

# JavaScript

---

总结JavaScript相关知识点

## URLs

[XHR和Fetch区别](#)

[Data URL和Blob Url](#)

## JavaScript

[操作符执行顺序](#)

[杂谈](#)

[解构赋值和可迭代协议的关系](#)

[add\["1"\]\["2"\]\["3"\] + 4](#)

[闭包提权漏洞](#)

[void和undefined区别](#)

[不同函数的赋值操作](#)

[为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片？](#)

[实现 \(5\).add\(3\).minus\(2\) 功能](#)

[前端加密方式](#)

[promise状态吸收](#)

[ES5/ES6 的继承除了写法以外还有什么区别](#)

[为什么WeakSet 和 WeakMap 不能包含原始值](#)

[WeakSet 和 WeakMap 有哪些应用](#)

[观察者模式和订阅-发布模式的区别](#)

[实现sleep函数](#)

[如何引用本地npm包](#)

[import A from B和import \\* as A from B有什么区别](#)

[关于作用域的输出](#)

[关于箭头函数的this指向](#)

[在类型判断中为什么用Object.prototype.toString而不是toString](#)

[宏任务、微任务](#)

[for in 和 for of区别](#)

[typeof、instanceof、isPrototypeOf、In的区别](#)

[new一个对象的过程中发生了什么](#)

[JavaScript代码执行过程](#)

[关于let和var的for循环问题](#)

[原型链继承](#)

[垃圾回收机制（GC）](#)

[作用域](#)

## ES6

[let和const](#)

[变量的解构赋值](#)

[字符串的扩展](#)

[字符串的新增方法](#)

[正则的扩展](#)

[数值的扩展](#)

[函数的扩展](#)

[数组的扩展](#)

[对象的扩展](#)

[对象新增方法](#)

[运算符扩展](#)

[Symbol](#)

[Set 和 Map 数据结构](#)

[Proxy](#)

[Reflect](#)

[Promise](#)

[Iterator 和 for...of 循环](#)

[Class](#)

## 代码规范

[React中，不要使用renderXXX,要使用函数式组件](#)

[少用undefined](#)

# XHR和Fetch区别

功能点	XHR	Fetch
基本的请求能力	✓	✓
基本的获取响应能力	✓	✓
监控请求进度	✓	✗
监控响应进度	✓	✓
Service Worker中是否可用	✗	✓
控制cookie的携带	✗	✓
控制重定向	✗	✓
请求取消	✓	✓
自定义referrer	✗	✓
流	✗	✓
API风格	Event	Promise
活跃度	停止更新	不断更新

# Data URL和Blob Url

## Data URL

Data URL，即前缀为 `data:` 协议的 URL，其允许内容创建者向文档中嵌入小文件。它们之前被称作“data URI”，直到这个名字被 WHATWG 弃用。

备注：现代浏览器将 Data URL 视作唯一的不透明来源，而不是可以用于导航的 URL。

```
▼ HTML |  
1 <!-- 前缀 (data:)、指示数据类型的 MIME 类型、如果非文本则为可选的 base64 标记、数据本身 -->  
2 data:<mediatype>[;base64],<data>
```

`mediatype` 是个 [MIME 类型](#)的字符串，例如 `'image/jpeg'` 表示 JPEG 图像文件。如果被省略，则默认值为 `text/plain; charset=US-ASCII`。

如果数据包含 [RFC 3986 中定义为保留字符](#)的字符或包含空格符、换行符或者其他非打印字符，这些字符必须进行[百分号编码](#)（又名“URL 编码”）。

如果数据是文本类型，你可以直接将文本嵌入（根据文档类型，使用合适的实体字符或转义字符）。否则，你可以指定 `base64` 来嵌入 base64 编码的二进制数据。

## Blob

`Blob` 对象表示一个不可变、原始数据的类文件对象。它的数据可以按文本或二进制的格式进行读取，也可以转换成 [ReadableStream](#) 来用于数据操作。

Blob 表示的不一定是 JavaScript 原生格式的数据。`File` 接口基于 `Blob`，继承了 blob 的功能并将其扩展以支持用户系统上的文件。

## 创建一个表示类型化数组内容的URL

以下代码创建了一个 JavaScript [类型化数组](#)，并创建一个新的包含类型化数组中的数据的 `Blob`。然后调用 `URL.createObjectURL()` 方法，将 blob 转换为一个 [URL](#)。

```
1 const code = `
2   <script>alert(1)</script>
3 `;
4
5 const blob = new Blob([code], { type: 'text/html' });
6 URL.createObjectURL(blob)
```

# 操作符执行顺序

在 JavaScript 中，操作符的执行顺序（也称为操作符优先级）决定了在表达式中多个操作符同时出现时，哪些操作符会先执行，哪些会后执行。连续操作符（例如加法、乘法等）在执行时的顺序，通常遵循两个规则：优先级和结合性。

## 1. 操作符优先级 (Operator Precedence)

操作符优先级决定了多个操作符在同一表达式中被求值的顺序。优先级较高的操作符会先于优先级较低的操作符执行。优先级的顺序可以通过参考一个操作符优先级表来理解。

## 2. 操作符的结合性 (Operator Associativity)

结合性决定了当多个操作符具有相同优先级时，它们的执行顺序。JavaScript 操作符的结合性有两种：

- **从左到右结合性 (Left-to-right)**：大多数操作符（例如加法、减法、乘法、除法等）是从左到右执行的。
- **从右到左结合性 (Right-to-left)**：少数操作符（如赋值操作符、三元运算符、逻辑赋值操作符等）是从右到左执行的。

## 3. 连续操作符的执行顺序示例

让我们通过几个常见的例子来展示操作符优先级和结合性的应用。

### 例 1：运算符优先级

```
1 let result = 3 + 4 * 5;  
2 console.log(result);
```

解释：

- 乘法 (`*`) 的优先级高于加法 (`+`)。
- 所以，先计算 `4 * 5`，结果是 `20`，然后再与 `3` 相加。
- 最终结果是：`3 + 20 = 23`。

## 例 2：左到右结合性

```
1 let result = 3 + 4 - 5;  
2 console.log(result);
```

解释：

- 加法（`+`）和减法（`-`）的优先级相同，并且它们是从左到右结合的。
- 所以，首先执行 `3 + 4 = 7`，然后再执行 `7 - 5 = 2`。

## 例 3：赋值运算符的右到左结合性

```
1 let a = 1;  
2 let b = 2;  
3 let c = 3;  
4  
5 a = b = c;  
6 console.log(a, b, c);
```

解释：

- 赋值操作符（`=`）的结合性是从右到左。
- 所以，首先执行 `b = c`，即 `b = 3`，然后执行 `a = b`，即 `a = 3`。
- 最终结果是：`a = 3, b = 3, c = 3`。

## 例 4：三元运算符的右到左结合性

```
1 let result = true ? false ? 1 : 2 : 3;  
2 console.log(result);
```

解释：

- 三元运算符（`?:`）的结合性是从右到左。
- 首先计算右侧的 `false ? 1 : 2`，结果是 `2`。
- 然后，外层三元运算符计算 `true ? 2 : 3`，结果是 `2`。

## 4. 详细操作符优先级表

以下是一些常见的操作符及其优先级：

操作符	优先级	结合性
-----	-----	-----

( ) (小括号)	1	-
[ ] (数组访问)	1	-
. (对象属性访问)	1	-
++ (前缀递增), -- (前缀递减)	2	从右到左
! (逻辑非), typeof , void , delete	3	从右到左
* , / , % (乘、除、取余)	4	从左到右
+ , - (加法、减法)	5	从左到右
<< , >> , >>> (位移操作)	6	从左到右
== , != , === , !== (比较运算)	7	从左到右
&& (逻辑与)	8	从左到右
'		' (逻辑或)
? : (三元运算符)	10	从右到左
= , += , -= , *= , /= 等	11	从右到左
, (逗号操作符)	12	从左到右

注意：

- 小括号 ( ) 的优先级最高，意味着它会在其他所有操作符之前执行。
- 赋值操作符 (=) 有较低的优先级，但它的结合性是 从右到左，这意味着表达式 a = b = c 会先从右到左进行求值。

## 5. 总结

- **优先级:** 操作符优先级决定了不同操作符之间的执行顺序，优先级高的操作符先执行。
- **结合性:** 当多个操作符具有相同的优先级时，结合性决定了它们的执行顺序。大部分操作符是从 **左到右** 结合的，少数（如赋值操作符）是 **从右到左** 结合的。
- **连续操作符:** 连续的操作符在计算时会遵循优先级和结合性规则。你可以使用括号来明确指定执行顺序，避免潜在的混淆。

希望这些解释和示例能帮助你理解 JavaScript 中操作符的执行顺序！

# 解构赋值和可迭代协议的关系

解构赋值（Destructuring Assignment）和可迭代协议（Iterator Protocol）在 JavaScript 中是两个不同的概念，但它们在不同的数据类型上会产生不同的影响。

## 1. 解构赋值和可迭代协议的关系

解构赋值有两种模式：

- 对象解构（{}）：基于属性名匹配，不依赖 `Symbol.iterator`。
- 数组解构（[]）：基于可迭代协议，依赖 `Symbol.iterator`。

👉 关键点：

- 对象（Object）解构：不需要 `Symbol.iterator`，只要有匹配的属性即可解构。
- 数组（Iterable）解构：必须实现 `Symbol.iterator`，否则解构会报错。

## 2. 不同数据类型的影响

### 2.1 对象解构（Object Destructuring）

对象解构只基于 键名匹配，不会使用 `Symbol.iterator`。

#### 示例

```
1 const obj = { a: 1, b: 2, c: 3 };
2 const { a, b } = obj;
3 console.log(a, b); // 1 2
```

✓ 对象解构不会调用 `Symbol.iterator`，它只是按属性名提取值。

## 2.2 数组解构 (Array Destructuring)

数组解构基于 可迭代协议 (Iterable Protocol) , 它要求对象必须实现 `Symbol.iterator`。

### 示例

```
1 const arr = [1, 2, 3];
2 const [x, y] = arr;
3 console.log(x, y); // 1 2
```

数组解构是基于迭代器 (`Symbol.iterator`) 的。

### 不具有 `Symbol.iterator` 的对象无法解构

```
1 const fakeArray = { 0: "a", 1: "b", length: 2 };
2 const [x, y] = fakeArray; // ✗ TypeError: fakeArray is not iterable
```

原因: `fakeArray` 只是一个普通对象, 没有 `Symbol.iterator`, 不能按数组方式解构。

### 手动添加 `Symbol.iterator`

```
1 const iterableObj = {
2   0: "a",
3   1: "b",
4   length: 2,
5   [Symbol.iterator]: function* () {
6     for (let i = 0; i < this.length; i++) {
7       yield this[i];
8     }
9   }
10 };
11
12 const [x, y] = iterableObj;
13 console.log(x, y); // "a" "b"
```

当对象实现 `Symbol.iterator` 后, 它就可以进行数组解构。

## 2.3 其他可迭代对象

字符串（String）、Map、Set 都是可迭代的，它们实现了 `Symbol.iterator`，因此可以进行数组解构：

### 字符串

```
1 const [a, b] = "hello";
2 console.log(a, b); // h e
```

### Set

```
1 const set = new Set([10, 20, 30]);
2 const [first, second] = set;
3 console.log(first, second); // 10 20
```

### Map

```
1 const map = new Map([["a", 1], ["b", 2]]);
2 const [[key1, value1], [key2, value2]] = map;
3 console.log(key1, value1); // "a" 1
4 console.log(key2, value2); // "b" 2
```

✓ 因为 Set 和 Map 都实现了 `Symbol.iterator`，所以可以解构。

## 2.4 `Symbol.iterator` 影响哪些数据类型？

数据类型	是否可用于数组解构 ( <code>Symbol.iterator</code> )	是否可用于对象解构
Object	✗ 不可迭代	✓ 基于键匹配
Array	✓ 可迭代	✗ 无法按属性匹配
String	✓ 可迭代	✗ 无法按属性匹配
Set	✓ 可迭代	✗ 无法按属性匹配

Map	<input checked="" type="checkbox"/> 可迭代	✗ 无法按属性匹配
Arguments	<input checked="" type="checkbox"/> 可迭代	✗ 无法按属性匹配
NodeList	✗ 不能解构 (在老版本浏览器中)	✗ 无法按属性匹配

## 3. ECMA262 规范

ECMA262 规范明确规定了解构赋值的行为：

- 对象解构 (Object Destructuring) 适用于所有对象，不需要 `Symbol.iterator`。 (ECMA-262 13.3.3.5)
- 数组解构 (Array Destructuring) 适用于可迭代对象，要求 `Symbol.iterator` 方法返回一个迭代器。 (ECMA-262 13.3.3.7)

### 🔗 相关规范

- [Object Destructuring in ECMA-262](#)
- [Array Destructuring in ECMA-262](#)

## 4. 总结

解构类型	依赖 <code>Symbol.iterator</code>	作用机制
对象解构 <code>{}</code>	✗ 不依赖	基于属性名匹配
数组解构 <code>[]</code>	<input checked="" type="checkbox"/> 依赖	依赖 <code>Symbol.iterator</code> 迭代协议

👉 如果一个对象没有 `Symbol.iterator`，它就不能被数组解构。

👉 对象解构只是匹配属性，不涉及 `Symbol.iterator`。

🚀 解构赋值和迭代器是不同的概念，但数组解构依赖迭代器，而对象解构不需要迭代器。

# add["1"]["2"]["3"] + 4

JavaScript |

```
1  function createProxy(values = []) {
2      return new Proxy(
3          {},
4          {
5              get(target, key) {
6                  if (key == Symbol.toPrimitive) {
7                      return () =>
8                          values.reduce((acc, cur) => {
9                              return acc + cur;
10                         }, 0);
11                 }
12                 return createProxy([...values, +key]);
13             },
14         }
15     );
16 }
17
18 const add = createProxy();
19 const proxy = add["1"]["2"]["3"] + 4; // 10
```

# 闭包提权漏洞

JavaScript |

```
1 var o = ()=>{
2   let obj = {
3     a: 1,
4     b: 2
5   }
6   return {
7     get: (key)=>{
8       return obj[key]
9     }
10  }
11 })()
12
13 // 不改变上面代码的情况下，修改obj对象
14 Object.defineProperty(Object.prototype, 'getter',{
15   get(){
16     return this
17   }
18 })
19 // o.get('getter')['test']=123
20 // console.log(o.get('test'))
21 console.log(o.get('getter'))
```

# void和undefined区别

在全局对象中，`undefined`不是关键字，可以被声明，但是不能被赋值，因为是只读的

```
1 'undefined' in window
2 // true
3
4 window.undefined = 10;
5 window.undefined
6 // undefined
```

但是在函数作用域就可以被赋值

```
1 function fn(){
2     let undefined;
3     undefined = 123;
4     console.log(undefined)
5 }
6 fn();
7 // 123
```

所以当你想使用 `undefined` 时，不要直接用 `undefined`，要使用 `void 0`；

# 不同函数的赋值操作

```
1  var a = 10;
2  function a() {
3      a = 20;
4      console.log(Object.getOwnPropertyDescriptors(a));
5      console.log(a);
6  }
7  a();
8
9  var b = 10;
10 var b = function () {
11     b = 20;
12     console.log(Object.getOwnPropertyDescriptors(b));
13     console.log(b);
14 }
15 b();
16
17 var c = 10;
18 (function c() {
19     c = 20;
20     console.log(Object.getOwnPropertyDescriptors(c));
21     console.log(c);
22 })();
23
```

JavaScript

## 函数 a()

```
1  function a() {
2      a = 20;
3      console.log(Object.getOwnPropertyDescriptors(a));
4      console.log(a);
5  }
6  a();
```

JavaScript

- `a()` 是一个命名函数，但是在函数内部，`a = 20;` 这行代码试图在函数作用域内给 `a` 赋值。然而，由于 `a` 是函数的名称，它在这个作用域内是不可配置的，所以赋值不会改变

函数本身，而是会在全局作用域创建一个新的变量 `a`（如果不在严格模式下）。

- `Object.getOwnPropertyDescriptors(a);` 在严格模式下会输出函数 `a` 的属性描述符，而在非严格模式下，由于 `a` 被赋值为数字 20，它将输出数字 20 的属性描述符。
- `console.log(a);` 在严格模式下会抛出错误，因为 `a` 是函数的名称，不能被赋值为其他值。在非严格模式下，会输出数字 20。

## 函数 `b()`

```
1 var b = function () {
2   b = 20;
3   console.log(Object.getOwnPropertyDescriptors(b));
4   console.log(b);
5 };
6 b();
```

JavaScript

- `b()` 是一个匿名函数，赋值给了变量 `b`。在函数内部，`b = 20;` 会覆盖外部作用域中的 `b` 变量，将 `b` 的值设置为数字 20。
- `Object.getOwnPropertyDescriptors(b);` 会输出数字 20 的属性描述符，因为 `b` 已经被赋值为数字 20。
- `console.log(b);` 会输出数字 20。

## 立即执行函数 `(function c() {...})();`

```
1 var c = 10;
2 (function c() {
3   c = 20;
4   console.log(Object.getOwnPropertyDescriptors(c));
5   console.log(c);
6 })();
```

JavaScript

- `(function c() {...})();` 是一个立即执行的函数表达式 (IIFE)，它有一个名称 `c`。在函数内部，`c = 20;` 试图给函数名 `c` 赋值，但是这是不允许的，因为函数名在函数作用域内是不可配置的。如果在严格模式下，这行代码会抛出错误。
- `Object.getOwnPropertyDescriptors(c);` 在严格模式下会输出函数 `c` 的属性描述符，而在非严格模式下，由于 `c` 被赋值（虽然赋值无效），它可能不会抛出错误，但也

不会输出正确的属性描述符。

- `console.log(c);` 在严格模式下会抛出错误，因为不能给函数名赋值。在非严格模式下，它将输出函数 `c` 的引用，因为赋值无效。

现在，让我们预测一下在非严格模式下的输出：

对于 `a()` 和 `(function c() {...})();`，由于它们尝试给函数名赋值，在非严格模式下，这些赋值不会改变函数本身，而是会在全局作用域创建新的变量。然而，`console.log(c);` 仍然会输出函数 `c` 的引用，因为赋值操作是无效的。

对于 `b()`，赋值会成功，并且会覆盖外部作用域中的 `b` 变量。

# 为什么通常在发送数据埋点请求的时候使用的是1x1像素的透明 gif 图片？

---

1. **不会阻塞页面加载**: 1x1像素的图片文件大小非常小，几乎不会对页面加载速度产生影响。
2. **不触发HTTP缓存**: 由于图片很小，一般不会被浏览器缓存，这可以确保每次请求都会发送到服务器，从而能够准确记录用户的每次行为。
3. **跨域请求**: 图片请求不受同源策略的限制，可以跨域发送。这对于在不同域名下跟踪用户行为非常有用。
4. **无需JavaScript执行环境**: 即使用户的浏览器禁用了JavaScript，图片请求仍然可以发送，这保证了数据收集的可靠性。
5. **简单易实现**: 使用图片作为埋点请求的实现方式非常简单，只需要在页面上插入一个 `<img>` 标签即可。
6. **不会影响页面布局**: 1x1像素的透明图片不会在页面上显示，因此不会影响页面的布局和设计。
7. **兼容性好**: 几乎所有的浏览器都支持图片请求，因此这种方式有着非常好的兼容性。
8. 图片请求不占用Ajax请求占额

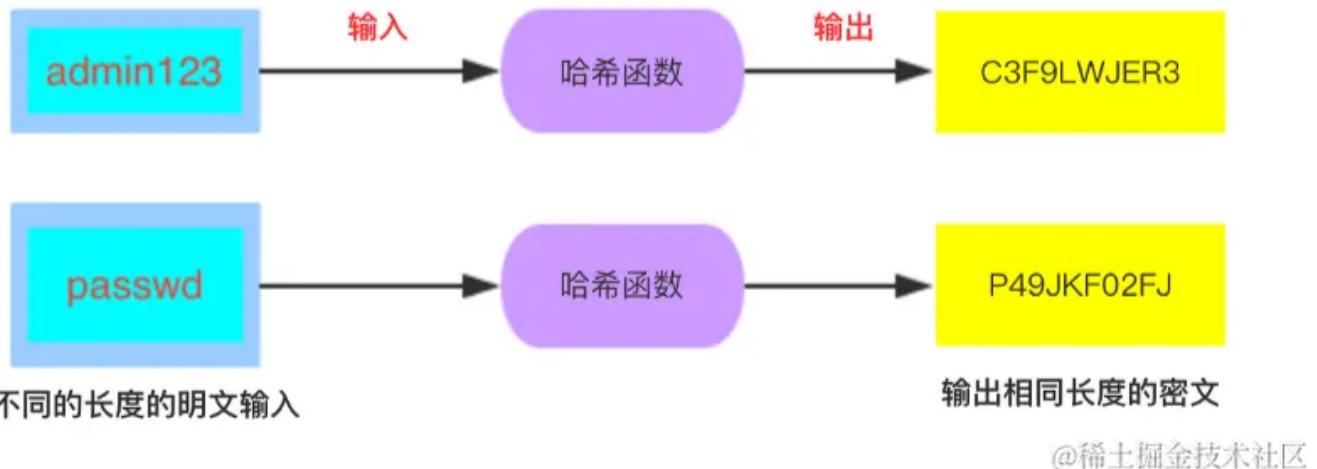
# 实现 (5).add(3).minus(2) 功能

JavaScript |

```
1 Number.prototype.add = function(n){  
2     // this = this+val;  
3     return this.valueOf() + n;  
4 }  
5 Number.prototype.minus = function(n){  
6     // this = this-val;  
7     return this.valueOf() - n;  
8 }  
9 console.log((5).add(3).minus(2))  
10 // 注意要用普通函数，不能用箭头函数
```

# 前端加密方式

## 哈希



- **定义：**哈希也叫散列，是指将任意长度的消息映射为固定长度的输出的算法，该输出一般叫做散列值或者哈希值，也叫做摘要（Digest）。简单来说，这种映射就是一种数据压缩，而且散列是不可逆的，也就是无法通过输出还原输入。
- **特点：**不可逆性（单向性）、抗碰撞性（消息不同其散列值也不同）、长度固定
- **常见应用场景：**由于不可逆性，常用于 **密码存储、数字签名、电子邮件验证、验证下载** 等方面，更多的是用在 **验证数据的完整性** 方面。
  - **密码存储：**明文保存密码是危险的。通常我们把密码哈希加密之后保存，这样即使泄漏了密码，因为是散列后的值，也没有办法推导出密码明文（字典攻击难以破解）。验证的时候，只需要对密码（明文）做同样的散列，对比散列后的输出和保存的密码散列值，就可以验证同一性。
  - **可用于验证下载文件的完整性以及防篡改：**比如网站提供安装包的时候，通常也同时提供 md5 值，这样用户下载之后，可以重算安装包的 md5 值，如果一致，则证明下载到本地的安装包跟网站提供的安装包是一致的，网络传输过程中没有出错。
- **优势：**不可逆，速度快、存储体积小，可以帮助保护数据的完整性和减轻篡改风险。
- **缺点：**安全性不高、容易受到暴力破解（可以加盐处理）

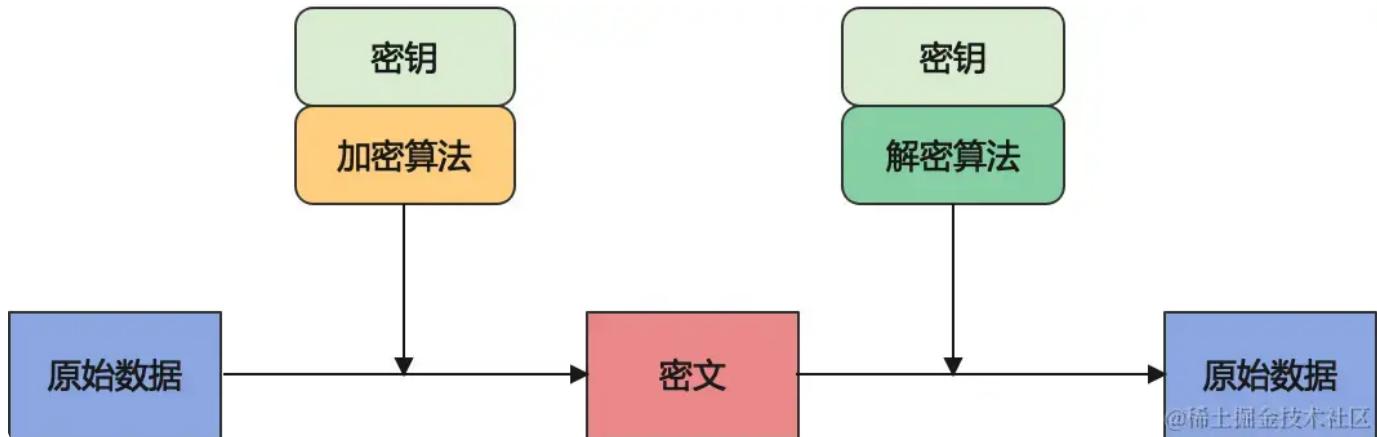
常见类型：SHA-512、SHA-256、MD5（MD5生成的散列码是128位）等。

