN"); // false

Number.isNaN(undefined); // false

Number.isNaN({}); // false

Number.isNaN("true"); // false

从全局移植到 Number 对象的方法

逐步减少全局方法，用于全局变量的模块化。

方法的行为没有发生改变。

Number.parseInt()

用于将给定字符串转化为指定进制的整数。

// 不指定进制时默认为 10 进制

Number.parseInt('12.34'); // 12

Number.parseInt(12.34); // 12

// 指定进制

Number.parseInt('0011',2); // 3

// 与全局的 parseInt() 函数是同一个函数

Number.parseInt === parseInt; // true

Number.parseFloat()

用于把一个字符串解析成浮点数。

Number.parseFloat('123.45') // 123.45

Number.parseFloat('123.45abc') // 123.45

// 无法被解析成浮点数，则返回 NaN

Number.parseFloat('abc') // NaN

// 与全局的 parseFloat() 方法是同一个方法

Number.parseFloat === parseFloat // true

Number.isInteger()

用于判断给定的参数是否为整数。

Number.isInteger(value)

Number.isInteger(0); // true

// JavaScript 内部，整数和浮点数采用的是同样的储存方法,因此 1 与 1.0 被视为相同的值

Number.isInteger(1); // true

Number.isInteger(1.0); // true

Number.isInteger(1.1); // false

Number.isInteger(Math.PI); // false

// NaN 和正负 Infinity 不是整数

Number.isInteger(NaN); // false

Number.isInteger(Infinity); // false

Number.isInteger(-Infinity); // false

Number.isInteger("10"); // false

Number.isInteger(true); // false

Number.isInteger(false); // false

Number.isInteger([1]); // false

// 数值的精度超过 53 个二进制位时，由于第 54 位及后面的位被丢弃，会产生误判

Number.isInteger(1.0000000000000001) // true

// 一个数值的绝对值小于 Number.MIN\_VALUE（5E-324），即小于 JavaScript 能够分辨

// 的最小值，会被自动转为 0，也会产生误判

Number.isInteger(5E-324); // false

Number.isInteger(5E-325); // true

Number.isSafeInteger()