- **MD5 (Message Digest Algorithm 5)** : 是 RSA 数据安全公司开发的一种单向散列算法，非

可逆，相同的明文产生相同的密文。

- **SHA (Secure Hash Algorithm)** : 可以对任意长度的数据运算生成一个固定位数的数值。
- **SHA/MD5对比**: SHA在安全性方面优于MD5，并且可以选择多种不同的密钥长度。但是，由于内存需求更高，运行速度可能会更慢。不过，MD5因其速度而得到广泛使用，但是由于存在碰撞攻击风险，因此不再推荐使用。

## 对称加密



- **特点**: 优点是速度快，通信效率高；缺点是安全性相对较低。信息传输使用一对一，需要共享相同的密码，密码的安全是保证信息安全的基础，服务器和N个客户端通信，需要维持N个密码记录且不能修改密码。
- **优势**: 效率高，算法简单，系统开销小，速度快，适合大数量级的加解密，安全性中等。
- **缺点**: 密钥管理比较难，密钥存在泄漏风险。
- **常见应用场景**: 适用于需要高速加密/解密的场景，例如 `HTTP` 传输的 `SSL/TLS` 部分，适用于加密大量数据，如 `文件加密、网络通信加密、数据加密、电子邮件、Web 聊天` 等。
  - **文件加密**: 将文件用相同的密钥加密后传输或存储，只有拥有密钥的用户才能解密文件。
  - **数据库加密**: 对数据库中的敏感信息进行加密保护，防止未经授权的人员访问。
  - **通信加密**: 将网络数据通过对称加密算法进行加密，确保数据传输的机密性，比较适合大量短消息的加密和解密。
  - **个人硬盘加密**: 对称加密可以为硬盘加密提供较好的安全性和高处理速度，这对个人电脑而言可能是一个不错的选择。
- **常见类型**: `DES, 3DES, AES` 等：
  - **DES (Data Encryption Standard)** : 分组式加密算法，以64位为分组对数据加密，加解密使用同一个算法，速度较快，适用于加密大量数据的场合。

- 3DES (Triple DES) : 三重数据加密算法，是基于DES，对每个数据块应用三次DES加密算法，强度更高。
- AES (Advanced Encryption Standard) : 高级加密标准算法，速度快，安全级别高，目前已广泛应用于加密大量数据，如文件加密、网络通信加密等。

## AES与DES区别

AES与DES之间的主要区别在于加密过程。在DES中，将明文分为两半，然后再进行进一步处理；而在AES中，整个块不进行除法，整个块一起处理以生成密文。相对而言，AES比DES快得多，与DES相比，AES能够在几秒钟内加密大型文件。

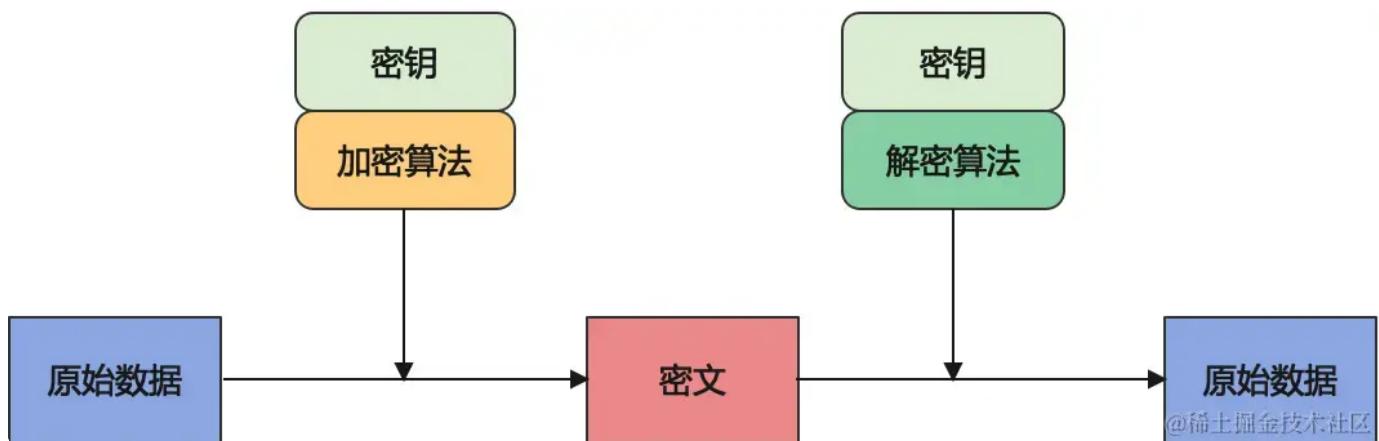
- DES

- 优点：DES算法具有极高安全性，到目前为止，除了用穷举搜索法对DES算法进行攻击外，还没有发现更有效的办法。
- 缺点：分组比较短、密钥太短、密码生命周期短、运算速度较慢。

- AES

- 优点：运算速度快，对内存的需求非常低，适合于受限环境。分组长度和密钥长度设计灵活，AES标准支持可变分组长度；具有很好的抵抗差分密码分析及线性密码分析的能力。
- 缺点：目前尚未存在对AES 算法完整版的成功攻击，但已经提出对其简化算法的攻击。

## 非对称加密



- 特点：缺点是加密解密速度较慢，通信效率较低，优点是安全性高，需要两个不同密钥，信息一对多。因为它使用的是不同的密钥，所以需要耗费更多的计算资源。服务器只需要维持一个

私钥就可以和多个客户端进行通信，但服务器发出的信息能够被所有的客户端解密，且该算法的计算复杂，加密的速度慢。

- **优势：**秘钥容易管理，不存在密钥的交换问题，安全性好，主要用在数字签名，更适用于区块链技术的点对点之间交易的安全性与可信性。
- **缺点：**加解密的计算量大，比对称加密算法计算复杂，性能消耗高，速度慢，适合小数据量或数据签名
- **常见应用场景：**在实际应用中，非对称加密通常用于 **需要确保数据完整性和安全性的场合**，**例如数字证书的颁发、SSL/TLS 协议的加密、数字签名、加密小文件、密钥交换、实现安全的远程通信 等。**

  - **数字签名：**数字签名是为了保证数据的真实性和完整性，通常使用非对称加密实现。发送方使用自己的私钥对数据进行签名，接收方使用发送方的公钥对签名进行验证，如果验证通过，则可以确认数据的来源和完整性。常见的数字签名算法都基于非对称加密，如 RSA、DSA等。
  - **\*\* 身份认证\*\*：**Web浏览器和服务器使用SSL/TLS技术来进行安全通信，其中就使用了非对称加密技术。Web浏览器在与服务器建立连接时，会对服务器进行身份验证并请求其证书。服务器将其证书发送给浏览器，证书包含服务器的公钥。浏览器使用该公钥来加密随机生成的“对话密钥”，然后将其发送回服务器。服务器使用自己的私钥解密此“对话密钥”，以确保双方之间的会话是安全的。
  - **安全电子邮件：**非对称加密可用于电子邮件中，确保邮件内容只能由预期的收件人看到。发件人使用收件人的公钥对邮件进行加密，收件人使用自己的私钥对其进行解密。这确保了只有目标收件人才能读取邮件。

- **常见类型：***RSA, DSA, DSS, ECC* 等
  - **RSA：**由 RSA 公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的。RSA 是一种非对称加密算法，即加密和解密使用一对不同的密钥，分别称为公钥和私钥。公钥用于加密数据，私钥用于解密数据。RSA 算法的安全性基于大数分解问题，密钥长度通常选择 1024 位、2048 位或更长。RSA 算法用于保护数据的机密性、确保数据的完整性和实现数字签名等功能。
  - **DSA (Digital Signature Algorithm) :** 数字签名算法，仅能用于签名，不能用于加解密。
  - **ECC (Elliptic Curves Cryptography) :** 椭圆曲线密码编码学。
  - **DSS:** 数字签名标准，可用于签名，也可以用于加解密。

# promise状态吸收

JavaScript |

```
1 new Promise((resolve, reject) => {
2   resolve(2)
3 new Promise((resolve, reject) => {
4   resolve(5)
5 }).then(res=>console.log(res))
6 }).then(res=>console.log(res))
7
8 // 5
9 // 2
```

JavaScript |

```
1 new Promise( resolve, reject ) => {
2 setTimeout(() => {
3   resolve(2)
4 new Promise( resolve, reject ) => {
5   resolve(5)
6 }).then(res => console.log(res))
7 })
8 }).then(res => console.log(res))
9
10 // 2
11 // 5
```

对于第一段代码,resolve函数执行之后,promise的状态就为已完成,两个promise都是这样,后面执行到then时,直接将回调函数加入微队列执行

对于第二段,当底部的then执行时, promise的状态还是pending,将回调函数加入等待队列,等定时器执行时,先执行resolve(2),此时将等待队列的回调函数加入微队列,然后再执行第二个promise

JavaScript |

```
1 const p1 = new Promise((resolve, reject)=>{
2     console.log(1)
3     resolve()
4 }).then((res)=>{
5     console.log(2)
6     return Promise.resolve()
7 }).then(()=>{
8     console.log(3)
9 })
10
11 const p2 = new Promise((resolve, reject)=>{
12     console.log(4)
13     resolve()
14 }).then((res)=>{
15     console.log(5)
16 }).then(()=>{
17     console.log(6)
18 }).then(()=>{
19     console.log(7)
20 }).then(()=>{
21     console.log(8)
22 })
23
24 // 1 4 2 5 6 7 3 8
```

Plain Text |

```
1 执行栈: 1 4
2 微队列: 2 5
3 当前输出结果: 无
```

Plain Text |

```
1 执行栈: 2 5
2 微队列: 准备 6
3 当前输出结果: 1 4
```

Plain Text |

1 执行栈：准备 6  
2 微队列：吸收 7  
3 当前输出结果：1 4 2 5

Plain Text |

1 执行栈：吸收 7  
2 微队列：3 8  
3 当前输出结果：1 4 2 5 6

Plain Text |

1 执行栈：3 8  
2 微队列：无  
3 当前输出结果：1 4 2 5 6 7

Plain Text |

1 执行栈：无  
2 微队列：无  
3 当前输出结果：1 4 2 5 6 7 3 8

# ES5/ES6 的继承除了写法以外还有什么区别

1. `class` 声明会提升，但不会初始化赋值。`Foo` 进入暂时性死区，类似于 `let`、`const` 声明变量。

```
Plain Text | ▾  
1 const bar = new Bar(); // it's ok  
2 function Bar() {  
3     this.bar = 42;  
4 }  
5  
6 const foo = new Foo(); // ReferenceError: Foo is not defined  
7 class Foo {  
8     constructor() {  
9         this.foo = 42;  
10    }  
11 }
```

2. `class` 声明内部会启用严格模式。

```
Plain Text | ▾  
1 // 引用一个未声明的变量  
2 function Bar() {  
3     baz = 42; // it's ok  
4 }  
5 const bar = new Bar();  
6  
7 class Foo {  
8     constructor() {  
9         fol = 42; // ReferenceError: fol is not defined  
10    }  
11 }  
12 const foo = new Foo();
```

3. `class` 的所有方法（包括静态方法和实例方法）都是不可枚举的。

```
1 // 引用一个未声明的变量
2 function Bar() {
3     this.bar = 42;
4 }
5 Bar.answer = function() {
6     return 42;
7 };
8 Bar.prototype.print = function() {
9     console.log(this.bar);
10};
11 const barKeys = Object.keys(Bar); // ['answer']
12 const barProtoKeys = Object.keys(Bar.prototype); // ['print']
13
14 class Foo {
15     constructor() {
16         this.foo = 42;
17     }
18     static answer() {
19         return 42;
20     }
21     print() {
22         console.log(this.foo);
23     }
24 }
25 const fooKeys = Object.keys(Foo); // []
26 const fooProtoKeys = Object.keys(Foo.prototype); // []
```

4. `class` 的所有方法（包括静态方法和实例方法）都没有原型对象 `prototype`，所以也没有 `[[construct]]`，不能使用 `new` 来调用。

Plain Text |

```
1  function Bar() {
2      this.bar = 42;
3  }
4  Bar.prototype.print = function() {
5      console.log(this.bar);
6  };
7
8  const bar = new Bar();
9  const barPrint = new bar.print(); // it's ok
10
11 class Foo {
12     constructor() {
13         this.foo = 42;
14     }
15     print() {
16         console.log(this.foo);
17     }
18 }
19 const foo = new Foo();
20 const fooPrint = new foo.print(); // TypeError: foo.print is not a constructor
```

5. 必须使用 `new` 调用 `class`。

Plain Text |

```
1  function Bar() {
2      this.bar = 42;
3  }
4  const bar = Bar(); // it's ok
5
6  class Foo {
7      constructor() {
8          this.foo = 42;
9      }
10 }
11 const foo = Foo(); // TypeError: Class constructor Foo cannot be invoked without 'new'
```

6. `class` 内部无法重写类名。

```

1  function Bar() {
2      Bar = 'Baz'; // it's ok
3      this.bar = 42;
4  }
5  const bar = new Bar();
6  // Bar: 'Baz'
7  // bar: Bar {bar: 42}
8
9  class Foo {
10    constructor() {
11        this.foo = 42;
12        Foo = 'Fol'; // TypeError: Assignment to constant variable
13    }
14 }
15 const foo = new Foo();
16 Foo = 'Fol'; // it's ok

```

7. ES5 和 ES6 子类 `this` 生成顺序不同,ES5是先生成子类实例再用父类构造函数修饰子类实例, ES6继承先使用 `super()` 生成父类实例, 然后再子类构造函数中修饰实例

```

1  function MyES5Array() {
2      Array.call(this, arguments);
3  }
4
5  // it's useless
6  const arrayES5 = new MyES5Array(3); // arrayES5: MyES5Array {}
7
8  class MyES6Array extends Array {}
9
10 // it's ok
11 const arrayES6 = new MyES6Array(3); // arrayES6: MyES6Array(3) []

```

# 为什么 WeakSet 和 WeakMap 不能包含原始值

WeakSet 的成员只能是对象和 Symbol 值，而不能是其他类型的值。

WeakMap 只接受对象（`null` 除外）和 `Symbol` 值作为键名，不接受其他类型的值作为键名。`WeakMap` 的键名所指向的对象，不计入垃圾回收机制。

## 弱引用的概念

`WeakSet` 和 `WeakMap` 提供了一种对对象的“弱引用”。这意味着它们不会阻止垃圾回收器回收其所引用的对象。如果一个对象只被 `WeakSet` 或 `WeakMap` 弱引用，并且没有其他强引用指向它，那么垃圾回收器可以自由地回收这个对象。

## 原始值的引用性质

原始值（如字符串、数字和布尔值）在 JavaScript 中是原始数据类型，不是对象。它们是按值传递的，而不是按引用传递。这意味着原始值没有引用的概念，它们存储的是值本身，而不是指向值的引用。

## 设计意图

`WeakSet` 和 `WeakMap` 被设计用来处理对象，特别是那些可能被垃圾回收的对象。以下几个原因说明为什么它们只接受对象作为键（对于 `WeakMap`）或成员（对于 `WeakSet`）：

1. 避免内存泄漏：原始值不是对象，它们在 JavaScript 中是简单的数据值，不会产生垃圾回收的问题。弱引用对于管理对象的生命周期是有用的，但对于原始值则没有必要。
2. 语义一致性：由于 `WeakSet` 和 `WeakMap` 旨在处理对象，允许原始值将引入不一致性。例如，如果 `WeakMap` 允许原始值作为键，那么由于原始值是不可变的，弱引用的语义就不适用了。
3. 简化实现：如果 `WeakSet` 和 `WeakMap` 需要处理原始值，那么它们的内部实现将会更加复杂。例如，需要考虑如何处理原始值的相等性（如 `+0` 和 `-0`，或者两个相同的字符串字面量）。

## 结论

由于原始值不涉及垃圾回收的问题，且它们在 JavaScript 中是按值传递的，因此 `WeakSet` 和 `WeakMap` 不支持它们作为成员或键。这种设计决策有助于保持这些结构的简洁性和专注于它们的主要用例：管理可能被垃圾回收的对象引用。

# WeakSet 和 WeakMap 有哪些应用

## WeakSet 的应用

- 存储 DOM 节点：**当你想跟踪一组 DOM 节点，而又不希望这些节点从 DOM 中移除后仍然保留在内存中时，可以使用 `WeakSet`。
- 实现私有属性：**在类中，可以使用 `WeakSet` 来存储类的实例，从而实现私有属性。当实例不再被引用时，它们会自动从 `WeakSet` 中移除，不会导致内存泄漏。

```
▼ JavaScript |  
1 const privateProps = new WeakSet();  
2 class MyClass {  
3   constructor() {  
4     privateProps.add(this);  
5   }  
6   method() {  
7     if (!privateProps.has(this)) {  
8       throw new TypeError('Cannot call method from outside class instance  
s');  
9     }  
10    // ... method implementation ...  
11  }  
12 }
```

- 缓存：**在某些缓存实现中，可以使用 `WeakSet` 来存储缓存对象，当这些对象不再被使用时，它们会被自动清除，从而避免缓存无限增长。

## WeakMap 的应用

- 关联额外数据：**当你想为一个对象存储一些额外信息，而又不希望这些信息导致对象被保留在内存中时，可以使用 `WeakMap`。
- 事件监听器管理：**在事件管理系统中，可以使用 `WeakMap` 来存储事件监听器，当目标对象被垃圾回收时，相关的事件监听器也会被自动移除。
- 实现非侵入式私有属性：**与 `WeakSet` 类似，`WeakMap` 可以用来存储类的实例与它们的私有属性，这样当实例被销毁时，私有属性也会被自动清理。

```

1  const privateData = new WeakMap();
2  class MyClass {
3    constructor() {
4      privateData.set(this, { secret: 'private value' });
5    }
6    getSecret() {
7      return privateData.get(this).secret;
8    }
9  }

```

4. 元数据存储：在元编程中，`WeakMap` 可以用来存储关于对象的元数据，例如缓存、标记或状态，而不影响对象的生命周期。

```

1  const wm = new WeakMap();
2  const m = new Map();
3
4  var a = {haha:123};
5  wm.set(a, 456);
6  console.log('weakmap', wm.get(a)); // 456
7  m.set(a, 789);
8  console.log('map', m.get(a)); // 789
9  a = null
10 console.log(a)
11 console.log('weakmap', wm.get(a)); // undefined
12 console.log('map', m.get(a)); // undefined

```

# 观察者模式和订阅-发布模式的区别

## 观察者模式 (Observer Pattern)

- 直接关系**: 观察者 (Observer) 直接订阅目标 (Subject) 的事件。目标对象需要维护一个观察者列表，以便在状态变化时通知它们。
- 通知机制**: 目标对象负责通知观察者。当目标对象的状态发生变化时，它会自动调用观察者的更新方法。
- 耦合度**: 目标和观察者之间存在一定的耦合。观察者需要知道目标对象的具体实现，以便能够订阅其事件。
- 实现方式**: 通常，观察者模式通过实现特定的接口或者继承特定的类来实现。

## 发布订阅模式 (Publish–Subscribe Pattern)

- 间接关系**: 发布者 (Publisher) 和订阅者 (Subscriber) 之间不直接通信。它们通过一个中间件 (Event Channel或Event Bus) 进行通信。
- 通知机制**: 发布者发布事件到事件总线，而订阅者从事件总线订阅感兴趣的事件。事件总线负责将事件通知给订阅者。
- 耦合度**: 发布者和订阅者完全解耦。它们不需要知道对方的存在，只需要知道事件总线的接口。
- 实现方式**: 发布订阅模式通常通过一个全局的事件系统来实现，该系统允许动态地添加和删除订阅者。

## 具体区别

- 耦合度**: 观察者模式中，目标和观察者之间紧密耦合；而在发布订阅模式中，发布者和订阅者之间是松耦合的。
- 通知方式**: 在观察者模式中，目标直接调用观察者的方法；而在发布订阅模式中，事件通过一个中间件（事件总线）来传递。
- 动态性**: 发布订阅模式更加灵活，可以在运行时动态地添加或删除订阅者，而观察者模式通常需要在代码中静态地定义观察者和目标的关系。

# 观察者模式

**场景：** 用户界面更新

**例子：** 在一个用户界面（UI）应用中，当用户进行某些操作（如点击按钮）时，需要更新多个不同的UI组件（如标签、图表等）。

```
1 // Subject类: UI组件
2 class UIComponent {
3   constructor() {
4     this.observers = [];
5   }
6
7   addObserver(observer) {
8     this.observers.push(observer);
9   }
10
11  removeObserver(observer) {
12    const index = this.observers.indexOf(observer);
13    if (index > -1) {
14      this.observers.splice(index, 1);
15    }
16  }
17
18  notifyObservers() {
19    this.observers.forEach(observer => observer.update());
20  }
21
22  // 当数据更新时, 通知所有观察者
23  updateData(newData) {
24    // 更新UI组件的数据...
25    this.notifyObservers();
26  }
27 }
28
29 // Observer类: 具体的UI组件, 如标签
30 class UILabel {
31   constructor(name) {
32     this.name = name;
33   }
34
35   update() {
36     console.log(` ${this.name} has been updated.`);
37   }
38 }
39
40 // 使用
41 const dataComponent = new UIComponent();
42 const label1 = new UILabel('Label 1');
43 const label2 = new UILabel('Label 2');
44
45 dataComponent.addObserver(label1);
```

```
46 dataComponent.addObserver(label2);
47
48 // 当用户点击按钮，更新数据
49 dataComponent.updateData({ /* 新数据 */ }); // 输出：
50 // Label 1 has been updated.
51 // Label 2 has been updated.
52
```

## 发布订阅模式

**场景：** 系统模块解耦

**例子：** 在一个大型应用中，订单服务和库存服务是两个不同的模块。当订单服务创建了一个新订单时，它需要通知库存服务减少相应商品的库存，但两者不应该直接相互依赖。

```
1 // EventEmitter类: 事件总线
2 class EventEmitter {
3   constructor() {
4     this.events = {};
5   }
6
7   on(event, listener) {
8     if (!this.events[event]) {
9       this.events[event] = [];
10    }
11    this.events[event].push(listener);
12  }
13
14   emit(event, ...args) {
15     if (!this.events[event]) return;
16     this.events[event].forEach(listener => listener(...args));
17   }
18 }
19
20 // 订单服务
21 class OrderService {
22   constructor(eventEmitter) {
23     this.eventEmitter = eventEmitter;
24   }
25
26   createOrder(orderDetails) {
27     // 创建订单逻辑...
28     console.log('Order created:', orderDetails);
29     // 发布订单创建事件
30     this.eventEmitter.emit('orderCreated', orderDetails);
31   }
32 }
33
34 // 库存服务
35 class InventoryService {
36   constructor(eventEmitter) {
37     this.eventEmitter = eventEmitter;
38     // 订阅订单创建事件
39     this.eventEmitter.on('orderCreated', this.handleOrderCreated.bind(this));
40   }
41
42   handleOrderCreated(orderDetails) {
43     // 更新库存逻辑...
44     console.log('Inventory updated for order:', orderDetails);
```

```
45     }
46   }
47
48 // 使用
49 const eventEmitter = new EventEmitter();
50 const orderService = new OrderService(eventEmitter);
51 const inventoryService = new InventoryService(eventEmitter);
52
53 orderService.createOrder({ productId: 123, quantity: 2 }); // 输出:
54 // Order created: { productId: 123, quantity: 2 }
55 // Inventory updated for order: { productId: 123, quantity: 2 }
56
```

# 实现sleep函数

JavaScript |

```
1 // const sleep = (time, cb) => {
2 //   let now = Date.now();
3 //   console.log(Date.now() - now);
4 //   while (Date.now() - now < time) {}
5 //   cb();
6 // };
7 // sleep(1000, () => console.log(123));
8
9 // const sleep = (cb, timeout) => {
10 //   new Promise((resolve) => {
11 //     resolve	setTimeout(cb, timeout);
12 //   });
13 // };
14 // sleep(() => console.log(123), 1000);
15
16 // const sleep = function*(timeout) {
17 //   yield new Promise((resolve) => {
18 //     setTimeout(resolve, timeout);
19 //   });
20 // };
21 // sleep(1000)
22 //   .next()
23 //   .value.then(() => console.log(123));
24
25 const sleep = async (cb, timeout) => {
26   await setTimeout(cb, timeout);
27 };
28 sleep(() => console.log(123), 1000);
```

# 如何引用本地npm包

## 使用相对路径

```
▼ JavaScript  
1 my-npm-package/  
2   └── node_modules/  
3     └── src/  
4       ├── index.js  
5       └── myLocalModule.js  
6     └── package.json  
7  
8 // 在 index.js 中引用本地模块  
9 const myLocalModule = require('./src/myLocalModule');  
10
```

## 使用npm链接

```
▼ JavaScript  
1 cd path/to/my-npm-package  
2 npm link  
3  
4 cd path/to/other-project  
5 npm link my-npm-package  
6
```

## 使用file:协议

```
1 ▾ {  
2 ▾   "dependencies": {  
3     "my-local-package": "file:../path/to/local/package"  
4   }  
5 }  
6
```

# import A from B和import \* as A from B有什么区别

```
▼ b.js                                         JavaScript |  
1  export const a = 1;  
2  export const b = 2;  
3  export default { c: 3 };  
4
```

```
▼ index.js                                     JavaScript |  
1  import * as a from "./test";  
2  import b from "./test";  
3  console.log(a);  
4  console.log(b);  
5
```

```
▼ Module  {__esModule: true, Symbol(Symbol.toStringTag): 'Module'} ⓘ  
  a: 1  
  b: 2  
  ▶ default: Object  
    __esModule: true  
    Symbol(Symbol.toStringTag): "Module"  
  ▶ get a: () => /* binding */ a  
  ▶ get b: () => /* binding */ b  
  ▶ get default: () => __WEBPACK_DEFAULT_EXPORT__  
  ▶ [[Prototype]]: Object
```

```
▼ a                                              JavaScript |  
1  {  
2    "a": 1,  
3    "b": 2,  
4    "default": {  
5      "c": 3  
6    }  
7 }
```

▼ b

JavaScript |

```
1 ▼ {  
2     "c": 3  
3 }
```

# 关于作用域的输出

JavaScript |

```
1  var a = 10;
2  (function () {
3      console.log(a)
4      a = 5
5      console.log(window.a)
6      var a = 20;
7      console.log(a)
8  })()
9  // undefined
10 // 10
11 // 20
```

- `var a = 20;` 由于 `var` 会有变量提升，声明会提升到函数作用域顶部，所以第一个a是 `undefined`（因为只有声明没有赋值）
- 因为函数作用域已经有声明 `a`，所以 `a = 5` 就是给 `a` 赋值
- 最后根据就近原则获取 `a=20`

JavaScript |

```
1  var a = 10;
2  (function () {
3      console.log(a)
4      a = 5
5      console.log(window.a)
6      console.log(a)
7  })()
8  // 10
9  // 5
10 // 5
```

- 根据就近原则找到外部的a
- `a=5` 没有声明就是全局的
- 就近原则取得 `a=5`

# 关于箭头函数的this指向

JavaScript |

```
1  a = {
2    val: 1,
3    test1: function() {
4      console.log(this)
5    },
6    test2: ()=> {
7      console.log(this)
8    },
9  };
10 b = {
11   c:2
12 }
13 a.test1.call(b)
14 a.test2.call(b)
15 a.test1()
16 a.test2()
```

1. a.test1.call(b): 这里我们使用call方法来调用test1，并将this指向对象b。因此，当test1内部的console.log(this)执行时，它将输出对象b。控制台将显示{ c: 2 }。
2. a.test2.call(b): 尽管我们尝试使用call方法来改变test2的this指向对象b，但是由于箭头函数不绑定自己的this，它会继承自父执行上下文的this。在这个例子中，test2的父执行上下文是全局执行上下文，所以this指向全局对象（在浏览器中通常是window对象）。控制台将显示全局对象。
3. a.test1(): 这里我们直接调用test1方法，没有使用call。因此，this将指向调用者，即对象a。控制台将显示{ val: 1, test1: [Function: test1], test2: [Function: test2] }。
4. a.test2(): 同样，由于test2是一个箭头函数，它将继承自父执行上下文的this。在这种情况下，父执行上下文是对象a，但由于箭头函数的特性，this仍然指向全局对象。控制台将显示全局对象。

```

1 { c: 2 }
2 [object Window]
3 { val: 1, test1: [Function: test1], test2: [Function: test2] }
4 [object Window]
5

```

```

1 a = {
2   val: 1,
3   test1: function() {
4     console.log(this)
5   },
6   test2: ()=> {
7     console.log(this)
8   },
9   test3:{ 
10    cc:2,
11    test33:()=>{
12      console.log(this)
13    }
14  }
15 };
16 b = {
17   c:2
18 }
19
20 a.test3.test33()

```

在这个例子中，`a.test3.test33`是一个箭头函数。箭头函数不绑定自己的`this`，它们继承自父作用域（不是执行上下文）的`this`值。因此，当`test33`被调用时，它里面的`this`指向的是定义它的上下文中的`this`。

在对象`a`中，`test3`是一个对象，它包含一个箭头函数`test33`。由于`test33`是作为一个对象属性被定义的，它的父作用域是全局作用域（因为在定义`a`时，我们处于全局执行上下文中）。在全局作用域中，`this`指向全局对象（在浏览器中通常是`window`对象）。

因此，当`a.test3.test33()`被调用时，`test33`中的`this`指向全局对象，而不是`a.test3`。这是因为`test33`是一个箭头函数，它继承了定义它时所在上下文中的`this`值。

所以，当你运行`a.test3.test33()`时，控制台将输出全局对象，这通常是一个包含了全局变量和函数的`window`对象（在浏览器环境中）。如果代码是在Node.js环境中运行，则`this`将指向一个全局

的global对象。

```
1 function foo() {  
2   setTimeout(() => {  
3     console.log('id:', this.id);  
4   }, 100);  
5 }  
6  
7 var id = 21;  
8  
9 foo.call({ id: 42 });
```

JavaScript |

在这个例子中，foo函数通过call方法被调用，并传入了{ id: 42 }作为this的值。foo函数内部有一个setTimeout，它延迟执行一个箭头函数。

箭头函数不绑定自己的this，它们继承自父执行上下文的this。因此，当setTimeout中的箭头函数被调用时，它的this指向的是foo函数的this，而不是全局对象或setTimeout的this。

由于foo函数是通过call方法调用的，并且传入了{ id: 42 }作为this的值，所以foo函数内部的this指向了{ id: 42 }。这意味着箭头函数中的this也将指向{ id: 42 }。

当setTimeout中的箭头函数执行时，它会输出id:跟随着this.id的值，即42。

# 在类型判断中为什么用Object.prototype.toString而不是toString

在JavaScript中，`Object.prototype.toString` 方法是一个非常有用的工具，用于判断对象的类型。这是因为 `Object.prototype.toString` 的行为是可靠的，它返回一个包含对象类型的字符串，这个字符串通常是 `[object Type]` 的形式，其中 `Type` 是对象的内部类型。

直接使用 `toString` 方法可能会产生误导，因为 `toString` 方法可以被任何对象覆盖。例如，数组、函数、日期等内置对象都可能有自己版本的 `toString` 方法，这些方法返回的值通常与 `Object.prototype.toString` 返回的值不同。

来看一个例子：

```
1 const arr = [1, 2, 3];
2 const func = function() {};
3 const date = new Date();
4 console.log(arr.toString()); // "1,2,3"
5 console.log(func.toString()); // "function() {}"
6 console.log(date.toString()); // "Tue Feb 23 2023 12:00:00 GMT-0500 (Eastern Standard Time)"
7 console.log(Object.prototype.toString.call(arr)); // "[object Array]"
8 console.log(Object.prototype.toString.call(func)); // "[object Function]"
9 console.log(Object.prototype.toString.call(date)); // "[object Date]"
```

在这个例子中，直接调用 `toString` 方法返回的是数组、函数和日期对象的默认字符串表示，这些表示与它们的类型无关。而通过 `Object.prototype.toString.call` 方法，我们可以得到它们的实际类型。

因此，如果你想得到一个对象的准确类型，你应该使用 `Object.prototype.toString.call(obj)` 而不是 `obj.toString()`。这样可以确保你得到的是对象的内部类型，而不是它可能被覆盖的 `toString` 方法的返回值。

# 宏任务、微任务

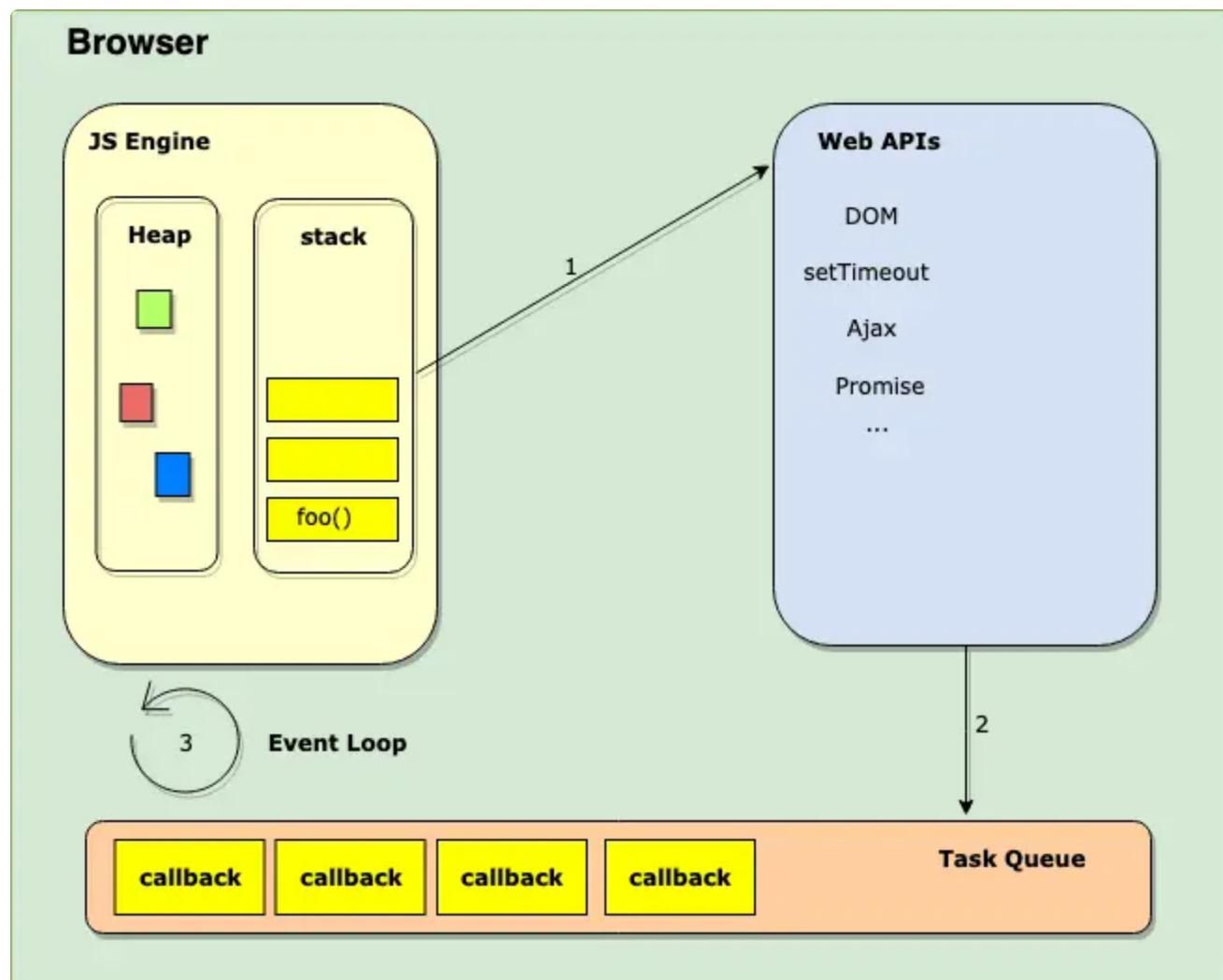
- 是否存在堆栈溢出错误?

JavaScript |

```
1 function foo() {  
2     setTimeout(foo, 0);  
3 }  
4 foo()
```

不会溢出

浏览器的主要组件包括调用堆栈，事件循环，任务队列和Web API。像setTimeout, setInterval和Promise这样的全局函数不是JavaScript的一部分，而是 Web API 的一部分。JavaScript 环境的可视化形式如下所示：



JS调用栈是后进先出(LIFO)的。引擎每次从堆栈中取出一个函数，然后从上到下依次运行代码。每当它遇到一些异步代码，如setTimeout，它就把它交给Web API(箭头1)。因此，每当事件被触发时，callback都会被发送到任务队列(箭头2)。

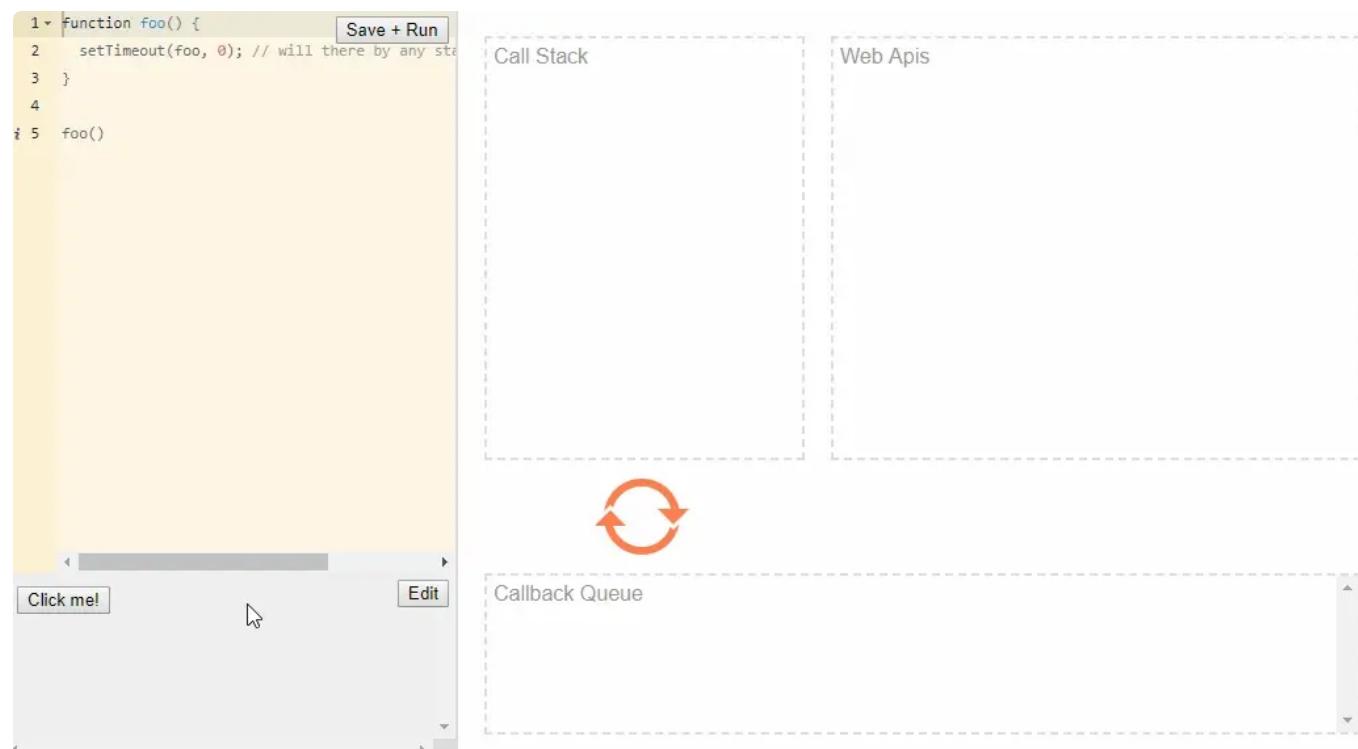
事件循环(Event loop)不断地监视任务队列(Task Queue)，并按它们排队的顺序一次处理一个回调。每当调用堆栈(call stack)为空时，Event loop获取回调并将其放入堆栈(stack)(箭头3)中进行处理。请记住，如果调用堆栈不是空的，则事件循环不会将任何回调推入堆栈。

现在，有了这些知识，让我们来回答前面提到的问题：

## 步骤

1. 调用 foo()会将foo函数放入调用堆栈(call stack)。
2. 在处理内部代码时，JS引擎遇到setTimeout。
3. 然后将foo回调函数传递给Web APIs(箭头1)并从函数返回，调用堆栈再次为空
4. 计时器被设置为0，因此foo将被发送到任务队列<Task Queue>(箭头2)。
5. 由于调用堆栈是空的，事件循环将选择foo回调并将其推入调用堆栈进行处理。
6. 进程再次重复，堆栈不会溢出。

运行示意图如下所示：



▼ 控制台中运行函数，页面(选项卡)的 UI 是否仍然响应

JavaScript |

```
1 function foo() {  
2     return Promise.resolve().then(foo);  
3 };  
4 foo()
```

## 不会响应

大多数时候，开发人员假设在事件循环<event loop>图中只有一个任务队列。但事实并非如此，我们可以有多个任务队列。由浏览器选择其中的一个队列并在该队列中处理回调<callbacks>。

在底层来看，JavaScript中有宏任务和微任务。setTimeout回调是**宏任务**，而Promise回调是**微任务**。

主要的区别在于他们的执行方式。宏任务在单个循环周期中一次一个地推入堆栈，但是微任务队列总是在执行后返回到事件循环之前清空。因此，如果你以处理条目的速度向这个队列添加条目，那么你就永远在处理微任务。只有当微任务队列为空时，事件循环才会重新渲染页面、

所以上述代码，每次调用'foo'都会继续在微任务队列上添加另一个'foo'回调，因此事件循环无法继续处理其他事件（滚动，单击等），直到该队列完全清空为止。因此，它会阻止渲染。

# for in 和 for of区别

- `for...in` 循环遍历对象的键（或数组的索引），而 `for...of` 循环遍历对象的值（或数组的元素）。
- `for...in` 循环可以遍历对象和数组，而 `for...of` 循环主要用于遍历可迭代对象（`Array`, `Map`, `Set`, `String`, `TypedArray`, `arguments` 对象等等）。
- `for...in` 循环会遍历对象的所有可枚举属性，包括原型链上的属性，而 `for...of` 循环不会遍历原型链上的属性。
- `for...of` 循环提供了统一的遍历接口，可以用于遍历多种数据结构，而 `for...in` 循环主要用于遍历对象。
- 不建议使用 `for...in` 遍历数组，因为输出的值还会包含对象属性。
- `for...in` 会跳过空值，`for...of` 对于空值会转为 `undefined`

```
JavaScript | ▾

1 // 遍历对象
2 const person = {
3   firstName: "张",
4   lastName: "三",
5   age: 30
6 };
7
8 for (const key in person) {
9   console.log(key, person[key]); // 输出键和值
10 }
11 console.log('-----')
12
13 // 遍历数组
14 const colors = ["red", undefined, null, "green", "blue"];
15
16 colors.test=123
17 for (const index in colors) {
18   console.log(index, colors[index]); // 输出索引和对应的值
19 }
20 console.log('-----')
21
22 for (const val of colors) {
23   console.log(val); // 输出索引和对应的值
24 }
```

firstName 張

lastName 三

age 30

---

0 red

1 undefined

3 null

4 green

5 blue

test 123

---

red

② undefined

null

green

blue

# typeof、instanceof、isPrototypeOf、In的区别

---

## typeof

typeof是一个运算符，返回一个字符串，表示操作数的类型

注：

- 1、null返回的是“object”，这是由于对象的类型标签是 0。由于 null 代表的是空指针（大多数平台下值为 0x00），因此，null 的类型标签是 0，typeof null 也因此返回 "object"。
- 2、function返回的是“function”

## instanceof

instanceof运算符用于检测构造函数的 `prototype` 属性是否出现在某个实例对象的原型链上

## isPrototypeOf

isPrototypeOf用于检查一个对象是否存在另一个对象的原型链中

```
1 class A{}  
2 class B extends A{}  
3 let b = new B();  
4  
5 // B.__proto__ === A //true  
6 // B.prototype.__proto__=A.prototype //true  
7 // b.__proto__ === B.prototype //true  
8  
9 A.isPrototypeOf(B);//true  
10 A.isPrototypeOf(b);//false  
11 b instanceof B;//true  
12 b instanceof A;//true  
13 B instanceof A;//false  
14  
15 Y instanceof X判断的是X的prototype是否在Y的原型链上,  
16 而我们知道实例的原型链（__proto__）指向的就是其构造函数的prototype,  
17 即Y instanceof X判断Y是否是X的一个实例  
18  
19 X.isPrototypeOf(Y)判断的是X对象是否在Y的原型链上,  
20 同样Y继承X的关系是X对象在Y对象的原型链上,  
21 即X.isPrototypeOf(Y)判断X是否继承至Y
```

## in

in表示如果指定的属性在指定的对象或原型链中，则in运算符返回true

## ▼ in运算符

JavaScript |

```
1 // 数组
2 var trees = new Array("redwood", "bay", "cedar", "oak", "maple");
3 0 in trees; // 返回 true
4 3 in trees; // 返回 true
5 6 in trees; // 返回 false
6 "bay" in trees; // 返回 false (必须使用索引号, 而不是数组元素的值)
7
8 "length" in trees; // 返回 true (length 是一个数组属性)
9
10 Symbol.iterator in trees; // 返回 true (数组可迭代, 只在 ES2015+ 上有效)
11
12 // 内置对象
13 "PI" in Math; // 返回 true
14
15 // 自定义对象
16 var mycar = { make: "Honda", model: "Accord", year: 1998 };
17 "make" in mycar; // 返回 true
18 "model" in mycar; // 返回 true
```

# new一个对象的过程中发生了什么

在JavaScript中，使用 `new` 操作符创建一个对象的过程可以分为以下几个步骤：

1. 创建一个新的空对象：这是通过调用构造函数 `[[Construct]]` 实现的。这个新对象是所有新创建的对象的基础。
2. 将新对象的原型（`[[Prototype]]`）设置为构造函数的 `prototype` 属性：这使得新对象可以继承构造函数的实例属性和方法。
3. 使用新对象调用函数，将新对象作为 `this` 的上下文绑定到构造函数上：这意味着在构造函数内部，`this` 指向新创建的对象。
4. 如果构造函数返回一个非空对象，那么这个对象会作为 `new` 操作符的结果返回：这是通过调用构造函数的 `[[Call]]` 实现的。如果构造函数没有返回对象，或者返回的是 `undefined`，那么 `new` 操作符会返回步骤1中创建的新对象。
5. 如果构造函数中使用了 `new.target`，那么它会被设置为指向构造函数本身：这允许构造函数检查是否真的使用了 `new` 操作符。

现在，让我们通过一个简单的例子来演示这个过程：

```
JavaScript | 1  function MyObject() {  
2      this.myProperty = 'Hello, World!';  
3  }  
4  
5  // 构造函数的prototype属性  
6  MyObject.prototype.myMethod = function() {  
7      console.log('This is a method');  
8  };  
9  
10 // 使用new操作符创建对象实例  
11 const myObject = new MyObject();  
12  
13 // 对象实例继承自构造函数的prototype  
14 console.log(myObject.myMethod()); // 输出: This is a method  
15
```

在这个例子中，`new MyObject()` 会创建一个新的对象，并将这个新对象作为 `this` 的上下文绑定到 `MyObject` 函数上。然后，`this.myProperty = 'Hello, World!'` 会被执行，为

新对象添加一个属性。

`MyObject` 构造函数的 `prototype` 属性被设置为一个对象，该对象包含一个名为 `myMethod` 的方法。当我们使用 `new MyObject()` 创建一个新对象时，新对象的原型被设置为 `MyObject.prototype`。这意味着，当 `myObject` 调用 `myMethod` 时，它实际上是调用其原型（`MyObject.prototype`）上的 `myMethod` 方法。这就是原型链的工作原理，它允许对象继承其他对象的方法和属性。

最后，`new MyObject()` 返回这个新对象，并将其赋值给 `myObject` 变量。

这个过程不仅创建了一个具有正确 `this` 上下文的新对象，还确保了新对象能够继承构造函数的 `prototype` 对象上的属性和方法。

# JavaScript代码执行过程

JavaScript的代码执行过程通常遵循以下步骤：

## 编译阶段

### 词法分析

JS引擎会把代码当成字符串分解成词法单元（token）。[JavaScript代码解析](#)

The screenshot shows a developer tool interface for analyzing JavaScript code. On the left, there is a code editor window containing two lines of code: '1 var a=2' and '2'. To the right of the code editor are three tabs: 'Tree', 'Syntax', and 'Tokens'. The 'Tokens' tab is currently selected, indicated by a blue underline. Below the tabs is a large text area displaying the tokenized output:

```
[{"type": "Keyword", "value": "var"}, {"type": "Identifier", "value": "a"}, {"type": "Punctuator", "value": "="}, {"type": "Numeric", "value": "2"}]
```

### 语法分析

语法分析会把上面说的token转为抽象语法树（AST）

```

1 var a=2
2
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "a"
          },
          "init": {
            "type": "Literal",
            "value": 2,
            "raw": "2"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "script"
}

```

No error.