用于判断数值是否在安全范围内。

Number.isSafeInteger(Number.MIN\_SAFE\_INTEGER - 1); // false

Number.isSafeInteger(Number.MAX\_SAFE\_INTEGER + 1); // false

Math 对象的扩展

Math.cbrt

用于计算一个数的立方根。

Math.cbrt(1); // 1

Math.cbrt(0); // 0

Math.cbrt(-1); // -1

// 会对非数值进行转换

Math.cbrt('1'); // 1

// 非数值且无法转换为数值时返回 NaN

Math.cbrt('hhh'); // NaN

Math.imul

两个数以 32 位带符号整数形式相乘的结果，返回的也是一个 32 位的带符号整数。

// 大多数情况下，结果与 a \* b 相同

Math.imul(1, 2); // 2

Math.hypot

用于计算所有参数的平方和的平方根。

Math.hypot(3, 4); // 5

// 非数值会先被转换为数值后进行计算

Math.hypot(1, 2, '3'); // 3.741657386773941

Math.hypot(true); // 1

Math.hypot(false); // 0

// 空值会被转换为 0

Math.hypot(); // 0

Math.hypot([]); // 0

// 参数为 Infinity 或 -Infinity 返回 Infinity

Math.hypot(Infinity); // Infinity

Math.hypot(-Infinity); // Infinity

// 参数中存在无法转换为数值的参数时返回 NaN

Math.hypot(NaN); // NaN

Math.hypot(3, 4, 'foo'); // NaN

Math.hypot({}); // NaN

Math.trunc

用于返回数字的整数部分。

Math.trunc(12.3); // 12

Math.trunc(12); // 12

// 整数部分为 0 时也会判断符号

Math.trunc(-0.5); // -0

Math.trunc(0.5); // 0

// Math.trunc 会将非数值转为数值再进行处理

Math.trunc("12.3"); // 12

// 空值或无法转化为数值时时返回 NaN

Math.trunc(); // NaN

Math.trunc(NaN); // NaN

Math.trunc("hhh"); // NaN

Math.trunc("123.2hhh"); // NaN

Math.fround

用于获取数字的32位单精度浮点数形式。

// 对于 2 的 24 次方取负至 2 的 24 次方之间的整数（不含两个端点），返回结果与参数本身一致

Math.fround(-(2\*\*24)+1); // -16777215

Math.fround(2 \*\* 24 - 1); // 16777215

// 用于将 64 位双精度浮点数转为 32 位单精度浮点数

Math.fround(1.234) // 1.125

// 当小数的精度超过 24 个二进制位，会丢失精度

Math.fround(0.3); // 0.30000001192092896

// 参数为 NaN 或 Infinity 时返回本身

Math.fround(NaN) // NaN

Math.fround(Infinity) // Infinity

// 参数为其他非数值类型时会将参数进行转换

Math.fround('5'); // 5

Math.fround(true); // 1

Math.fround(null); // 0

Math.fround([]); // 0

Math.fround({}); // NaN

判断

Math.sign

判断数字的符号（正、负、0）。

Math.sign(1); // 1

Math.sign(-1); // -1

// 参数为 0 时，不同符号的返回不同

Math.sign(0); // 0

Math.sign(-0); // -0

// 判断前会对非数值进行转换

Math.sign('1'); // 1

Math.sign('-1'); // -1

// 参数为非数值（无法转换为数值）时返回 NaN

Math.sign(NaN); // NaN

Math.sign('hhh'); // NaN

指数运算符

1 \*\* 2; // 1

// 右结合，从右至左计算

2 \*\* 2 \*\* 3; // 256

// \*\*=

let exam = 2;

exam \*\* = 2; // 4

P

ES6 Reflect 与 Proxy

概述

Proxy 与 Reflect 是 ES6 为了操作对象引入的 API 。

Proxy 可以对目标对象的读取、函数调用等操作进行拦截，然后进行操作处理。它不直接操作对象，而是像代理模式，通过对象的代理对象进行操作，在进行这些操作时，可以添加一些需要的额外操作。

Reflect 可以用于获取目标对象的行为，它与 Object 类似，但是更易读，为操作对象提供了一种更优雅的方式。它的方法与 Proxy 是对应的。

基本用法

Proxy

一个 Proxy 对象由两个部分组成： target 、 handler 。在通过 Proxy 构造函数生成实例对象时，需要提供这两个参数。 target 即目标对象， handler 是一个对象，声明了代理 target 的指定行为。

let target = {

name: 'Tom',

age: 24

}

let handler = {

get: function(target, key) {

console.log('getting '+key);

return target[key]; // 不是target.key

},

set: function(target, key, value) {

console.log('setting '+key);

target[key] = value;

}

}

let proxy = new Proxy(target, handler)

proxy.name // 实际执行 handler.get

proxy.age = 25 // 实际执行 handler.set

// getting name

// setting age

// 25

// target 可以为空对象

let targetEpt = {}

let proxyEpt = new Proxy(targetEpt, handler)

// 调用 get 方法，此时目标对象为空，没有 name 属性

proxyEpt.name // getting name

// 调用 set 方法，向目标对象中添加了 name 属性

proxyEpt.name = 'Tom'

// setting name

// "Tom"

// 再次调用 get ，此时已经存在 name 属性

proxyEpt.name

// getting name

// "Tom"

// 通过构造函数新建实例时其实是对目标对象进行了浅拷贝，因此目标对象与代理对象会互相

// 影响

targetEpt

// {name: "Tom"}

// handler 对象也可以为空，相当于不设置拦截操作，直接访问目标对象

let targetEmpty = {}

let proxyEmpty = new Proxy(targetEmpty,{})

proxyEmpty.name = "Tom"

targetEmpty // {name: "Tom"}

apply(target, ctx, args)

用于拦截函数的调用、call 和 reply 操作。target 表示目标对象，ctx 表示目标对象上下文，args 表示目标对象的参数数组。

function sub(a, b){

return a - b;

}

let handler = {

apply: function(target, ctx, args){

console.log('handle apply');

return Reflect.apply(...arguments);

}

}

let proxy = new Proxy(sub, handler)

proxy(2, 1)

// handle apply

// 1

Reflect

ES6 中将 Object 的一些明显属于语言内部的方法移植到了 Reflect 对象上（当前某些方法会同时存在于 Object 和 Reflect 对象上），未来的新方法会只部署在 Reflect 对象上。

Reflect 对象对某些方法的返回结果进行了修改，使其更合理。

Reflect 对象使用函数的方式实现了 Object 的命令式操作。

P

ES6 Generator 函数

Iterator 遍历器与 for...of 循环