Index-based node location  
 Line and column-based node location

## 执行阶段

执行程序需要执行环境，在JavaScript中我们称为“**执行上下文**”。

### 执行上下文

- 全局执行上下文：当JS引擎执全局行代码时，会编译全局代码并创建执行上下文
  - i. 创建一个全局对象**GO**（浏览器为window），并将 `this` 的值设为该对象
  - ii. 变量声明提前，将所有变量的声明放在最前面，赋值为 `undefined`，存在变量名相同，只声明一个。
  - iii. 函数声明提前，将函数声明放在最前面。函数名如果跟变量名相同，函数名会覆盖变量名 值是函数体
- 函数执行上下文
  - i. 创建**AO对象**
  - ii. 找出形参和变量声明，值赋予 `undefined`
  - iii. 实参，形参值相统一
  - iv. 在函数体里面找到函数声明，值赋予函数体
- `eval`执行上下文

### 执行栈

一种LIFO（后进先出）栈的数据结构，用来存储代码运行时的所有执行上下文

创建执行上下文

i. 创建阶段

1. 绑定 `this`
2. 词法环境（就是一种标识符—变量的映射，标识符指变量/函数的名字，变量指实际对象/基础对象类型的引用）
  - 环境记录：在创建阶段被称为变量对象（VO），在执行阶段被称为活动对象（AO），之所以被称为变量对象是因为此时该对象只是存储执行上下文中变量和函数声明，之后代码开始执行，变量会逐渐被初始化或是修改，然后这个对象就被称为活动对象
    - 声明类环境记录：存储的是变量和函数声明，函数的词法环境内部就包含一个声明类环境记录
    - 对象环境记录：
  - 外部环境引用：作用域链

```
▼ 词法环境伪代码 JSON |  
1 GlobalExectionContext = { // 全局执行上下文  
2   this: <global object> // this 值绑定  
3   LexicalEnvironment: { // 全局执行上下文词法环境  
4     EnvironmentRecord: { // 环境记录  
5       Type: "Object",  
6       // 标识符在这里绑定  
7     }  
8     outer: <null> // 外部引用  
9   }  
10 }  
11 FunctionExectionContext = { // 函数执行上下文  
12   this: <depends on how function is called> // this 值绑定  
13   LexicalEnvironment: { // 函数执行上下文词法环境  
14     EnvironmentRecord: { // 环境记录  
15       Type: "Declarative",  
16       // 标识符在这里绑定  
17     }  
18     outer: <Global or outer function environment reference> // 引用全局环境  
19   }  
20 }
```

### 3. 变量环境

它同样是一个词法环境，其环境记录器持有变量声明语句在执行上下文中创建的绑定关系。

在 ES6 中，词法环境和变量环境的一个不同就是前者被用来存储**函数声明和变量**（`let` 和 `const`）绑定，而后者只用来存储 `var` 变量绑定。

## ▼ 变量环境伪代码

JavaScript |

```
1  let a = 20;
2  const b = 30;
3  var c;
4
5  function multiply(e, f) {
6      var g = 20;
7      return e * f * g;
8  }
9
10 c = multiply(20, 30);
11
12
13 GlobalExectionContext = {
14
15     ThisBinding: <Global Object>,
16
17     LexicalEnvironment: {
18         EnvironmentRecord: {
19             Type: "Object",
20             // 在这里绑定标识符
21             a: < uninitialized >,
22             b: < uninitialized >,
23             multiply: < func >
24         }
25         outer: <null>
26     },
27
28     VariableEnvironment: {
29         EnvironmentRecord: {
30             Type: "Object",
31             // 在这里绑定标识符
32             c: undefined,
33         }
34         outer: <null>
35     }
36 }
37
38 FunctionExectionContext = {
39     ThisBinding: <Global Object>,
40     LexicalEnvironment: {
41         EnvironmentRecord: {
42             Type: "Declarative",
43             // 在这里绑定标识符
44             Arguments: {0: 20, 1: 30, length: 2},
45         },
46     }
47 }
```

```
46      outer: <GlobalLexicalEnvironment>
47    },
48
49  VariableEnvironment: {
50    EnvironmentRecord: {
51      Type: "Declarative",
52      // 在这里绑定标识符
53      g: undefined
54    },
55    outer: <GlobalLexicalEnvironment>
56  }
57 }
```

可能你已经注意到 `let` 和 `const` 定义的变量并没有关联任何值，但 `var` 定义的变量被设成了 `undefined`。

这是因为在创建阶段时，引擎检查代码找出变量和函数声明，虽然函数声明完全存储在环境中，但是变量最初设置为 `undefined` (`var` 情况下)，或者未初始化 (`let` 和 `const` 情况下)。

这就是为什么你可以在声明之前访问 `var` 定义的变量（虽然是 `undefined`），但是在声明之前访问 `let` 和 `const` 的变量会得到一个引用错误。

这就是我们说的变量声明提升。

## ii. 执行阶段

一旦预编译完成，JavaScript引擎就会开始执行代码。这个过程是逐行执行的，除非遇到特定的语句（如循环、条件判断、函数调用等）改变执行流程。

**事件循环：**JavaScript有一个基于事件循环的异步模型。当遇到异步操作（如 `setTimout`、`AJAX` 请求等）时，这些操作会交给浏览器或其他环境处理，JavaScript引擎继续执行后续代码。当异步操作完成时，它们会被放入事件队列中。事件循环系统会监控事件队列，当主线程空闲时，事件循环会按照先进先出的顺序执行队列中的事件回调。

# 垃圾回收

JavaScript具有自动垃圾回收机制，用来管理内存。垃圾回收器会定期找出不再使用的变量和对象，并释放它们所占用的内存。这个过程是自动的，开发者通常不需要手动干预。在整个执行过程中，JavaScript引擎会不断地在调用栈（Call Stack）和任务队列（Task Queue）之间切换，以处理同步代码和异步事件。调用栈用于跟踪函数的调用和返回，而任务队列

用于存储等待执行的异步回调。

JavaScript代码执行过程示例

```
1 console.log('1. Start');
2
3 // 使用var声明的变量会被提升到当前作用域的顶部
4 console.log('2. ' + typeof variable); // 输出: undefined
5 var variable = 'I am a var';
6
7 // 使用let声明的变量也会被提升，但不会被初始化
8 // console.log('3. ' + typeof letVariable); // 会抛出ReferenceError
9 let letVariable = 'I am a let';
10
11 - function hello() {
12     console.log('Hello, function');
13 }
14
15 - setTimeout(function() {
16     console.log('5. Timeout callback 1 (Macro Task)');
17 -   Promise.resolve().then(function() {
18     console.log('8. Promise callback 1 (Micro Task)');
19   });
20 }, 0);
21
22 - Promise.resolve().then(function() {
23     console.log('6. Promise callback 2 (Micro Task)');
24 });
25
26 hello();
27
28 console.log('4. End');
```

执行步骤解释：

1. 语法分析：浏览器解析代码，确保没有语法错误。

2. 预编译阶段：

- `var variable` 和 `let letVariable` 被提升到它们的作用域顶部，但 `letVariable` 不会被初始化。
- `hello` 函数被提升到全局作用域。

3. 执行阶段：

- `console.log('1. Start');` 执行，输出 `1. Start`。
- `console.log('2. ' + typeof variable);` 执行，由于变量提升，`variable` 存在但未初始化，因此输出 `2. undefined`。
- `var variable = 'I am a var';` 执行，`variable` 被赋值为 `'I am a var'`。
- `let letVariable = 'I am a let';` 执行，`letVariable` 被初始化并赋值为 `'I am a let'`。
- `hello();` 执行，调用 `hello` 函数，输出 `Hello, function`。
- `console.log('4. End');` 执行，输出 `4. End`。

#### 4. 事件循环：

- 同步代码执行完毕后，事件循环检查微任务队列。此时，有一个 `Promise.resolve().then()` 的回调等待执行。
- `Promise callback 2 (Micro Task)` 执行，输出 `6. Promise callback 2 (Micro Task)`。

#### 5. 宏任务：

- 微任务队列清空后，事件循环检查宏任务队列。此时，有一个 `setTimeout` 的回调等待执行。
- `Timeout callback 1 (Macro Task)` 执行，输出 `5. Timeout callback 1 (Macro Task)`。
- 在 `setTimeout` 回调中，有一个 `Promise.resolve().then()` 的回调被添加到微任务队列。

#### 6. 微任务：

- 宏任务执行完毕后，事件循环再次检查微任务队列。此时，有一个新的 `Promise callback 1 (Micro Task)` 等待执行。
- `Promise callback 1 (Micro Task)` 执行，输出 `8. Promise callback 1 (Micro Task)`。

#### 7. 垃圾回收：

- 在代码执行过程中，JavaScript 引擎的垃圾回收器会监控内存使用情况，并在适当的时候释放不再使用的内存。

以上需要注意的是 `let` 声明的变量在预编译阶段会被提升到作用域顶部，但它们不会被初始化，直到它们的声明被执行。

`let` 声明变量分为三部分：1. 创建（当前环境实例化完成），2. 初始化，3. 赋值。

const 声明变量分为两部分：1. 创建，2. 初始化（因为const是个常量，声明时就要赋值，所以初始化和赋值是一起的）。

var 命令声明变量分为两部分：第一部分创建的同时进行初始化，第二部分赋值。

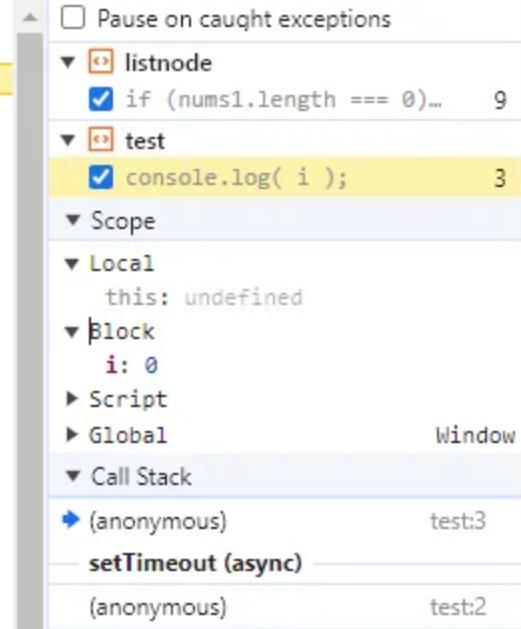
| 深究一下let、const到底有没有提升？ – 掘金

# 关于let和var的for循环问题

```
▼ JavaScript |  
1 for (let i=0; i<10; i++) {  
2   setTimeout(()=>{  
3     console.log( i );  
4   })  
5 }  
6  
7 for (var i=0; i<10; i++) {  
8   setTimeout(()=>{  
9     console.log( i );  
10  })  
11 }  
12 }
```

使用let，因为let每次循环都创造新的作用域，i的值每次都能读到最新的

```
1 for (let i=0; i<10; i++) {  
2   setTimeout(()=>{  
3     console.log( i );  
4   })  
5 }  
6  
7
```



The screenshot shows the browser's developer tools with the scope inspector open. The variable 'i' is highlighted in yellow, indicating it is being inspected. The scope tree shows the following structure:

- listnode (checked): if (nums1.length === 0)... 9
- test (checked): console.log( i ); 3
- Scope
- Local:
  - this: undefined
- Block:
  - i: 0
- Script
- Global Window
- Call Stack
  - (anonymous) test:3
  - setTimeout (async)
  - (anonymous) test:2

使用var，因为var没有块作用域，所以i的值会挂载到全局

```
1  for (var i=0; i<10; i++) {  
2    setTimeout(()=>{  
3      console.log( i );  
4    })  
5  }
```

```
► getComputedStyle: f getComputedStyle  
► getCursorPosition: f (e)  
► getFrameLocation: f getFrameLocation  
► getScreenDetails: f getScreenDetails  
► getSelection: f getSelection()  
► global: Window {window: Window,  
► h: f h(e)  
► handler: f handler(event)  
► hash: f (e,t)  
► hashMatch: null  
► history: History {length: 2, scr  
i: 10  
► icodetest: 12  
► ie: null  
► ie6: null  
► indexedDB: TDBFactory {}
```

# 原型链继承

## 方法继承

JavaScript |

```
1 Teacher.prototype = Object.create(Person.prototype)
2 Teacher.prototype.constructor = Teacher
```

## 属性继承

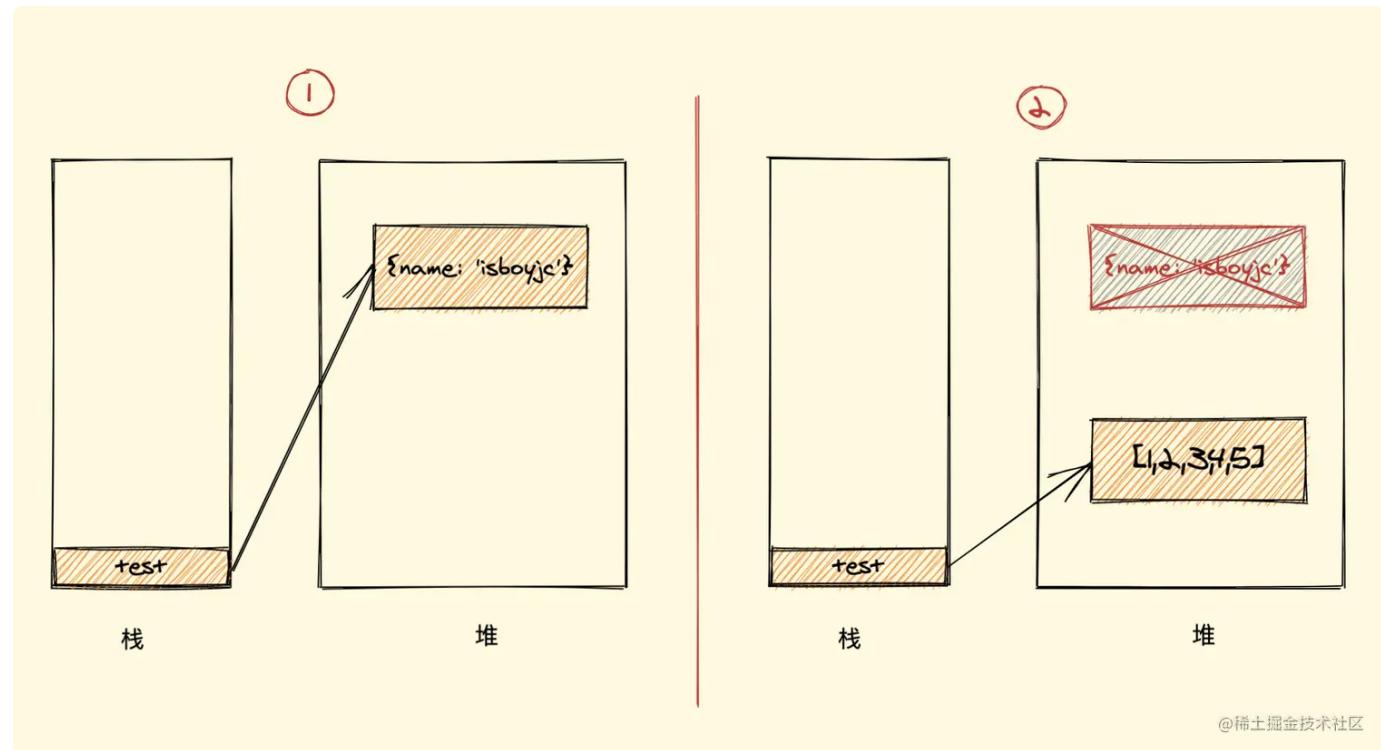
JavaScript |

```
1 function Person (name, age) {
2     this.name = name
3     this.age = age
4 }
5
6 // 方法定义在构造函数的原型上
7 Person.prototype.getName = function () { console.log(this.name)}
8
9 function Teacher (name, age, subject) {
10    Person.call(this, name, age)
11    this.subject = subject
12 }
```

# 垃圾回收机制 (GC)

## 垃圾产生、回收

```
JavaScript | ▾  
1 let test = {  
2   name: "isboyjc"  
3 };  
4 test = [1,2,3,4,5]  
5
```



上述代码中可以看到 `test` 变量先赋值给了一个对象，然后又赋值给了一个数组，那之前的对象就没有引用了，就需要垃圾回收。

## 垃圾回收策略

在 JavaScript 内存管理中有一个概念叫做 **可达性**，就是那些以某种方式可访问或者说可用的值，它们被保证存储在内存中，反之不可访问则需回收。

## 标记清除

引擎在执行 **GC**（使用标记清除算法）时，需要从出发点去遍历内存中所有的对象去打标记，而这个出发点有很多，我们称之为一组 **根对象**，而所谓的**根对象**，其实在浏览器环境中包括又不止于 **全局Window对象、文档DOM树 等**

## 策略

- 垃圾收集器在运行时会给内存中的所有变量都加上一个标记，假设内存中所有对象都是垃圾，全标记为0
- 然后从各个**根对象**开始遍历，把不是垃圾的节点改成1
- 清理所有标记为0的垃圾，销毁并回收它们所占用的内存空间
- 最后，把所有内存中对象标记修改为0，等待下一轮垃圾回收

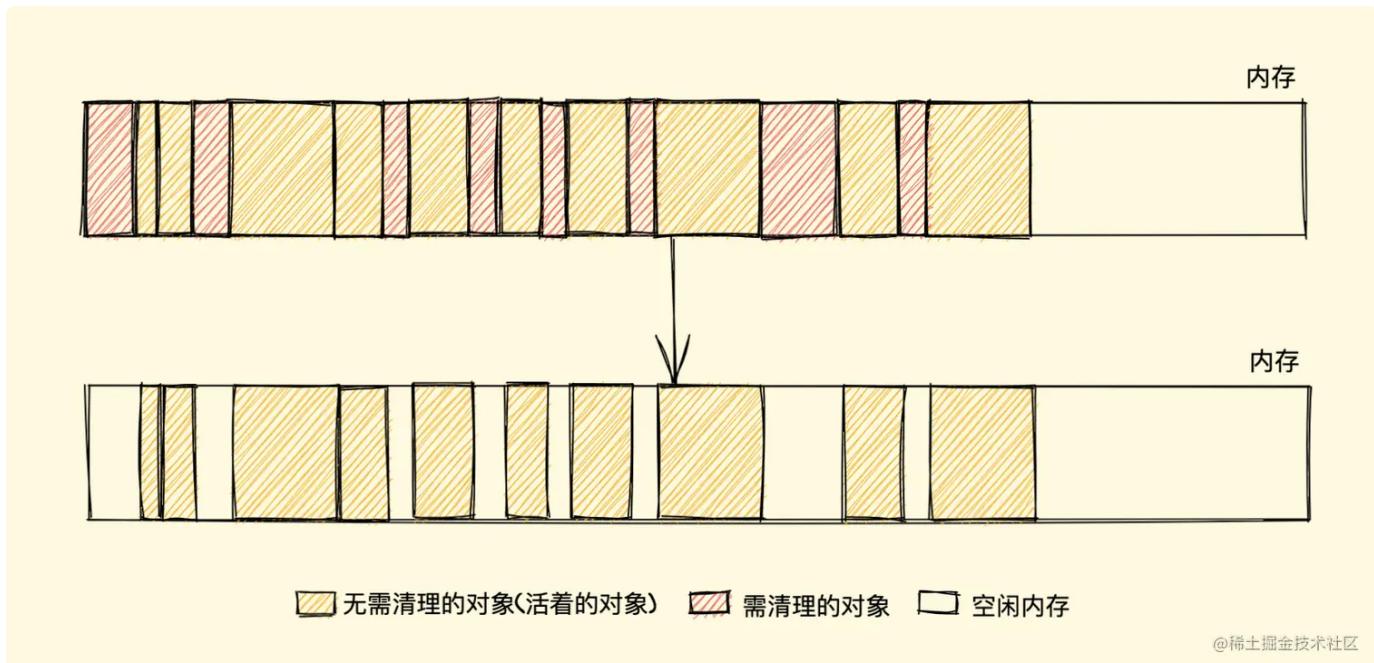
## 优点

标记清除算法的优点只有一个，那就是实现比较简单，打标记也无非打与不打两种情况，这使得一位二进制位（0和1）就可以为其标记，非常简单

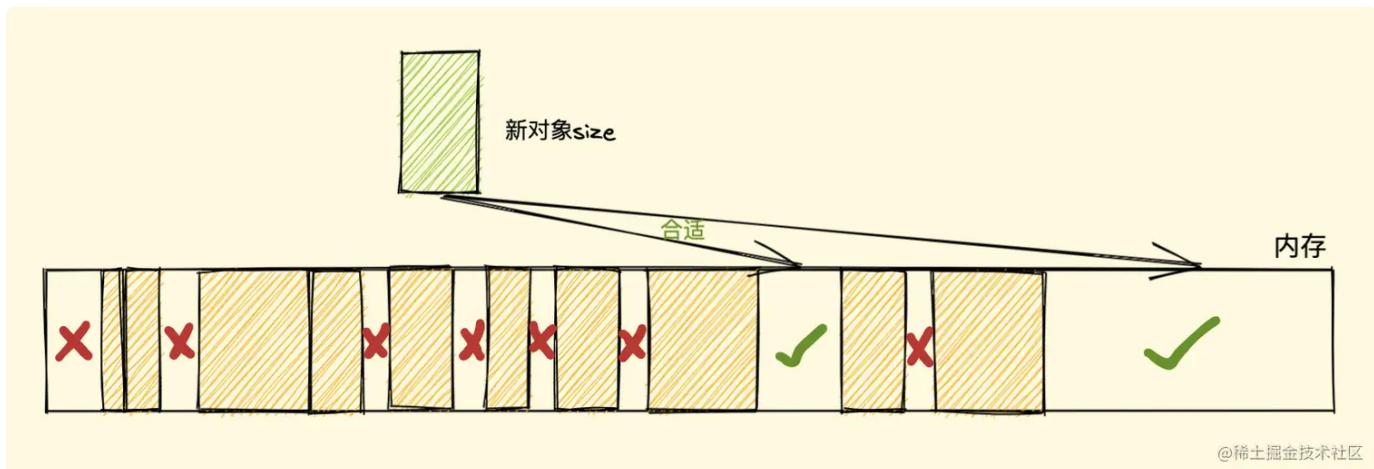
## 缺点

- **内存碎片化**，空闲内存块是不连续的，容易出现很多空闲内存块，还可能会出现分配所需内存过大的对象时找不到合适的块
- **分配速度慢**，因为即便是使用 **First-fit** 策略，其操作仍是一个  $O(n)$  的操作，最坏情况是每次都要遍历到最后，同时因为碎片化，大对象的分配效率会更慢

标记清除算法有一个很大的缺点，就是在清除之后，剩余的对象内存位置是不变的，也会导致空闲内存空间是不连续的，出现了 **内存碎片**（如下图），并且由于剩余空闲内存不是一整块，它是由不同大小内存组成的内存列表，这就牵扯出了内存分配的问题



假设我们新建对象分配内存时需要大小为 `size`, 由于空闲内存是间断的、不连续的, 则需要对空闲内存列表进行一次单向遍历找出大于等于 `size` 的块才能为其分配 (如下图)



那如何找到合适的块呢? 我们可以采取下面三种分配策略

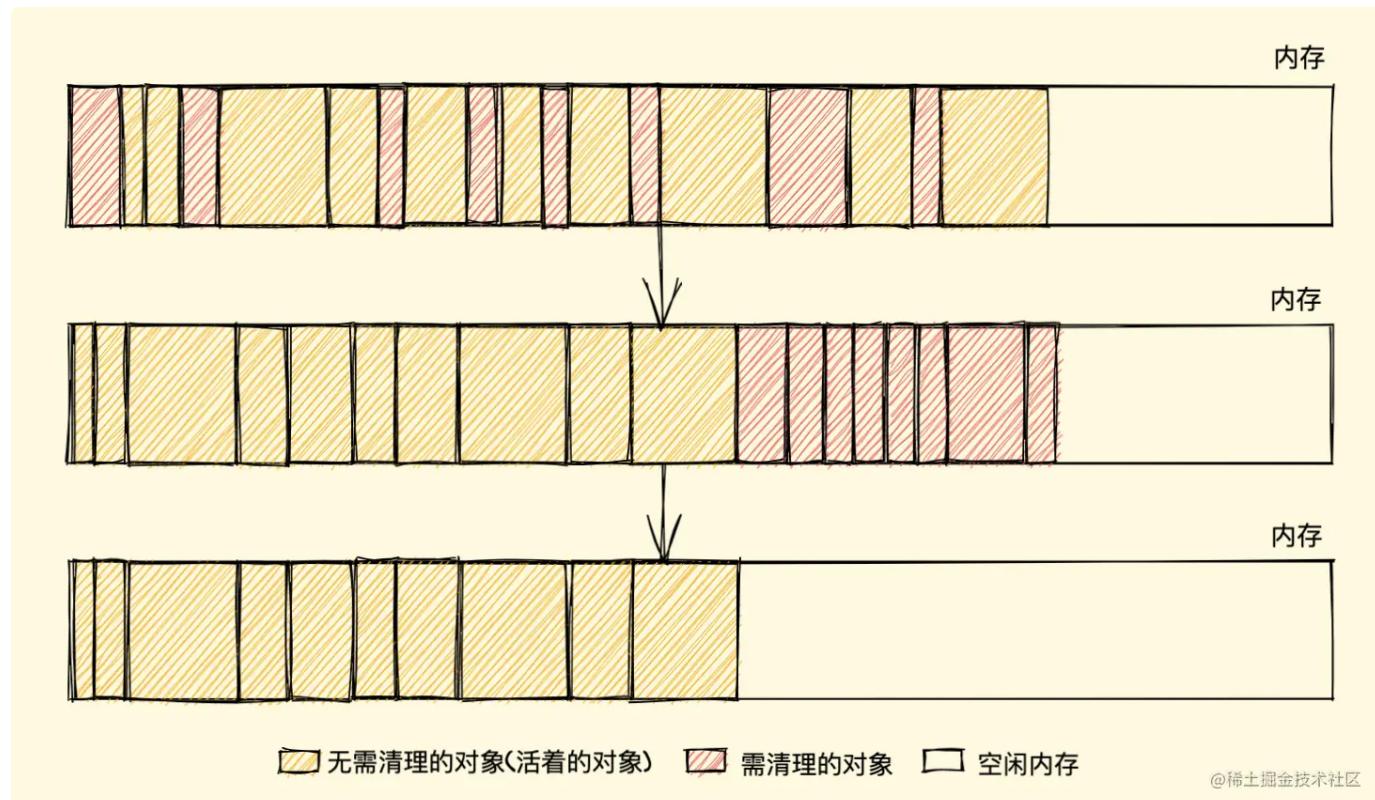
- **First-fit**, 找到大于等于 `size` 的块立即返回
- **Best-fit**, 遍历整个空闲列表, 返回大于等于 `size` 的最小分块
- **Worst-fit**, 遍历整个空闲列表, 找到最大的分块, 然后切成两部分, 一部分 `size` 大小, 并将该部分返回

这三种策略里面 **Worst-fit** 的空间利用率看起来是最合理, 但实际上切分之后会造成更多的小块, 形成内存碎片, 所以不推荐使用, 对于 **First-fit** 和 **Best-fit** 来说, 考虑到分配的速度和效率 **First-fit** 是更为明智的选择

## 优化

归根结底，标记清除算法的缺点在于清除之后剩余的对象位置不变而导致的空闲内存不连续，所以只要解决这一点，两个缺点都可以完美解决了

而 **标记整理（Mark-Compact）** 算法就可以有效地解决，它的标记阶段和标记清除算法没有什么不同，只是标记结束后，标记整理算法会将活着的对象（即不需要清理的对象）向内存的一端移动，最后清理掉边界的内存（如下图）



## 引用计数

引用计数（Reference Counting），这其实是早先的一种垃圾回收算法，它把 **对象是否不再需要** 简化定义为 **对象有没有其他对象引用到它**，如果没有引用指向该对象（零引用），对象将被垃圾回收机制回收，目前很少使用这种算法了，因为它的问题很多，不过我们还是需要了解一下它的策略是跟踪记录每个变量值被使用的次数

- 当声明了一个变量并且将一个引用类型赋值给该变量的时候这个值的引用次数就为 1
- 如果同一个值又被赋给另一个变量，那么引用数加 1
- 如果该变量的值被其他的值覆盖了，则引用次数减 1
- 当这个值的引用次数变为 0 的时候，说明没有变量在使用，这个值没法被访问了，回收空

间，垃圾回收器会在运行的时候清理掉引用次数为 0 的值占用的内存

```
1 let a = new Object() // 此对象的引用计数为 1 (a引用)
2 let b = a           // 此对象的引用计数是 2 (a,b引用)
3 a = null           // 此对象的引用计数为 1 (b引用)
4 b = null           // 此对象的引用计数为 0 (无引用)
5 ...               // GC 回收此对象
6
```

JavaScript

## 优点

- **可以立即回收垃圾**，引用计数算法的优点我们对比标记清除来看就会清晰很多，首先引用计数在引用值为 0 时，也就是在变成垃圾的那一刻就会被回收，所以它可以立即回收垃圾，而标记清除算法需要每隔一段时间进行一次，那在应用程序（JS脚本）运行过程中线程就必须要有暂停去执行一段时间的 GC，
- **操作简单**，标记清除算法需要遍历堆里的活动以及非活动对象来清除，而引用计数则只需要在引用时计数就可以了

## 缺点

- **需要空间存储计时器**，它需要一个计数器，而此计数器需要占很大的位置，因为我们也不知道被引用数量的上限，
- **无法解决循环引用无法回收的问题**，这也是最严重的

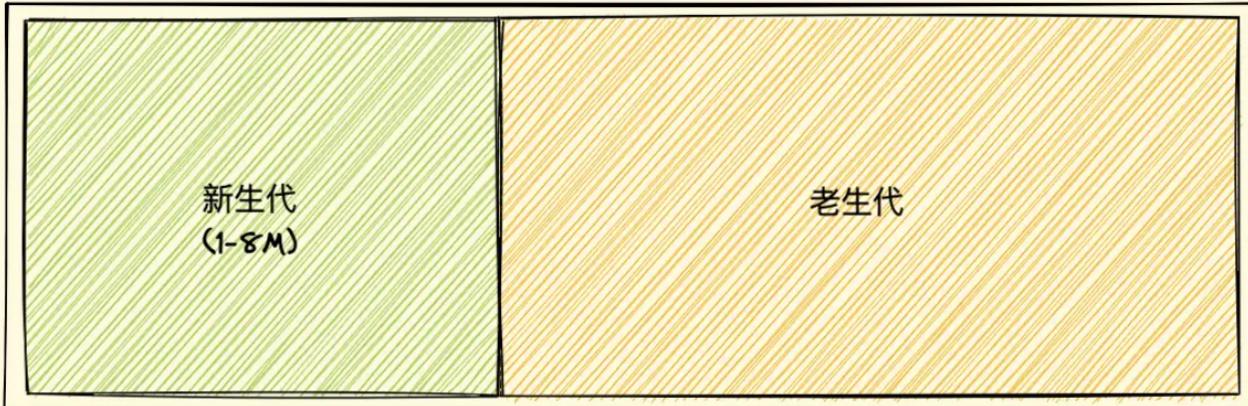
## V8的GC策略

### 分代回收

V8 的垃圾回收策略主要基于分代式垃圾回收机制，V8 中将堆内存分为**新生代**和**老生代**两区域，采用不同的垃圾回收器也就是不同的策略管理垃圾回收

新生代的对象为存活时间较短的对象，简单来说就是新产生的对象，通常只支持 1~8M 的容量，而老生代的对象为存活事件较长或常驻内存的对象，简单来说就是经历过新生代垃圾回收后还存活下来的对象，容量通常比较大

V8 整个堆内存的大小就等于新生代加上老生代的内存（如下图）



堆内存

@稀土掘金技术社区

对于新老两块内存区域的垃圾回收，V8 采用了两个垃圾回收器来管控，我们暂且将管理新生代的垃圾回收器叫做新生代垃圾回收器，同样的，我们称管理老生代的垃圾回收器叫做老生代垃圾回收器好了

## 新生代垃圾回收

新生代对象是通过一个名为 **Scavenge** 的算法进行垃圾回收，在 Scavenge 算法的具体实现中，主要采用了一种复制式的方法即 **Cheney** 算法，我们细细道来

**Cheney** 算法 中将堆内存一分为二，一个是处于使用状态的空间我们暂且称之为 使用区，一个是处于闲置状态的空间我们称之为 空闲区，如下图所示



堆内存

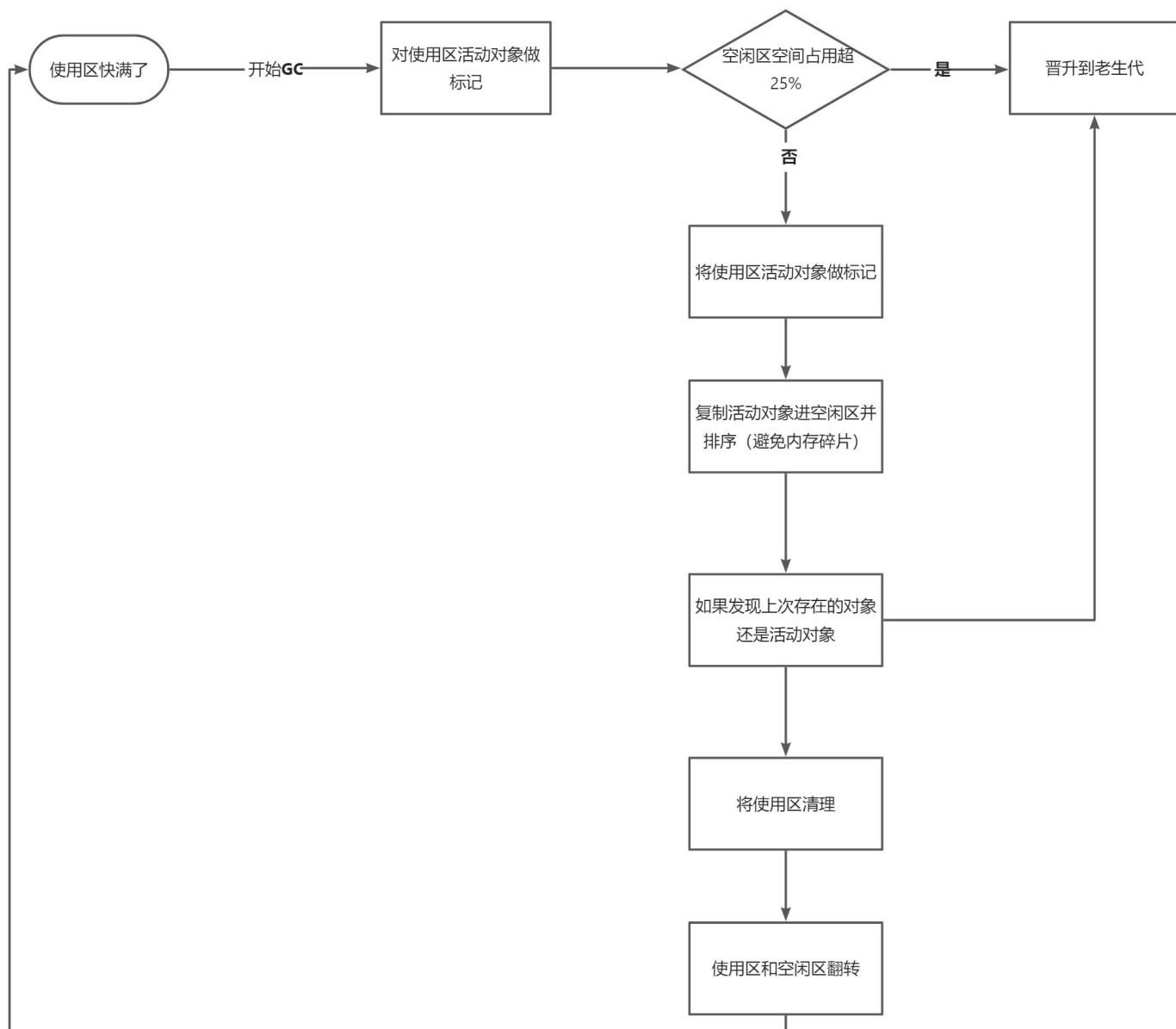
@稀土掘金技术社区

新加入的对象都会存放到使用区，当使用区快被写满时，就需要执行一次垃圾清理操作

当开始进行垃圾回收时，新生代垃圾回收器会对使用区中的活动对象做标记，标记完成之后将使用区的活动对象复制进空闲区并进行排序，随后进入垃圾清理阶段，即将非活动对象占用的空间清理掉。最后进行角色互换，把原来的使用区变成空闲区，把原来的空闲区变成使用区

当一个对象经过多次复制后依然存活，它将会被认为是生命周期较长的对象，随后会被移动到老生代中，采用老生代的垃圾回收策略进行管理

另外还有一种情况，如果复制一个对象到空闲区时，空闲区空间占用超过了 25%，那么这个对象会被直接晋升到老生代空间中，设置为 25% 的比例的原因是，当完成 Scavenge 回收后，空闲区将翻转成使用区，继续进行对象内存的分配，若占比过大，将会影响后续内存分配



## 老生代垃圾回收

相比于新生代，老生代的垃圾回收就比较容易理解了，上面我们说过，对于大多数占用空间大、存活时间长的对象会被分配到老生代里，因为老生代中的对象通常比较大，如果再如新生代一般分区然后复制来复制去就会非常耗时，从而导致回收执行效率不高，所以老生代垃圾回收器来管理其垃圾回收执行，它的整个流程就采用的就是上文所说的**标记清除算法**了

首先是标记阶段，从一组根元素开始，递归遍历这组根元素，遍历过程中能到达的元素称为活动对象，没有到达的元素就可以判断为非活动对象

清除阶段老生代垃圾回收器会直接将非活动对象，也就是数据清理掉

前面我们也提过，标记清除算法在清除后会产生大量不连续的内存碎片，过多的碎片会导致大对象无法分配到足够的连续内存，而 V8 中就采用了我们上文中说的**标记整理算法**来解决这一问题来优化空间

## 为什么需要分代机制

分代式机制把一些新、小、存活时间短的对象作为新生代，采用一小块内存频率较高的快速清理，而一些大、老、存活时间长的对象作为老生代，使其很少接受检查，新老生代的回收机制及频率是不同的，可以说此机制的出现很大程度提高了垃圾回收机制的效率

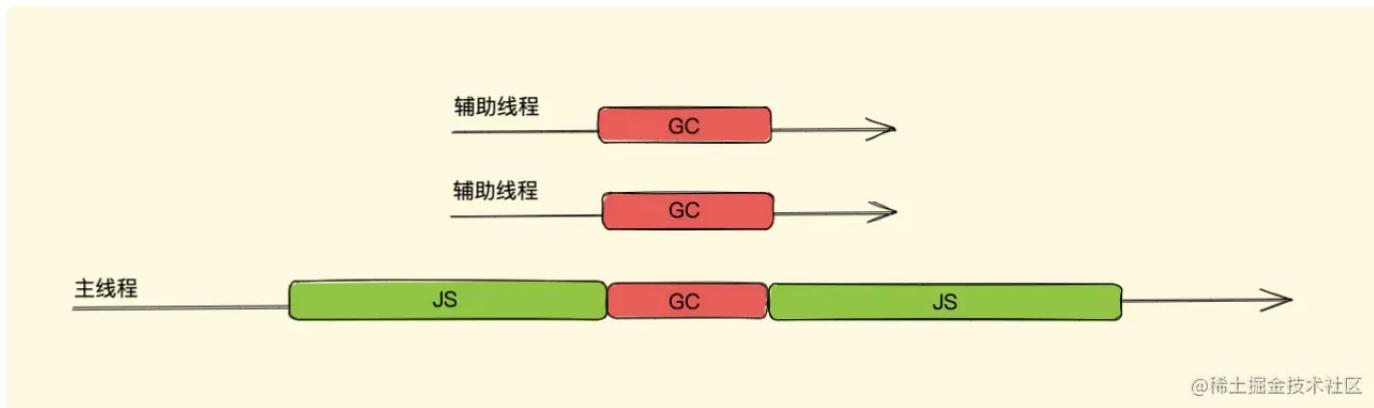
## 并行回收(Parallel)

在介绍并行之前，我们先要了解一个概念 **全停顿** (Stop-The-World)，我们都知道 JavaScript 是一门单线程的语言，它是运行在主线程上的，那在进行垃圾回收时就会阻塞 JavaScript 脚本的执行，需等待垃圾回收完毕后再恢复脚本执行，我们把这种行为叫做 **全停顿**

比如一次 GC 需要 60ms，那我们的应用逻辑就得暂停 60ms，假如一次 GC 的时间过长，对用户来说就可能造成页面卡顿等问题

既然存在执行一次 GC 比较耗时的情况，考虑到一个人盖房子难，那两个人、十个人...呢？切换到程序这边，那我们可不可以引入多个辅助线程来同时处理，这样是不是就会加速垃圾回收的执行速度呢？因此 V8 团队引入了并行回收机制

所谓并行，也就是同时的意思，它指的是垃圾回收器在主线程上执行的过程中，开启多个辅助线程，同时执行同样的回收工作



简单来说，使用并行回收，假如本来是主线程一个人干活，它一个人需要 3 秒，现在叫上了 2 个辅助线程和主线程一块干活，那三个人一块干一个人干 1 秒就完事了，但是由于多人协同办公，所以需要加上一部分多人协同（同步开销）的时间我们算 0.5 秒好了，也就是说，采用并行策略后，本来要 3 秒的活现在 1.5 秒就可以干完了

不过虽然 1.5 秒就可以干完了，时间也大大缩小了，但是这 1.5 秒内，主线程还是需要让出来的，也正是因为主线程还是需要让出来，这个过程内存是静态的，不需要考虑内存中对象的引用关系改变，只需要考虑协同，实现起来也很简单

新生代对象空间就采用并行策略，在执行垃圾回收的过程中，会启动了多个线程来负责新生代中的垃圾清理操作，这些线程同时将对象空间中的数据移动到空闲区域，这个过程中由于数据地址会发生改变，所以还需要同步更新引用这些对象的指针，此即并行回收

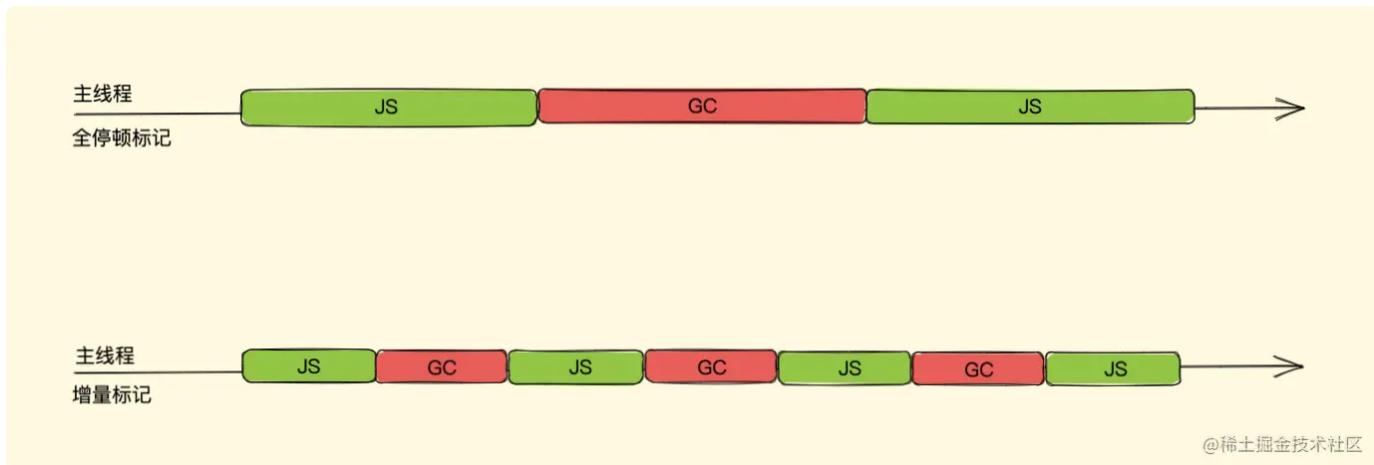
## 增量标记与懒性清理

我们上面所说的并行策略虽然可以增加垃圾回收的效率，对于新生代垃圾回收器能够有很好的优化，但是其实它还是一种全停顿式的垃圾回收方式，对于老生代来说，它的内部存放的都是一些比较大的对象，对于这些大的对象 GC 时哪怕我们使用并行策略依然可能会消耗大量时间

所以为了减少全停顿的时间，在 2011 年，V8 对老生代的标记进行了优化，从全停顿标记切换到增量标记

### 什么是增量

增量就是将一次 GC 标记的过程，分成了很多小步，每执行完一小步就让应用逻辑执行一会儿，这样交替多次后完成一轮 GC 标记（如下图）



试想一下，将一次完整的 GC 标记分次执行，那在每一小次 GC 标记执行完之后如何暂停下来去执行任务程序，而后又怎么恢复呢？那假如我们在一次完整的 GC 标记分块暂停后，执行任务程序时内存中标记好的对象引用关系被修改了又怎么办呢？

可以看出增量的实现要比并行复杂一点，V8 对这两个问题对应的解决方案分别是三色标记法与写屏障

### 三色标记法(暂停与恢复)

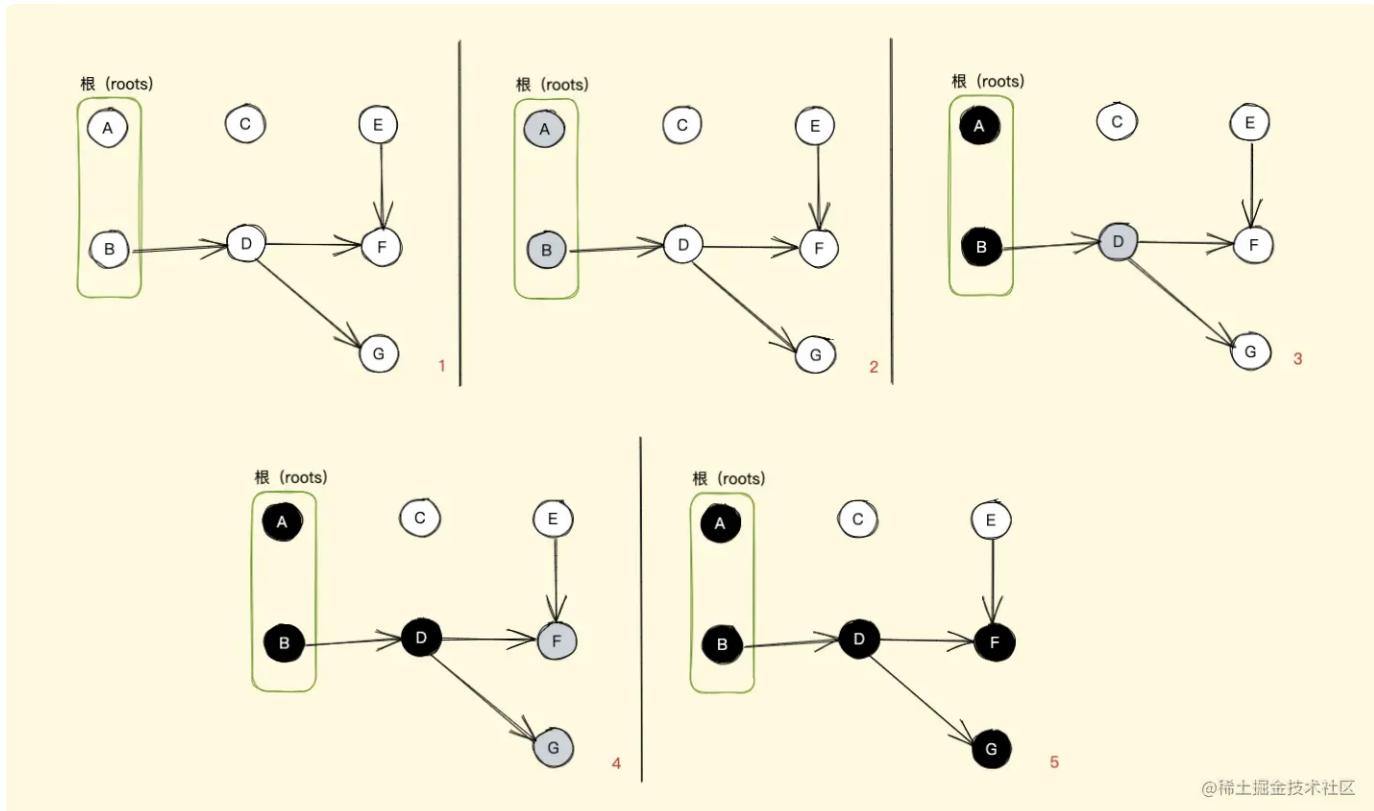
我们知道老生代是采用标记清理算法，而上文的标记清理中我们说过，也就是在没有采用增量算法之前，单纯使用黑色和白色来标记数据就可以了，其标记流程即在执行一次完整的 GC 标记前，垃圾回收器会将所有的数据置为白色，然后垃圾回收器会从一组跟对象出发，将所有能访问到的数据标记为黑色，遍历结束之后，标记为黑色的数据对象就是活动对象，剩余的白色数据对象也就是待清理的垃圾对象

如果采用非黑即白的标记策略，那在垃圾回收器执行了一段增量回收后，暂停后启用主线程去执行了应用程序中的一段 JavaScript 代码，随后当垃圾回收器再次被启动，这时候内存中黑白色都有，我们无法得知下一步走到哪里了

为了解决这个问题，V8 团队采用了一种特殊方式：三色标记法

三色标记法即使用每个对象的两个标记位和一个标记工作表来实现标记，两个标记位编码三种颜色：白、灰、黑

- 白色指的是未被标记的对象
- 灰色指自身被标记，成员变量（该对象的引用对象）未被标记
- 黑色指自身和成员变量皆被标记



@稀土掘金技术社区

如上图所示，我们用最简单的表达方式来解释这一过程，最初所有的对象都是白色，意味着回收器没有标记它们，从一组根对象开始，先将这组根对象标记为灰色并推入到标记工作表中，当回收器从标记工作表中弹出对象并访问它的引用对象时，将其自身由灰色转变成黑色，并将自身的下一个引用对象转为灰色

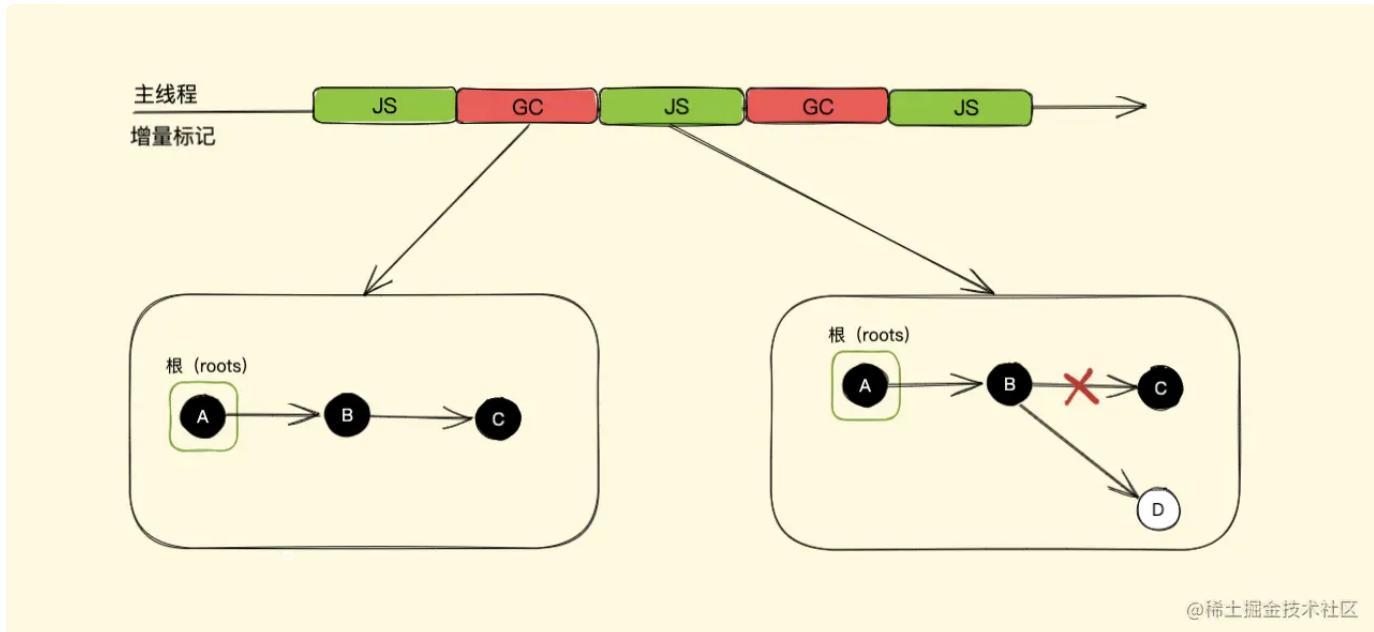
就这样一直往下走，直到没有可标记灰色的对象时，也就是无可达（无引用到）的对象了，那么剩下的所有白色对象都是无法到达的，即等待回收（如上图中的 C、E 将要等待回收）

采用三色标记法后我们在恢复执行时就好办多了，可以直接通过当前内存中有没有灰色节点来判断整个标记是否完成，如没有灰色节点，直接进入清理阶段，如还有灰色标记，恢复时直接从灰色的节点开始继续执行就可以

三色标记法的 mark 操作可以渐进执行的而不需每次都扫描整个内存空间，可以很好的配合增量回收进行暂停恢复的一些操作，从而减少 全停顿 的时间

## 写屏障(增量中修改引用)

一次完整的 GC 标记分块暂停后，执行任务程序时内存中标记好的对象引用关系被修改了，增量中修改引用，可能不太好理解，我们举个例子（如图）



@稀土掘金技术社区

假如我们有 A、B、C 三个对象依次引用，在第一次增量分段中全部标记为黑色（活动对象），而后暂停开始执行应用程序也就是 JavaScript 脚本，在脚本中我们将对象 B 的指向由对象 C 改为了对象 D，接着恢复执行下一次增量分段

这时其实对象 C 已经无引用关系了，但是目前它是黑色（代表活动对象）此一轮 GC 是不会清理 C 的，不过我们可以不考虑这个，因为就算此轮不清理等下一轮 GC 也会清理，这对我们程序运行并没有太大影响

我们再看新的对象 D 是初始的白色，按照我们上面所说，已经没有灰色对象了，也就是全部标记完毕接下来要进行清理了，新修改的白色对象 D 将在次轮 GC 的清理阶段被回收，还有引用关系就被回收，后面我们程序里可能还会用到对象 D 呢，这肯定是不对的

为了解决这个问题，V8 增量回收使用 **写屏障** (Write-barrier) 机制，即一旦有黑色对象引用白色对象，该机制会强制将引用的白色对象改为灰色，从而保证下一次增量 GC 标记阶段可以正确标记，这个机制也被称作 **强三色不变性**

那在我们上图的例子中，将对象 B 的指向由对象 C 改为对象 D 后，白色对象 D 会被强制改为灰色

## 懒性清理

增量标记其实只是对活动对象和非活动对象进行标记，对于真正的清理释放内存 V8 采用的是惰性清理(Lazy Sweeping)

增量标记完成后，惰性清理就开始了。当增量标记完成后，假如当前的可用内存足以让我们快速的执行代码，其实我们是没必要立即清理内存的，可以将清理过程稍微延迟一下，让 JavaScript 脚

本代码先执行，也无需一次性清理完所有非活动对象内存，可以按需逐一进行清理直到所有的非活动对象内存都清理完毕，后面再接着执行增量标记

### 增量标记与惰性清理的优缺点？

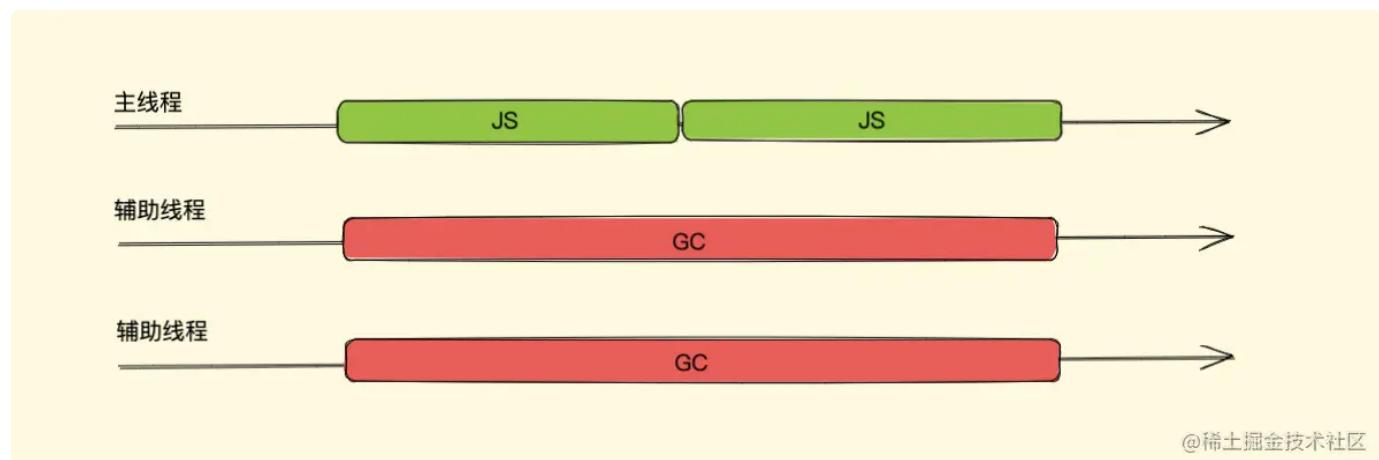
增量标记与惰性清理的出现，使得主线程的停顿时间大大减少了，让用户与浏览器交互的过程变得更加流畅。但是由于每个小的增量标记之间执行了 JavaScript 代码，堆中的对象指针可能发生了变化，需要使用写屏障技术来记录这些引用关系的变化，所以增量标记缺点也很明显：

首先是并没有减少主线程的总暂停的时间，甚至会略微增加，其次由于写屏障机制的成本，增量标记可能会降低应用程序的吞吐量（吞吐量是啥总不用说了吧）

## 并发回收(Concurrent)

前面我们说并行回收依然会阻塞主线程，增量标记同样有增加了总暂停时间、降低应用程序吞吐量两个缺点，那么怎么才能在不阻塞主线程的情况下执行垃圾回收并且与增量相比更高效呢？

这就要说到并发回收了，它指的是主线程在执行 JavaScript 的过程中，辅助线程能够在后台完成执行垃圾回收的操作，辅助线程在执行垃圾回收的时候，主线程也可以自由执行而不会被挂起（如下图）



辅助线程在执行垃圾回收的时候，主线程也可以自由执行而不会被挂起，这是并发的优点，但同样也是并发回收实现的难点，因为它需要考虑主线程在执行 JavaScript 时，堆中的对象引用关系随时都有可能发生变化，这时辅助线程之前做的一些标记或者正在进行的标记就会要有所改变，所以它需要额外实现一些读写锁机制来控制这一点，这里我们不再细说

## 再说V8中GC优化

V8 的垃圾回收策略主要基于分代式垃圾回收机制，这我们说过，关于新生代垃圾回收器，我们说使用并行回收可以很好的增加垃圾回收的效率，那老生代垃圾回收器用的哪个策略呢？我上面说了并行回收、增量标记与惰性清理、并发回收这几种回收方式来提高效率、优化体验，看着一个比一个好，那老生代垃圾回收器到底用的哪个策略？难道是并发？？内心独白：“好像。。。貌似。。。并发回收效率最高”

其实，这三种方式各有优缺点，所以在老生代垃圾回收器中这几种策略都是融合使用的  
老生代主要使用并发标记，主线程在开始执行 JavaScript 时，辅助线程也同时执行标记操作（标记操作全都由辅助线程完成）

标记完成之后，再执行并行清理操作（主线程在执行清理操作时，多个辅助线程也同时执行清理操作）

同时，清理的任务会采用增量的方式分批在各个 JavaScript 任务之间执行

## 并行回收

### 优点

- 由于是全停顿，内存中数据引用关系不变，所以实现简单

### 缺点

- 对于老生代中一些大对象，消耗时间过久可能会阻塞程序，有卡顿

## 增量标记

### 优点

- 交替执行GC，更加流畅

### 缺点

- 由于写屏障机制的成本，增量标记可能会降低应用程序的吞吐量，相比并行回收，引入了三色标记来控制GC的中断，引入写屏障来保证增量修改数据引用的正确性
- 没有减少主线程的总暂停的时间，甚至会略微增加

## 并发回收

### 优点

- 辅助线程在执行垃圾回收的时候，主线程也可以自由执行而不会被挂起

### 缺点

- 由于GC的时候主线程在执行，堆中的对象引用关系随时都有可能发生变化，这时辅助线程之

前做的一些标记或者正在进行的标记就要有所改变，所以它需要额外实现一些读写锁机制来控制

# 作用域

作用域是指程序源代码中定义变量的区域。

作用域规定了如何查找变量，也就是确定当前执行代码对变量的访问权限。

JavaScript 采用词法作用域(lexical scoping)，也就是静态作用域。

## 静态作用域

JavaScript 采用的是词法作用域，函数的作用域在函数定义的时候就决定了。

```
1 var value = 1;
2
3 function foo() {
4     console.log(value);
5 }
6
7 function bar() {
8     var value = 2;
9     foo();
10}
11
12 bar();
13
14 // 结果是 1
```

HTML |

执行 foo 函数，先从 foo 函数内部查找是否有局部变量 value，如果没有，就根据书写的位置，查找上面一层的代码，也就是 value 等于 1，所以结果会打印 1。

## 动态作用域

我们以bat脚本为例，bat脚本是动态作用域

```
1 @echo off
2 set value=1
3
4 call :bar
5 goto :eof
6
7 :bar
8 set value=2
9 echo bar: %value%
10 call :foo
11 goto :eof
12
13 :foo
14 echo foo: %value%
15 goto :eof
16
17 # bar: 2
18 # foo: 2
```

执行 foo 函数，依然从 foo 函数内部查找是否有局部变量 value。如果没有，就从调用函数的作用域，也就是 bar 函数内部查找 value 变量，所以结果会打印 2。

## 思考

```
1 var scope = "global scope";
2 function checkscope(){
3     var scope = "local scope";
4     function f(){
5         return scope;
6     }
7     return f();
8 }
9 checkscope();
10
11 var scope = "global scope";
12 function checkscope(){
13     var scope = "local scope";
14     function f(){
15         return scope;
16     }
17     return f;
18 }
19 checkscope()();
```

以上两段代码都会输出 `local scope`，因为JavaScript 函数的执行用到了作用域链，这个作用域链是在函数定义的时候创建的。嵌套的函数 `f()` 定义在这个作用域链里，其中的变量 `scope` 一定是局部变量，不管何时何地执行函数 `f()`，这种绑定在执行 `f()` 时依然有效。

# let和const

## let命令

`let` 声明的变量只在代码块内生效

`for` 循环就很适合 `let` 命令

### ▼ let在for循环中使用

JavaScript

```
1 var a = [];
2 for (let i = 0; i < 10; i++) {
3   a[i] = function () {
4     console.log(i);
5   };
6 }
7 a[6](); // 6
```

另外，`for` 循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
1 for (let i = 0; i < 3; i++)
2 {
3   let i = 'abc';
4   console.log(i);
5 }
6 // abc
7 // abc
8 // abc
9
10 //上面代码正确运行，输出了 3 次abc。这表明函数内部的变量i与循环变量i不在同一个作用域，有各自单独的作用域
```

JavaScript

## 不存在变量提升

## ▼ let不存在变量提升

JavaScript |

```
1 // var 的情况
2 console.log(foo); // 输出undefinedvar
3 foo = 2;
4
5 // let 的情况
6 console.log(bar); // 报错ReferenceError
7 let bar = 2;
```

## 暂时性死区

ES6 明确规定，如果区块中存在 `let` 和 `const` 命令，这个区块对这些命令声明的变量，从一开始 就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

### ▼ 暂时性死区 (TDZ)

JavaScript |

```
1 if (true) {
2     // TDZ开始
3     tmp = 'abc'; // ReferenceError
4     console.log(tmp); // ReferenceError
5
6     let tmp; // TDZ结束
7     console.log(tmp); // undefined
8
9     tmp = 123;
10    console.log(tmp); // 123
11 }
```

“暂时性死区”也意味着 `typeof` 不再是一个百分之百安全的操作。

### ▼ 暂时性死区导致`typeof`报错

JavaScript |

```
1 typeof x; // ReferenceError
2 let x;
```

有些“死区”比较隐蔽，不太容易发现。

```
1 function bar(x = y, y = 2) {  
2     return [x, y];  
3 }  
4  
5 bar(); // 报错  
6  
7 //参数x默认值等于另一个参数y，而此时y还没有声明，属于“死区”。如果y的默认值是x，就不会报  
错，因为此时x已经声明了。  
8  
9 function bar(x = 2, y = x) {  
10    return [x, y];  
11 }  
12 bar(); // [2, 2]
```

## 不允许重复声明

**let** 不允许在相同作用域内，重复声明同一个变量。

## 不允许重复声明

JavaScript |

```
1 // 报错
2 function func() {
3     let a = 10;
4     var a = 1;
5 }
6
7 // 报错
8 function func() {
9     let a = 10;
10    let a = 1;
11 }
12
13 function func(arg) {
14     let arg;
15 }
16 func() // 报错
17
18 function func(arg) {
19     {
20         let arg;
21     }
22 }
23 func() // 不报错
```

# 块级作用域

未引入块级作用域之前只有全局作用域和函数作用域，会有一下几个问题

1、内层变量覆盖外层变量

▼ 内层变量覆盖外层变量

JavaScript |

```
1 var tmp = new Date();
2
3 function f() {
4     console.log(tmp);
5     if (false) {
6         var tmp = 'hello world';
7     }
8 }
9
10 f(); // undefined
```

## 2、容易泄露变量到全局

▼ 用来计数的循环变量泄露为全局变量

JavaScript |

```
1 var s = 'hello';
2
3 for (var i = 0; i < s.length; i++) {
4     console.log(s[i]);
5 }
6
7 console.log(i); // 5
```

## 引入块级作用域之后

▼ 变量只在块级作用域生效

JavaScript |

```
1 function f1() {
2     let n = 5;
3     if (true) {
4         let n = 10;
5     }
6     console.log(n); // 5
7 }
8
9 //外层代码块不受内层代码块的影响。如果两次都使用var定义变量n，最后输出的值才是 10。
```

## 允许嵌套作用域

JavaScript

```
1  {{{{
2    let insane = 'Hello World'
3    console.log(insane); // 报错
4 }}};}
5
6 //上面代码使用了一个五层的块级作用域，每一层都是一个单独的作用域。第四层作用域无法读取第五层作用域的内部变量。
```

块级作用域的出现取代了立即执行函数的江湖地位

## 块级作用域取代IIFE

JavaScript

```
1 // IIFE 写法
2 (function () {
3   var tmp = ...
4   ...
5 })();
6
7 // 块级作用域写法
8 {
9   let tmp = ...
10  ...
11 }
```

# 块级作用域和函数声明

ES5 规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

## 情况一

JavaScript

```
1 // 情况一
2 if (true) {
3   function f() {}
4 }
5
6 // 情况二
7 try {
8   function f() {}
9 } catch(e) {
10   // ...
11 }
```

上面两种函数声明，根据 ES5 的规定都是非法的。

但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。

#### ▼ ES5环境下函数声明

JavaScript |

```
1  function f() { console.log('I am outside!'); }
2
3  (function () {
4    if (false) {
5      // 重复声明一次函数f
6      function f() { console.log('I am inside!'); }
7    }
8
9    f();
10 }());
11
12 //上面代码在 ES5 中运行，会得到“I am inside!”，因为在if内声明的函数f会被提升到函数头部，实际运行的代码如下。
13
14 // ES5 环境
15 function f() { console.log('I am outside!'); }
16
17 (function () {
18   function f() { console.log('I am inside!'); }
19   if (false) {
20     }
21   f();
22 }());
```

但是在ES6环境下如果改变了块级作用域内声明的函数的处理规则，显然会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6 在[附录 B](#)里面规定，浏览器的实现可以不遵守上面的规定，有自己的[行为方式](#)。

- 允许在块级作用域内声明函数。
- 函数声明类似于 `var`，即会提升到全局作用域或函数作用域的头部。
- 同时，函数声明还会提升到所在的块级作用域的头部。

```
1 // 浏览器的 ES6 环境
2 function f() { console.log('I am outside!'); }
3
4 (function () {
5   if (false) {
6     // 重复声明一次函数f
7     function f() { console.log('I am inside!'); }
8   }
9
10 f();
11 })();
12 // Uncaught TypeError: f is not a function
13
14 //等同于
15 // 浏览器的 ES6 环境
16 function f() { console.log('I am outside!'); }
17 (function () {
18   var f = undefined;
19   if (false) {
20     function f() { console.log('I am inside!'); }
21   }
22
23   f();
24 })();
25 // Uncaught TypeError: f is not a function
```

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

## 块级作用域内部，优先使用函数表达式

JavaScript |

```
1 // 块级作用域内部的函数声明语句，建议不要使用
2 {
3     let a = 'secret';
4     function f() {
5         return a;
6     }
7 }
8
9 // 块级作用域内部，优先使用函数表达式
10 {
11     let a = 'secret';
12     let f = function () {
13         return a;
14     };
15 }
```

另外，还有一个需要注意的地方。ES6 的块级作用域必须有大括号，如果没有大括号，JavaScript 引擎就认为不存在块级作用域。

## ES6 的块级作用域必须有大括号

JavaScript |

```
1 // 第一种写法，报错
2 if (true) let x = 1;
3
4 // 第二种写法，不报错
5 if (true) {
6     let x = 1;
7 }
```

# const命令

**const** 声明一个只读的常量。一旦声明，常量的值就不能改变。并且需要初始化，不能留到以后赋值。

**const** 实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。

对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。

但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，**const** 只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

## ▼ const只保证指针不变

JavaScript |

```
1 const foo = {};
2
3 // 为 foo 添加一个属性，可以成功
4 foo.prop = 123;
5 foo.prop // 123
6
7 // 将 foo 指向另一个对象，就会报错
8 foo = {}; // TypeError: "foo" is read-only
```

# ES6声明变量的6种方式

ES5 只有两种声明变量的方法：`var` 命令和 `function` 命令。ES6 除了添加 `let` 和 `const` 命令，后面章节还会提到，另外两种声明变量的方法：`import` 命令和 `class` 命令。所以，ES6 一共有 6 种声明变量的方法。

## 顶层对象属性

顶层对象的属性与全局变量挂钩，被认为是 JavaScript 语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，`window` 对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

ES6 为了改变这一点，一方面规定，为了保持兼容性，`var` 命令和 `function` 命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，`let` 命令、`const` 命令、`class` 命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

```
1 var a = 1;
2 // 如果在 Node 的 REPL 环境，可以写成 global.a
3 // 或者采用通用方法，写成 this.a
4 window.a // 1
5
6 let b = 1;
7 window.b // undefined
```

# globalThis对象

JavaScript 语言存在一个顶层对象，它提供全局环境（即全局作用域），所有代码都是在这个环境中运行。但是，顶层对象在各种实现里面是不统一的。

- 浏览器里面，顶层对象是 `window`，但 Node 和 Web Worker 没有 `window`。
- 浏览器和 Web Worker 里面，`self` 也指向顶层对象，但是 Node 没有 `self`。
- Node 里面，顶层对象是 `global`，但其他环境都不支持。

同一段代码为了能够在各种环境，都能取到顶层对象，现在一般是使用 `this` 关键字，但是有局限性。

- 全局环境中，`this` 会返回顶层对象。但是，Node.js 模块中 `this` 返回的是当前模块，ES6 模块中 `this` 返回的是 `undefined`。
- 函数里面的 `this`，如果函数不是作为对象的方法运行，而是单纯作为函数运行，`this` 会指向顶层对象。但是，严格模式下，这时 `this` 会返回 `undefined`。
- 不管是严格模式，还是普通模式，`new Function('return this')()`，总是会返回全局对象。但是，如果浏览器用了 CSP (Content Security Policy，内容安全策略)，那么 `eval`、`new Function` 这些方法都可能无法使用。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

## 各个环境获取顶层对象

JavaScript

```
1 // 方法一
2 (typeof window !== 'undefined'
3   ? window
4   : (typeof process === 'object' &&
5     typeof require === 'function' &&
6     typeof global === 'object')
7   ? global
8   : this);
9
10 // 方法二
11 var getGlobal = function () {
12   if (typeof self !== 'undefined') { return self; }
13   if (typeof window !== 'undefined') { return window; }
14   if (typeof global !== 'undefined') { return global; }
15   throw new Error('unable to locate global object');
16 };
```

ES2020 在语言标准的层面，引入 `globalThis` 作为顶层对象。也就是说，任何环境下，`globalThis` 都是存在的，都可以从它拿到顶层对象，指向全局环境下的 `this`。

# 变量的解构赋值

## 数组解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构。

### 结构基本用法

JavaScript

```
1 let [foo, [[bar], baz]] = [1, [[2], 3]];
2 foo // 1
3 bar // 2
4 baz // 3
5
6 let [ , , third] = ["foo", "bar", "baz"];
7 third // "baz"
8
9 let [x, , y] = [1, 2, 3];
10 x // 1
11 y // 3
12
13 let [head, ...tail] = [1, 2, 3, 4];
14 head // 1
15 tail // [2, 3, 4]
16
17 let [x, y, ...z] = ['a'];
18 x // "a"
19 y // undefined
20 z // []
```

如果解构不成功，变量的值就等于 `undefined`。

### 解构失败

JavaScript

```
1 let [foo] = [];
2 let [bar, foo] = [1];
3
4 foo //undefined
```

还有一种情况是不完全解构

## ▼ 不完全解构

JavaScript |

```
1 let [x, y] = [1, 2, 3];
2 x // 1
3 y // 2
4
5 let [a, [b], d] = [1, [2, 3], 4];
6 a // 1
7 b // 2
8 d // 4
```

只要某种数据结构具有 Iterator 接口，都可以采用数组形式的解构赋值。

## ▼ Set解构

JavaScript |

```
1 let [x, y, z] = new Set(['a', 'b', 'c']);
2 x // "a"
```

## ▼ Generate函数解构

JavaScript |

```
1 function* fibs() {
2     let a = 0;
3     let b = 1;
4     while (true) {
5         yield a;
6         [a, b] = [b, a + b];
7     }
8 }
9
10 let [first, second, third, fourth, fifth, sixth] = fibs();
11 sixth // 5
```

# 默认值

## ▼ 解构允许赋默认值

JavaScript |

```
1 let [foo = true] = [];
2 foo // true
3
4 let [x, y = 'b'] = ['a']; // x='a', y='b'
5 let [x, y = 'b'] = ['a', undefined]; // x='a', y='b'
```

ES6 内部使用严格相等运算符（`==`）, 判断一个位置是否有值。所以, 只有当一个数组成员严格等于 `undefined`, 默认值才会生效。

▼ 只有严格相等`undefined`, 默认值才生效

JavaScript |

```
1 let [x = 1] = [undefined];
2 x // 1
3
4 let [x = 1] = [null];
5 x // null
```

默认值可以引用解构赋值的其他变量, 但该变量必须已经声明。

▼ 引用其他变量做默认值

JavaScript |

```
1 let [x = 1, y = x] = [];
2 let [x = 1, y = x] = [2];
3 let [x = 1, y = x] = [1, 2];
4 let [x = y, y = 1] = [];
      // ReferenceError: y is not defined
```

## 对象的解构赋值

数组的解构赋值元素是按次序排列的, 变量的取值由它的位置决定;

而对象的属性没有次序, 变量必须与属性同名, 才能取到正确的值。

▼ 对象的解构赋值

JavaScript |

```
1 let { bar, foo } = { foo: 'aaa', bar: 'bbb' };
2 foo // "aaa"
3 bar // "bbb"
4
5 let { baz } = { foo: 'aaa', bar: 'bbb' };
6 baz // undefined
```

如果变量名与属性名不一致, 则需要使用别名

## 使用别名解构赋值

JavaScript

```
1 let { foo: baz } = { foo: 'aaa', bar: 'bbb' };
2 baz // "aaa"
3
4 let obj = { first: 'hello', last: 'world' };
5 let { first: f, last: l } = obj;
6 f // 'hello'
7 l // 'world'
```

上面代码中 `first` 和 `last` 是模式，真正被赋值的是 `f` 和 `l`

对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

对象的解构赋值是下面形式的简写

## 完整的对象解构赋值

JavaScript

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

# 默认值

默认值生效的条件是，对象的属性值严格等于 `undefined`。

## 对象解构赋值默认值

JavaScript

```
1 var {x = 3} = {};
2 x // 3
3
4 var {x, y = 5} = {x: 1};
5 x // 1
6 y // 5
7
8 var {x: y = 3} = {};
9 y // 3
10
11 var {x: y = 3} = {x: 5};
12 y // 5
13
14 var { message: msg = 'Something went wrong' } = {};
15 msg // "Something went wrong"
```

# 注意点

1、如果要将一个已经声明的变量用于解构赋值，必须非常小心。

▼ 将一个已经声明的变量用于解构赋值

JavaScript

```
1 // 错误的写法
2 let x;
3 {x} = {x: 1};
4 // SyntaxError: syntax error
5
6 // 正确的写法
7 let x;
8 ({x} = {x: 1});
```

因为 JavaScript 引擎会将 `{x}` 理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。

2、解构赋值允许等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式。

▼ 不声明变量也能解析

JavaScript

```
1 ({} = [true, false]);
2 ({} = 'abc');
3 ({} = []);
```

3、由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

▼ 对数组使用对象解构赋值

JavaScript

```
1 let arr = [1, 2, 3];
2 let {0 : first, [arr.length - 1] : last} = arr;
3 first // 1
4 last // 3
```

## 字符串解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

## 字符串解构赋值

JavaScript |

```
1 const [a, b, c, d, e] = 'hello';
2 a // "h"
3 b // "e"
4 c // "l"
5 d // "l"
6 e // "o"
```

# 数值和布尔值解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

## 数值和布尔值解构赋值

JavaScript |

```
1 let {toString: s} = 123;
2 s === Number.prototype.toString // true
3
4 let {toString: s} = true;
5 s === Boolean.prototype.toString // true
```

上面代码中，数值和布尔值的包装对象都有 `toString` 属性，因此变量 `s` 都能取到值。

解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于 `undefined` 和 `null` 无法转为对象，所以对它们进行解构赋值，都会报错。

## undefined和null解构报错

JavaScript |

```
1 let { prop: x } = undefined; // TypeError
2 let { prop: y } = null; // TypeError
```

# 函数参数解构赋值

## 函数参数解构赋值

JavaScript |

```
1 function add([x, y]){
2     return x + y;
3 }
4
5 add([1, 2]); // 3
```

并且函数参数解构也可以使用默认值

▼ 函数参数解构使用默认值

JavaScript

```
1 function move({x = 0, y = 0} = {}) {
2     return [x, y];
3 }
4
5 move({x: 3, y: 8}); // [3, 8]
6 move({x: 3}); // [3, 0]
7 move({}); // [0, 0]
8 move(); // [0, 0]
```

注意如果是对函数参数使用默认值则会有所不同

▼ 函数参数使用默认值

JavaScript

```
1 function move({x, y} = { x: 0, y: 0 }) {
2     return [x, y];
3 }
4
5 move({x: 3, y: 8}); // [3, 8]
6 move({x: 3}); // [3, undefined]
7 move({}); // [undefined, undefined]
8 move(); // [0, 0]
```

## 用途

1、交换变量的值

▼ 交换变量的值

JavaScript

```
1 let x = 1;
2 let y = 2;
3
4 [x, y] = [y, x];
```

2、从函数返回多个值

## 从函数返回多个值

JavaScript |

```
1 // 返回一个数组
2
3 function example() {
4     return [1, 2, 3];
5 }
6 let [a, b, c] = example();
7
8 // 返回一个对象
9
10 function example() {
11     return {
12         foo: 1,
13         bar: 2
14     };
15 }
16 let { foo, bar } = example();
```

## 3、函数参数的定义

### 函数参数的定义

JavaScript |

```
1 // 参数是一组有次序的值
2 function f([x, y, z]) { ... }
3 f([1, 2, 3]);
4
5 // 参数是一组无次序的值
6 function f({x, y, z}) { ... }
7 f({z: 3, y: 2, x: 1});
```

## 4、提取 JSON 数据

### 提取 JSON 数据

JavaScript |

```
1 let jsonData = {
2     id: 42,
3     status: "OK",
4     data: [867, 5309]
5 };
6
7 let { id, status, data: number } = jsonData;
8
9 console.log(id, status, number);
10 // 42, "OK", [867, 5309]
```

## 5、函数参数的默认值

### ▶ 函数参数的默认值

JavaScript

```
1  jQuery.ajax = function (url, {
2    async = true,
3    beforeSend = function () {},
4    cache = true,
5    complete = function () {},
6    crossDomain = false,
7    global = true,
8    // ... more config
9  } = {}) {
10   // ... do stuff
11 }
```

## 6、遍历 Map 结构

### ▶ 遍历 Map 结构

JavaScript

```
1  const map = new Map();
2  map.set('first', 'hello');
3  map.set('second', 'world');
4
5  for (let [key, value] of map) {
6    console.log(key + " is " + value);
7  }
8  // first is hello
9  // second is world
```

## 7、输入模块的指定方法

### ▶ 输入模块的指定方法

JavaScript

```
1  const { SourceMapConsumer, SourceNode } = require("source-map");
2
```

# 字符串的扩展

## 字符的Unicode表示

ES6加强了对Unicode码点的表示。允许采用 `\uxxxx` 形式表示一个字符，其中 `xxxx` 表示字符的 Unicode 码点。但是这种表示法只限于码点在 `\u0000 ~ \uFFFF` 之间的字符。超出这个范围的字符，必须用两个双字节的形式表示。或者将码点放入大括号即可正常解析

### 字符串码点表示

JavaScript

```
1  "\uD842\uDFB7"  
2  // "𠮷"  
3  
4  "\u20BB7"  
5  // " 7"  
6  
7  "\u{20BB7}"  
8  // "𠮷"
```

## 字符串的遍历器接口

ES6 为字符串添加了遍历器接口，使得字符串可以被 `for...of` 循环遍历。

这个遍历器最大的有点是可以识别码点

### for of遍历识别码点

JavaScript

```
1  let text = String.fromCodePoint(0x20BB7);  
2  
3  for (let i = 0; i < text.length; i++) {  
4      console.log(text[i]);  
5  }  
6  // "  
7  // "  
8  
9  for (let i of text) {  
10     console.log(i);  
11  }  
12  // "𠮷"
```

# 模板字符串

模板字符串解决了传统字符串拼接的问题

## 特点

1、如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

### ▼ 模板字符串保留空格和缩进

JavaScript

```
1  $('#list').html(`  
2    <ul>  
3      <li>first</li>  
4      <li>second</li>  
5    </ul>  
6  `);
```

2、模板字符串可以嵌套

3、如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的 `toString` 方法。

## 标签模板

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

但是，如果模板字符里面有变量，就不是简单的调用了，而是会将模板字符串先处理成多个参数，再调用函数。

## ▼ 标签模板

JavaScript |

```
1 let a = 5;
2 let b = 10;
3
4 tag`Hello ${ a + b } world ${ a * b }`;
5 // 等同于
6 tag(['Hello ', ' world ', ''], 15, 50);
7
8 function tag(stringArr, value1, value2){
9     // ...
10 }
11
12 // 等同于
13
14 function tag(stringArr, ...values){
15     // ...
16 }
```

**tag** 函数所有参数的实际值如下。

- 第一个参数: `['Hello ', ' world ', '']`
- 第二个参数: 15
- 第三个参数: 50

也就是说, **tag** 函数实际上以下面的形式调用。

```
tag(['Hello ', ' world ', ''], 15, 50)
```

标签模板的一个重要应用, 就是过滤 HTML 字符串, 防止用户输入恶意内容。

## ▼ 防止用户输入恶意内容

JavaScript |

```
1 let message =
2   SaferHTML`<p>${sender} has sent you a message.</p>`;
3
4 function SaferHTML(templateData) {
5   let s = templateData[0];
6   for (let i = 1; i < arguments.length; i++) {
7     let arg = String(arguments[i]);
8
9     // Escape special characters in the substitution.
10    s += arg.replace(/&/g, "&")
11      .replace(/</g, "<")
12      .replace(/>/g, ">");
13
14    // Don't escape special characters in the template.
15    s += templateData[i];
16  }
17  return s;
18}
19
20
21
22 let sender = '<script>alert("abc")</script>'; // 恶意代码
23 let message = SaferHTML`<p>${sender} has sent you a message.</p>`;
24
25 message
26 // <p>&lt;script&gt;alert("abc")&lt;/script&gt; has sent you a message.</p>
27 >
```

标签模板的另一个应用，就是多语言转换（国际化处理）。

## ▼ 多语言转换

JavaScript |

```
1 i18n`Welcome to ${siteName}, you are visitor number ${visitorNumber}!`
2 // "欢迎访问xxx, 您是第xxxx位访问者!"
```

除此之外，你甚至可以使用标签模板，在 JavaScript 语言之中嵌入其他语言

▼ jsx标签模板

JavaScript |

```
1  jsx`  
2    <div>  
3      <input  
4        ref='input'  
5        onChange='${this.handleChange}'  
6        defaultValue='${this.state.value}' />  
7        ${this.state.value}  
8    </div>  
9  `
```

# 字符串的新增方法

## String.fromCodePoint()

### fromCharCode和fromCodePoint

JavaScript |

```
1 //ES5 提供String.fromCharCode()方法，用于从 Unicode 码点返回对应字符，但是这个方  
2 // ES6 提供了String.fromCodePoint()方法，可以识别大于0xFFFF的字符，弥补了String.f  
3 // "𠮷"  
4 // 如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回  
5 // "𠮷"  
6 // 如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回  
7 // "𠮷"  
8 // 如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回  
9 // true
```

注意，`fromCodePoint` 方法定义在 `String` 对象上，而 `codePointAt` 方法定义在字符串的实例对象上。

## String.raw

该方法返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，往往用于模板字符串的处理方法。

### String.raw转义字符串

JavaScript |

```
1 String.raw`Hi\n${2+3}!`  
2 // 实际返回 "Hi\\n5!"，显示的是转义后的结果 "Hi\n5!"  
3  
4 String.raw`Hi\u000A!`;  
5 // 实际返回 "Hi\\u000A!"，显示的是转义后的结果 "Hi\u000A!"
```

# 实例方法

## codePointAt()

对于Unicode 码点大于 `0xFFFF` 的字符，`charAt()` 方法无法读取整个字符，`charCodeAt()` 方法只能分别返回前两个字节和后两个字节的值。

ES6 提供了 `codePointAt()` 方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。

#### 四字节码点表示

JavaScript

```
1 var s = "𠮷";
2
3 s.length // 2
4 s.charCodeAt(0) // ''
5 s.charCodeAt(1) // ''
6 s.charCodeAt(0) // 55362
7 s.charCodeAt(1) // 57271
8
9 let s = '𠮷a';
10
11 s.codePointAt(0) // 134071
12 s.codePointAt(1) // 57271
13
14 s.codePointAt(2) // 97
```

上面代码中，JavaScript 将“𠮷a”视为三个字符，`codePointAt` 方法在第一个字符上，正确地识别了“𠮷”，返回了它的十进制码点 134071（即十六进制的 `20BB7`）。在第二个字符（即“𠮷”的后两个字节）和第三个字符“a”上，`codePointAt()` 方法的结果与 `charCodeAt()` 方法相同。

你可能注意到了，`codePointAt()` 方法的参数，仍然是不正确的。比如，上面代码中，字符 `a` 在字符串 `s` 的正确位置序号应该是 1，但是必须向 `codePointAt()` 方法传入 2。解决这个问题的一个办法是使用 `for...of` 循环，因为它会正确识别 32 位的 UTF-16 字符。

#### for of遍历多字节字符

JavaScript

```
1 let s = '𠮷a';
2 for (let ch of s) {
3   console.log(ch.codePointAt(0).toString(16));
4 }
5 // 20bb7
6 // 61
```

## includes(), startsWith(), endsWith()

- `includes()`: 返回布尔值，表示是否找到了参数字符串。
- `startsWith()`: 返回布尔值，表示参数字符串是否在原字符串的头部。

- `endsWith()`: 返回布尔值，表示参数字符串是否在原字符串的尾部。

▼ 支持指定搜索位置

JavaScript

```

1 let s = 'Hello world!';
2
3 s.startsWith('Hello') // true
4 s.endsWith('!') // true
5 s.includes('o') // true
6
7 let s = 'Hello world!';
8
9 s.startsWith('world', 6) // true
10 s.endsWith('Hello', 5) // true
11 s.includes('Hello', 6) // false

```

## repeat()

返回一个新字符串，表示将原字符串重复n次。

▼ repeat使用

JavaScript

```

1 'x'.repeat(3) // "xxx"
2 'hello'.repeat(2) // "hellohello"
3 'na'.repeat(0) // ""
4
5 //小数会被取整
6 'na'.repeat(2.9) // "nana"
7
8 //负数会报错
9 'na'.repeat(Infinity)
10 // RangeError
11 'na'.repeat(-1)
12 // RangeError
13
14 //0 到-1 之间的小数，则等同于 0，这是因为会先进行取整运算
15 'na'.repeat(-0.9) // ""
16
17 //NaN等同于 0
18 'na'.repeat(NaN) // ""
19
20 //repeat的参数是字符串，则会先转换成数字
21 'na'.repeat('na') // ""
22 'na'.repeat('3') // "nanana"

```

## padStart(), padEnd()

如果某个字符串不够指定长度，会在头部或尾部补全。`padStart()` 用于头部补全，`padEnd()` 用于尾部补全。

```
▼ JavaScript

1  'x'.padStart(5, 'ab') // 'ababx'
2  'x'.padStart(4, 'ab') // 'abax'
3
4  'x'.padEnd(5, 'ab') // 'xabab'
5  'x'.padEnd(4, 'ab') // 'xaba'
6
7  //省略第二个参数默认用空格补全
8  'x'.padStart(4) // ' x'
9
10 //padStart()的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。
11 '1'.padStart(10, '0') // "0000000001"
12 '12'.padStart(10, '0') // "0000000012"
13 '123456'.padStart(10, '0') // "0000123456"
14
15 //另一个用途是提示字符串格式
16 '12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"
17 '09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"
```

## trimStart(), trimEnd()

### matchAll()

### replaceAll()

可以一次性替换所有匹配，但是如果 `searchValue` 是一个不带有 `g` 修饰符的正则表达式，`replaceAll()` 会报错。这一点跟 `replace()` 不同。

## ▼ replaceAll正则匹配必须带g修饰符

JavaScript |

```
1 // 不报错
2 'aabbcc'.replace(/b/, '_')
3
4 // 报错
5 'aabbcc'.replaceAll(/b/, '_')
```

`replaceAll()` 的第二个参数 `replacement` 是一个字符串，表示替换的文本，其中可以使用一些特殊字符串。

- `$&`：匹配的字符串。
- `$``：匹配结果前面的文本。
- `$'`：匹配结果后面的文本。
- `$n`：匹配成功的第 `n` 组内容，`n` 是从1开始的自然数。这个参数生效的前提是，第一个参数必须是正则表达式。
- `$$`：指代美元符号 `$`。

## ▼ replaceAll第二个参数

JavaScript |

```
1 // $& 表示匹配的字符串，即`b`本身
2 // 所以返回结果与原字符串一致
3 'abbc'.replaceAll('b', '$&')
4 // 'abbc'
5
6 // $` 表示匹配结果之前的字符串
7 // 对于第一个`b`，$` 指代`a`
8 // 对于第二个`b`，$` 指代`ab`
9 'abbc'.replaceAll('b', '$`')
10 // 'aaabc'
11
12 // $' 表示匹配结果之后的字符串
13 // 对于第一个`b`，$' 指代`bc`
14 // 对于第二个`b`，$' 指代`c`
15 'abbc'.replaceAll('b', '$`')
16 // 'abccc'
17
18 // $1 表示正则表达式的第一个组匹配，指代`ab`
19 // $2 表示正则表达式的第二个组匹配，指代`bc`
20 'abbc'.replaceAll(/(ab)(bc)/g, '$2$1')
21 // 'bcab'
22
23 // $$ 指代 $
24 'abc'.replaceAll('b', '$$')
25 // 'a$c'
```

## at()

**at()** 方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）

### ▼ at使用

JavaScript |

```
1 const str = 'hello';
2 str.at(1) // "e"
3 str.at(-1) // "o"
```

# 正则的扩展

## RegExp构造函数

### 两种构造函数方法

JavaScript |

```
1 //参数是字符串
2 var regex = new RegExp('xyz', 'i');
3 // 等价于
4 var regex = /xyz/i;
5
6 //参数是正则表达式
7 var regex = new RegExp(/xyz/i);
8 // 等价于
9 var regex = /xyz/i;
```

注：ES5不支持这种声明

```
var regex = new RegExp(/xyz/, 'i');
```

## 字符串的正则

在ES6之后，把字符串中用到的正则方法语言内部全部调用 `RegExp` 的实例方法，从而做到所有与正则相关的方法，全都定义在 `RegExp` 对象上。

- `String.prototype.match` 调用 `RegExp.prototype[Symbol.match]`
- `String.prototype.replace` 调用 `RegExp.prototype[Symbol.replace]`
- `String.prototype.search` 调用 `RegExp.prototype[Symbol.search]`
- `String.prototype.split` 调用 `RegExp.prototype[Symbol.split]`

## u修饰符

`u` 修饰符，含义为“Unicode 模式”，用来正确处理四个字节的 UTF-16 编码

## ▼ u修饰符处理四字节

JavaScript |

```
1  /^[^\uD83D/u.test('\uD83D\uDC2A') // false
2  /^[^\uD83D/.test('\uD83D\uDC2A') // true
```

## 点字符（.）

对于码点大于 `0xFFFF` 的 Unicode 字符，点字符不能识别，必须加上 `u` 修饰符

## ▼ 点字符无法识别四字节

JavaScript |

```
1  var s = '𠮷';
2
3  /^[.$/.test(s) // false
4  /^[.$/u.test(s) // true
```

## Unicode 字符表示法

## ▼ 将大括号识别为Unicode字符

JavaScript |

```
1  /\u{61}/.test('a') // false
2  /\u{61}/u.test('a') // true
3  /\u{20BB7}/u.test('𠮷') // true
```

如果不加 `u` 修饰符，正则表达式无法识别 `\u{61}` 这种表示法，只会认为这匹配 61 个连续的 `u`

## 量词

使用 `u` 修饰符后，所有量词都会正确识别码点大于 `0xFFFF` 的 Unicode 字符

## ▼ 通过u修饰符使量词正确识别

JavaScript |

```
1  /a{2}/.test('aa') // true
2  /a{2}/u.test('aa') // true
3  /𠮷{2}/.test('𠮷𠮷') // false
4  /𠮷{2}/u.test('𠮷𠮷') // true
```

## 预定义模式

## ▼ 预定义模式无法识别四字节

JavaScript |

```
1  /\^\$/.test('𠮷') // false  
2  /\^\$/u.test('𠮷') // true
```

\S 是预定义模式，匹配所有非空白字符。只有加了 u 修饰符，它才能正确匹配码点大于 0xFFFF 的 Unicode 字符

## RegExp.prototype.unicode 属性

### ▼ 是否设置了u修饰符

JavaScript |

```
1  const r1 = /hello/;  
2  const r2 = /hello/u;  
3  
4  r1.unicode // false  
5  r2.unicode // true
```

## y修饰符

“粘连”（sticky）修饰符，与g修饰符类似，但是下次匹配必须从开头开始就要匹配。

y 修饰符的设计本意，就是让头部匹配的标志^在全局匹配中都有效。

### ▼ y修饰符使用

JavaScript |

```
1  var s = 'aaa_aa_a';  
2  var r1 = /a+/g;  
3  var r2 = /a+/y;  
4  
5  r1.exec(s) // ["aaa"]  
6  r2.exec(s) // ["aaa"]  
7  
8  r1.exec(s) // ["aa"]  
9  r2.exec(s) // null
```

y 修饰符的一个应用，是从字符串提取 token（词元），y 修饰符确保了匹配之间不会有漏掉的字符。

## ▼ y修饰符检查字符

JavaScript |

```
1 const TOKEN_Y = /\s*(\+|[0-9]+)\s*/y;
2 const TOKEN_G = /\s*(\+|[0-9]+)\s*/g;
3
4 tokenize(TOKEN_Y, '3 + 4')
5 // [ '3', '+', '4' ]
6 tokenize(TOKEN_G, '3 + 4')
7 // [ '3', '+', '4' ]
8
9 - function tokenize(TOKEN_REGEX, str) {
10     let result = [];
11     let match;
12 -     while (match = TOKEN_REGEX.exec(str)) {
13         result.push(match[1]);
14     }
15     return result;
16 }
17
18 tokenize(TOKEN_Y, '3x + 4')
19 // [ '3' ]
20 tokenize(TOKEN_G, '3x + 4')
21 // [ '3', '+', '4' ]
```

上面代码中，`g` 修饰符会忽略非法字符，而 `y` 修饰符不会，这样就很容易发现错误。

## RegExp.prototype.sticky 属性

### ▼ 是否设置了y修饰符

JavaScript |

```
1 var r = /hello\d/y;
2 r.sticky // true
```

## RegExp.prototype.flags 属性

▼ 返回正则表达式的修饰符

JavaScript |

```
1 // ES5 的 source 属性
2 // 返回正则表达式的正文
3 /abc/ig.source
4 // "abc"
5
6 // ES6 的 flags 属性
7 // 返回正则表达式的修饰符
8 /abc/ig.flags
9 // 'gi'
```

# 数值的扩展

## 二进制和八进制表示法

ES6 提供了二进制和八进制数值的新的写法，分别用前缀 `0b` (或 `0B`) 和 `0o` (或 `0O`) 表示

### 二进制、八进制表示

JavaScript

```
1 0b111110111 === 503 // true
2 0o767 === 503 // true
3
4 //对于字符串转换要使用Number方法
5 Number('0b111') // 7
6 Number('0o10') // 8
```

# 数值分隔符

允许 JavaScript 的数值使用下划线（`_`）作为分隔符

### 数值分隔符

JavaScript

```
1 let budget = 1_000_000_000_000;
2 budget === 10 ** 12 // true
3
4 123_00 === 12_300 // true
5
6 // 二进制
7 0b1010_0001_1000_0101
8 // 十六进制
9 0xA0_B0_C0
```

数值分隔符有几个使用注意点。

- 不能放在数值的最前面 (leading) 或最后面 (trailing)。
- 不能两个或两个以上的分隔符连在一起。
- 小数点的前后不能有分隔符。
- 科学计数法里面，表示指数的 `e` 或 `E` 前后不能有分隔符。

## ▼ 分隔符错误使用

JavaScript |

```
1 // 全部报错
2 3_.141
3 3._141
4 1_e12
5 1e_12
6 123_456
7 _1464301
8 1464301_
9
10 0_b111111000
11 0b_111111000
```

# 新增Number方法

## Number.isFinite()

用来检查一个数值是否为有限的 (finite) , 即不是 `Infinity`

### ▼ isFinite使用

JavaScript |

```
1 Number.isFinite(15); // true
2 Number.isFinite(0.8); // true
3 Number.isFinite(NaN); // false
4 Number.isFinite(Infinity); // false
5 Number.isFinite(-Infinity); // false
6 Number.isFinite('foo'); // false
7 Number.isFinite('15'); // false
8 Number.isFinite(true); // false
```

## Number.isNaN()

用来检查一个值是否为 `Nan`

## ▼ isNaN使用

JavaScript |

```
1 Number.isNaN(NaN) // true
2 Number.isNaN(15) // false
3 Number.isNaN('15') // false
4 Number.isNaN(true) // false
5 Number.isNaN(9/NaN) // true
6 Number.isNaN('true' / 0) // true
7 Number.isNaN('true' / 'true') // true
```

它们与传统的全局方法 `isFinite()` 和 `isNaN()` 的区别在于，传统方法先调用 `Number()` 将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，`Number.isFinite()` 对于非数值一律返回 `false`，`Number.isNaN()` 只有对于 `Nan` 才返回 `true`，非 `Nan` 一律返回 `false`。

## ▼ Number方法的isFinite和isNaN与老方法区别

JavaScript |

```
1 isFinite(25) // true
2 isFinite("25") // true
3 Number.isFinite(25) // true
4 Number.isFinite("25") // false
5
6 isNaN(NaN) // true
7 isNaN("NaN") // true
8 Number.isNaN(NaN) // true
9 Number.isNaN("NaN") // false
10 Number.isNaN(1) // false
```

## Number.parseInt()

## Number.parseFloat()

ES6 将全局方法 `parseInt()` 和 `parseFloat()`，移植到 `Number` 对象上面，行为完全保持不变。

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

## ▼ parseInt()和parseFloat()

JavaScript

```
1 // ES5的写法
2 parseInt('12.34') // 12
3 parseFloat('123.45#') // 123.45
4
5 // ES6的写法
6 Number.parseInt('12.34') // 12
7 Number.parseFloat('123.45#') // 123.45
8
9 Number.parseInt === parseInt // true
10 Number.parseFloat === parseFloat // true
```

## Number.isInteger()

用来判断一个数值是否为整数，但是由于JS是64位双精度浮点数，所以超出该位数就会识别不到，还有就是超过最小值也识别不到

## ▼

JavaScript

```
1 Number.isInteger(25) // true
2 Number.isInteger(25.1) // false
3
4 //小数的精度达到了小数点后16个十进制位，转成二进制位超过了53个二进制位，导致最后的那个2被丢弃了
5 Number.isInteger(3.0000000000000002) // true
6
7 //Number.MIN_VALUE (5E-324)
8 //5E-325由于值太小，会被自动转为0
9 Number.isInteger(5E-324) // false
10 Number.isInteger(5E-325) // true
```

## Number.EPSILON

常量，表示 1 与大于 1 的最小浮点数之间的差，引入的作用是可以用来处理浮点数的精度问题

## Number.EPSILON使用

JavaScript |

```
1 Number.EPSILON === Math.pow(2, -52)
2
3 function withinErrorMargin (left, right) {
4     return Math.abs(left - right) < Number.EPSILON;
5 }
6 0.1 + 0.2 === 0.3 // false
7 withinErrorMargin(0.1 + 0.2, 0.3) // true
```

# Math对象扩展

## Math.trunc()

去除一个数的小数部分，返回整数部分

### Math.trunc

JavaScript |

```
1 Math.trunc(4.1) // 4
2 Math.trunc(-4.9) // -4
3 Math.trunc(-0.1234) // -0
4
5 Math.trunc('123.456') // 123
6 Math.trunc(true) // 1
7 Math.trunc(false) // 0
8 Math.trunc(null) // 0
9
10 Math.trunc(NaN); // NaN
11 Math.trunc('foo'); // NaN
12 Math.trunc(); // NaN
13 Math.trunc(undefined) // NaN
```

## Math.sign()

用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

它会返回五种值。

- 参数为正数，返回 `+1`；
- 参数为负数，返回 `-1`；
- 参数为 0，返回 `0`；

- 参数为-0，返回 `-0`；
- 其他值，返回 `NaN`。

```
▼ Math.sign()  
JavaScript |  
  
1 Math.sign(-5) // -1  
2 Math.sign(5) // +1  
3 Math.sign(0) // +0  
4 Math.sign(-0) // -0  
5 Math.sign(NaN) // NaN  
6  
7 Math.sign('') // 0  
8 Math.sign(true) // +1  
9 Math.sign(false) // 0  
10 Math.sign(null) // 0  
11 Math.sign('9') // +1  
12 Math.sign('foo') // NaN  
13 Math.sign() // NaN  
14 Math.sign(undefined) // NaN
```

## Math.cbrt()

返回一个数的立方根

## Math.clz32()

将参数转为 32 位无符号整数的形式，然后返回这个 32 位值里面有多少个前导 0

## Math.imul()

返回两个数以 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

## Math.fround()

返回一个数的32位单精度浮点数形式

## Math.hypot()

返回所有参数的平方和的平方根

▼ 其他Math方法

JavaScript

```
1 Math.cbrt(9) // 3
2 Math.cbrt('8') // 2
3 Math.cbrt('hello') // NaN
4
5 Math.clz32(1000) // 22
6
7 Math.imul(2, 4) // 8
8 Math.imul(-1, 8) // -8
9 Math.imul(-2, -2) // 4
10
11 Math.fround(0) // 0
12 Math.fround(1) // 1
13 Math.fround(2 ** 24 - 1) // 16777215
14 Math.fround(2 ** 24) // 16777216
15 Math.fround(2 ** 24 + 1) // 16777216 (精度丢失)
16
17 Math.hypot(3, 4); // 5
```

## 对数方法

### Math.expm1()

`Math.expm1(x)` 返回  $e^x - 1$ , 即 `Math.exp(x) - 1`

### Math.log1p()

`Math.log1p(x)` 方法返回  $1 + x$  的自然对数, 即 `Math.log(1 + x)`。如果 `x` 小于-1, 返回 `NaN`

### Math.log10()

`Math.log10(x)` 返回以 10 为底的 `x` 的对数。如果 `x` 小于 0, 则返回 `NaN`

### Math.log2()

**Math.log2(x)** 返回以 2 为底的  $x$  的对数。如果  $x$  小于 0，则返回 **NaN**

▼ 对数方法

JavaScript

```
1 Math.expm1(-1) // -0.6321205588285577
2 Math.expm1(0) // 0
3 Math.expm1(1) // 1.718281828459045
4
5 Math.log1p(1) // 0.6931471805599453
6 Math.log1p(0) // 0
7 Math.log1p(-1) // -Infinity
8 Math.log1p(-2) // NaN
9
10 Math.log10(2) // 0.3010299956639812
11 Math.log10(1) // 0
12 Math.log10(0) // -Infinity
13 Math.log10(-2) // NaN
14 Math.log10(100000) // 5
15
16 Math.log2(2) // 1
17 Math.log2(1) // 0
18 Math.log2(0) // -Infinity
19 Math.log2(-2) // NaN
20 Math.log2(1024) // 10
```

# 函数的扩展

## 函数参数默认值

### 基本用法

#### 函数默认参数使用

JavaScript

```
1 function Point(x = 0, y = 0) {
2     this.x = x;
3     this.y = y;
4 }
5
6 const p = new Point();
7 p // { x: 0, y: 0 }
8
9
10 //不能有同名参数
11 // 不报错
12 function foo(x, x, y) {
13     ...
14 }
15
16 // 报错
17 function foo(x, x, y = 1) {
18     ...
19 }
20 // SyntaxError: Duplicate parameter name not allowed in this context
```

### 与解构赋值结合使用

## 与解构赋值结合使用

JavaScript |

```
1 function foo({x, y = 5}) {  
2   console.log(x, y);  
3 }  
4  
5 foo({}) // undefined 5  
6 foo({x: 1}) // 1 5  
7 foo({x: 1, y: 2}) // 1 2  
8 foo() // TypeError: Cannot read property 'x' of undefined
```

上面代码只使用了对象的解构赋值默认值，没有使用函数参数的默认值。只有当函数 `foo()` 的参数是一个对象时，变量 `x` 和 `y` 才会通过解构赋值生成。如果函数 `foo()` 调用时没提供参数，变量 `x` 和 `y` 就不会生成，从而报错。通过提供函数参数的默认值，就可以避免这种情况。

## 与解构赋值结合使用

JavaScript |

```
1 function foo({x, y = 5} = {}) {  
2   console.log(x, y);  
3 }  
4  
5 foo() // undefined 5
```

## 函数参数的默认值生效以后，参数解构赋值依然会进行

JavaScript |

```
1 function f({ a, b = 'world' } = { a: 'hello' }) {  
2   console.log(b);  
3 }  
4  
5 f() // world  
6
```

函数 `f()` 调用时没有参数，所以参数默认值 `{ a: 'hello' }` 生效，然后再对这个默认值进行解构赋值，从而触发参数变量 `b` 的默认值生效。

```
1 // 写法一
2 function m1({x = 0, y = 0} = {}) {
3     return [x, y];
4 }
5
6 // 写法二
7 function m2({x, y} = {x: 0, y: 0}) {
8     return [x, y];
9 }
10
11 // 函数没有参数的情况
12 m1() // [0, 0]
13 m2() // [0, 0]
14
15 // x 和 y 都有值的情况
16 m1({x: 3, y: 8}) // [3, 8]
17 m2({x: 3, y: 8}) // [3, 8]
18
19 // x 有值, y 无值的情况
20 m1({x: 3}) // [3, 0]
21 m2({x: 3}) // [3, undefined]
22
23 // x 和 y 都无值的情况
24 m1({}) // [0, 0];
25 m2({}) // [undefined, undefined]
26
27 m1({z: 3}) // [0, 0]
28 m2({z: 3}) // [undefined, undefined]
```

## 参数默认值位置

应该是函数的尾参数。因为这样比较容易看出来，到底省略了哪些参数。如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

```

1 ▾ function f(x, y = 5, z) {
2     return [x, y, z];
3 }
4
5 f() // [undefined, 5, undefined]
6 f(1) // [1, 5, undefined]
7 f(1, ,2) // 报错
8 f(1, undefined, 2) // [1, 5, 2]

```

## 函数的length属性

函数指定参数默认值后，length属性会失效

### length属性失效

```

1 (function (a) {}).length // 1
2 (function (a = 5) {}).length // 0
3 (function (a, b = 1, c) {}).length // 1
4 (function (a, b, c = 5) {}).length // 2

```

## 作用域

函数参数默认值会有单独的作用域

```

1 let x = 1;
2
3 ▾ function f(y = x) {
4     let x = 2;
5     console.log(y);
6 }
7
8 f() // 1

```

## 函数参数作用域例子

JavaScript

```
1 var x = 1;
2 function foo(x, y = function() { x = 2; }) {
3     var x = 3;
4     y();
5     console.log(x);
6 }
7
8 foo() // 3
9 x // 1
```

上面代码中，函数 `foo` 的参数形成一个单独作用域。这个作用域里面，首先声明了变量 `x`，然后声明了变量 `y`，`y` 的默认值是一个匿名函数。这个匿名函数内部的变量 `x`，指向同一个作用域的第一个参数 `x`。函数 `foo` 内部又声明了一个内部变量 `x`，该变量与第一个参数 `x` 由于不是同一个作用域，所以不是同一个变量，因此执行 `y` 后，内部变量 `x` 和外部全局变量 `x` 的值都没变。

如果将 `var x = 3` 的 `var` 去除，函数 `foo` 的内部变量 `x` 就指向第一个参数 `x`，与匿名函数内部的 `x` 是一致的，所以最后输出的就是 `2`，而外层的全局变量 `x` 依然不受影响。

## 修改后

JavaScript

```
1 var x = 1;
2 function foo(x, y = function() { x = 2; }) {
3     x = 3;
4     y();
5     console.log(x);
6 }
7
8 foo() // 2
9 x // 1
```

# 应用

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

## ▼ 定义必传参数

JavaScript |

```
1  function throwIfMissing() {
2      throw new Error('Missing parameter');
3  }
4
5  function foo(mustBeProvided = throwIfMissing()) {
6      return mustBeProvided;
7  }
8
9  foo()
10 // Error: Missing parameter
```

# rest参数

用于获取函数的多余参数，这样就不需要使用 `arguments` 对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

注意，rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

函数的 `length` 属性，不包括 rest 参数。

## ▼ rest参数使用

JavaScript |

```
1  function add(...values) {
2      let sum = 0;
3
4  for (var val of values) {
5      sum += val;
6  }
7
8  return sum;
9 }
10
11 add(2, 5, 3) // 10
12
13 // 报错
14 function f(a, ...b, c) {
15     // ...
16 }
17
18 (function(a) {}).length // 1
19 (function(...a) {}).length // 0
20 (function(a, ...b) {}).length // 1
```

## name属性

JavaScript |

```
1 var f = function () {};
2
3 // ES5
4 f.name // ""
5
6 // ES6
7 f.name // "f"
8
9 //Function构造函数返回的函数实例，name属性的值为anonymous
10 (new Function).name // "anonymous"
11
12
13 //bind返回的函数，name属性值会加上bound前缀
14 function foo() {};
15 foo.bind({}).name // "bound foo"
16
17 (function(){}).bind({}).name // "bound "
```

## 箭头函数

- 箭头函数没有自己的 `this` 对象。
- 不可以当作构造函数，也就是说，不可以对箭头函数使用 `new` 命令，否则会抛出一个错误。
- 不可以使用 `arguments` 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
- 不可以使用 `yield` 命令，因此箭头函数不能用作 Generator 函数。

▼ 箭头函数没有自己的this对象

JavaScript |

```
1 function foo() {
2   setTimeout(() => {
3     console.log('id:', this.id);
4   }, 100);
5 }
6
7 var id = 21;
8
9 foo.call({ id: 42 });
10 // id: 42
```

如果是普通函数，执行时 `this` 应该指向全局对象 `window`，这时应该输出 `21`。但是，箭头函数导致 `this` 总是指向函数定义生效时所在的对象（本例是 `{id: 42}`），所以打印出来的是 `42`。

## 尾调用优化

### 函数参数尾逗号

### catch参数省略

# 数组的扩展

## 扩展运算符

扩展运算符主要用于将一个数组转为用逗号分隔的参数序列

### 扩展运算符

JavaScript

```
1 function fn(...args){  
2     console.log(...args);  
3     console.log(args);  
4 }  
5 fn(1,2,3)  
6 //1 2 3  
7 // [1,2,3]  
8 fn([1,2,3])  
9 // [1,2,3]  
10 // [ [1,2,3] ]
```

## 使用场景

### 函数参数

#### ▼

JavaScript

```
1 function add(x, y) {  
2     return x + y;  
3 }  
4  
5 const numbers = [4, 38];  
6 add(...numbers) // 42
```

### 替代apply方法

## 将数组转为函数参数

JavaScript |

```
1 // ES5 的写法
2 function f(x, y, z) {
3     ...
4 }
5 var args = [0, 1, 2];
6 f.apply(null, args);
7
8 // ES6 的写法
9 function f(x, y, z) {
10    ...
11 }
12 let args = [0, 1, 2];
13 f(...args);
```

## push数组

JavaScript |

```
1 // ES5 的写法
2 var arr1 = [0, 1, 2];
3 var arr2 = [3, 4, 5];
4 Array.prototype.push.apply(arr1, arr2);
5
6 // ES6 的写法
7 let arr1 = [0, 1, 2];
8 let arr2 = [3, 4, 5];
9 arr1.push(...arr2);
```

# 复制数组

通过扩展运算符实现浅拷贝

```
1 //ES5
2 var a1 = [1, 2];
3 var a2 = [].concat(a1);
4 //ES6
5 const a1 = [1, 2];
6 const a2 = [...a1];
```

# 合并数组

```

1 const arr1 = ['a', 'b'];
2 const arr2 = ['c'];
3 const arr3 = ['d', 'e'];
4
5 // ES5 的合并数组
6 arr1.concat(arr2, arr3);
7 // [ 'a', 'b', 'c', 'd', 'e' ]
8
9 // ES6 的合并数组
10 [...arr1, ...arr2, ...arr3]
11 // [ 'a', 'b', 'c', 'd', 'e' ]

```

## 与解构赋值结合

```

1 // ES5
2 a = list[0], rest = list.slice(1)
3
4 // ES6
5 [a, ...rest] = list
6
7 //注：扩展运算符用于赋值只能放在参数最后一位
8 const [...butLast, last] = [1, 2, 3, 4, 5];
9 // 报错
10
11 const [first, ...middle, last] = [1, 2, 3, 4, 5];
12 // 报错

```

## 字符串转数组

```

1 [...'hello']
2 // [ "h", "e", "l", "l", "o" ]

```

因为字符串有`Iterator`接口，所以可被转化。

## 可用于任何实现`Iterator`接口的对象

```

1 let nodeList = document.querySelectorAll('div');
2 //NodeList是一个类数组
3 let array = [...nodeList];
4
5 Number.prototype[Symbol.iterator] = function*() {
6     let i = 0;
7     let num = this.valueOf();
8     while (i < num) {
9         yield i++;
10    }
11 }
12
13 console.log([...5]) // [0, 1, 2, 3, 4]

```

## 与Map、Set、Generator函数结合

扩展运算符内部调用的是数据结构的 Iterator 接口，因此只要具有 Iterator 接口的对象，都可以使用扩展运算符

### 与Map结合使用

```

1 let map = new Map([
2     [1, 'one'],
3     [2, 'two'],
4     [3, 'three'],
5 ]);
6
7 let arr = [...map.keys()]; // [1, 2, 3]

```

### 与Generator结合使用

```

1 const go = function*(){
2     yield 1;
3     yield 2;
4     yield 3;
5 };
6
7 [...go()] // [1, 2, 3]

```

## Array.form()

用于将两类对象转为数组，一类是类数组对象；一类是可遍历（Iterable）对象，比如Map和Set。只要是部署了 `Iterator` 接口的数据结构，`Array.from()` 都能将其转为数组。

#### ▼ 转换类数组

JavaScript |

```
1 let arrayLike = {  
2     '0': 'a',  
3     '1': 'b',  
4     '2': 'c',  
5     length: 3  
6 };  
7  
8 // ES5 的写法  
9 var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']  
10  
11 // ES6 的写法  
12 let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

在实际应用用一般用于转换DOM操作返回的`NodeList`集合以及函数内部 `arguments` 对象。

对于还没部署该方法的环境，可以使用 `Array.prototype.slice()` 代替。

`Array.from()` 第二个参数类似于map，对每个元素进行处理，将处理后的值放入返回的数组。

#### ▼ `Array.from()`第二个参数

JavaScript |

```
1 Array.from(arrayLike, x => x * x);  
2 // 等同于  
3 Array.from(arrayLike).map(x => x * x);  
4  
5 Array.from([1, 2, 3], (x) => x * x)  
6 // [1, 4, 9]
```

`Array.from()` 和扩展运算符都可以将某些数据结构转为数组，但是扩展运算符背后调用的是遍历器接口（`Symbol.iterator`），如果一个对象没有部署这个接口，就无法转换。`Array.from()` 方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有 `length` 属性。因此，任何有 `length` 属性的对象，都可以通过 `Array.from()` 方法转为数组，而此时扩展运算符就无法转换。

```
1 Array.from({ length: 3 });  
2 // [ undefined, undefined, undefined ]
```

JavaScript |

# Array.of()

用于将一组值转为数组

```
1 Array.of(3, 11, 8) // [3,11,8]
2 Array.of(3) // [3]
3 Array.of(3).length // 1
```

JavaScript

主要为了弥补 `Array()` 的不足，因为参数不同 `Array()` 行为有差异。

```
1 Array() // []
2 Array(3) // [, , ,]
3 Array(3, 11, 8) // [3, 11, 8]
```

JavaScript

`Array.of()` 方法可以用下面的代码模拟实现。

```
1 function ArrayOf(){
2     return [].slice.call(arguments);
3 }
```

JavaScript

## 实例方法

### copyWithin()

在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

```
1 Array.prototype.copyWithin(target, start = 0, end = this.length)
2
```

JavaScript

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
- start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。

- end (可选) : 到该位置前停止读取数据, 默认等于数组长度。如果为负值, 表示从末尾开始计算。

```
1 [1, 2, 3, 4, 5, 6].copyWithin(0, 3)
2 // [4, 5, 6, 4, 5, 6]
```

## find(), findIndex(), findLast(), findLastIndex()

用于找出第一个符合条件的数组成员/成员位置

```
1 [1, 5, 10, 15].find(function(value, index, arr) {
2   return value > 9;
3 }) // 10
4
5 [1, 5, 10, 15].findIndex(function(value, index, arr) {
6   return value > 9;
7 }) // 2
```

第一个参数是执行的回调函数; 可以传入第二个参数作为回调函数绑定的this对象

```
1 function f(v){
2   return v > this.age;
3 }
4 let person = {name: 'John', age: 20};
5 [10, 12, 26, 15].find(f, person); // 26
```

另外, 也弥补了 `indexof` 无法发现 `NAN` 的不足。

```
1 [NaN].indexof(NaN)
2 // -1
3
4 [NaN].findIndex(y => Object.is(NaN, y))
5 // 0
```

## fill()

使用给定值，填充一个数组。

```
1  ['a', 'b', 'c'].fill(7)
2  // [7, 7, 7]
3
4  new Array(3).fill(7)
5  // [7, 7, 7]
```

JavaScript

可以通过第二个和第三个参数指定位置填充

```
1  ['a', 'b', 'c'].fill(7, 1, 2)
2  // ['a', 7, 'c']
```

JavaScript

注意如果填充的是对象，那这些对象都是一个引用地址

```
1  let arr = new Array(3).fill({name: "Mike"});
2  arr[0].name = "Ben";
3  arr
4  // [{name: "Ben"}, {name: "Ben"}, {name: "Ben"}]
5
6  let arr = new Array(3).fill([]);
7  arr[0].push(5);
8  arr
9  // [[5], [5], [5]]
```

JavaScript

## entries(), keys() 和 values()

这几个方法返回的是遍历器对象。可以用 `for...of` 遍历。

```

1 for (let index of ['a', 'b'].keys()) {
2   console.log(index);
3 }
4 // 0
5 // 1
6
7 for (let elem of ['a', 'b'].values()) {
8   console.log(elem);
9 }
10 // 'a'
11 // 'b'
12
13 for (let [index, elem] of ['a', 'b'].entries()) {
14   console.log(index, elem);
15 }
16 // 0 "a"
17 // 1 "b"

```

也可以手动遍历

```

1 let letter = ['a', 'b', 'c'];
2 let entries = letter.entries(); //Array Iterator {}
3 console.log(entries.next().value); // [0, 'a']
4 console.log(entries.next().value); // [1, 'b']
5 console.log(entries.next().value); // [2, 'c']

```

## includes()

返回一个布尔值，表示数组是否包含给定的值。

```

1 [1, 2, 3].includes(2)      // true
2 [1, 2, 3].includes(4)      // false
3 [1, 2, NaN].includes(NaN) // true

```

可以用第二个参数来指定起始位置

- 如果参数为负值，则表示倒数位置，如果该负值（绝对值）大于数组长度，则从0开始

```

1 [1, 2, 3].includes(3, 3); // false
2 [1, 2, 3].includes(3, -1); // true

```

没有该方法之前，我们通常使用数组的 `indexOf` 方法，检查是否包含某个值。但是 `indexOf` 有两个缺点

- 不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观
- 它内部使用严格相等运算符（`==`）进行判断，这会导致对NaN的误判

```

1 [NaN].indexOf(NaN)
2 // -1
3
4 [NaN].includes(NaN)
5 // true

```

## flat(), flatMap()

`flat` 用于拉平数组，默认拉平一层，可以通过参数指定

```

1 [1, 2, [3, [4, 5]]].flat()
2 // [1, 2, 3, [4, 5]]
3
4 [1, 2, [3, [4, 5]]].flat(2)
5 // [1, 2, 3, 4, 5]
6
7 //如果不管有多少层嵌套，都要转成一维数组，可以用Infinity关键字作为参数
8 [1, [2, [3]]].flat(Infinity)
9 // [1, 2, 3]
10
11 //flat会跳过空位
12 [1, 2, , 4, 5].flat()
13 // [1, 2, 4, 5]

```

`flatMap()` 方法对原数组的每个成员执行一个函数（相当于执行 `Array.prototype.map()`），然后对返回值组成的数组执行 `flat()` 方法。该方法返回一个新数组，不改变原数组。

```

1 // 相当于 [[2, 4], [3, 6], [4, 8]].flat()
2 [2, 3, 4].flatMap((x) => [x, x * 2])
3 // [2, 4, 3, 6, 4, 8]
4
5 //flatMap()只能展开一层数组
6 // 相当于 [[[2]], [[4]], [[6]], [[8]]].flat()
7 [1, 2, 3, 4].flatMap(x => [[x * 2]])
8 // [[2], [4], [6], [8]]

```

## at()

JavaScript不支持数组负索引，因为对象也能用方括号运算符，`obj [-1]` 引用的是键名为字符串-1的键

所以增加了 `at()` 方法，接受一个整数作为参数，返回对应位置的成员，并支持负索引。这个方法不仅可用于数组，也可用于字符串和类型数组（TypedArray）。

```

1 const arr = [5, 12, 8, 130, 44];
2 arr.at(2) // 8
3 arr.at(-2) // 130
4
5 const sentence = 'This is a sample sentence';
6
7 sentence.at(0); // 'T'
8 sentence.at(-1); // 'e'
9
10 sentence.at(-100) // undefined
11 sentence.at(100) // undefined

```

## toReversed(), toSorted(), toSpliced(), with()

`push()`、`pop()`、`shift()`、`unshift()` 等方法会改变原数组，所以增加新方法

- `toReversed()` 对应 `reverse()`，用来颠倒数组成员的位置。
- `toSorted()` 对应 `sort()`，用来对数组成员排序。
- `toSpliced()` 对应 `splice()`，用来在指定位置，删除指定数量的成员，并插入新成员。

- `with(index, value)` 对应 `splice(index, 1, value)`，用来将指定位置的成员替换为新的值。

上面是这四个新方法对应的原有方法，含义和用法完全一样，唯一不同的是不会改变原数组，而是返回原数组操作后的拷贝。

```
1 const sequence = [1, 2, 3];
2 sequence.toReversed() // [3, 2, 1]
3 sequence // [1, 2, 3]
4
5 const outOfOrder = [3, 1, 2];
6 outOfOrder.toSorted() // [1, 2, 3]
7 outOfOrder // [3, 1, 2]
8
9 const array = [1, 2, 3, 4];
10 array.toSpliced(1, 2, 5, 6, 7) // [1, 5, 6, 7, 4]
11 array // [1, 2, 3, 4]
12
13 const correctionNeeded = [1, 1, 3];
14 correctionNeeded.with(1, 2) // [1, 2, 3]
15 correctionNeeded // [1, 1, 3]
```

## group(), groupToMap() 【提案阶段】

用于数组成员分组

```
1 const array = [1, 2, 3, 4, 5];
2
3 array.group((num, index, array) => {
4   return num % 2 === 0 ? 'even': 'odd';
5 });
6 // { odd: [1, 3, 5], even: [2, 4] }
7
8 [6.1, 4.2, 6.3].group(Math.floor)
9 // { '4': [4.2], '6': [6.1, 6.3] }
```

`groupToMap()` 的作用和用法与 `group()` 完全一致，唯一的区别是返回值是一个 `Map` 结构，而不是对象。

```
1 const array = [1, 2, 3, 4, 5];
2
3 const odd  = { odd: true };
4 const even = { even: true };
5 array.groupToMap((num, index, array) => {
6   return num % 2 === 0 ? even: odd;
7 });
8 // Map { {odd: true}: [1, 3, 5], {even: true}: [2, 4] }
```

总之，按照字符串分组就使用 `group()`，按照对象分组就使用 `groupToMap()`。

# 对象的扩展

## 属性的简洁表示

ES6允许在大括号里直接写入变量和函数作为对象的属性和方法。

### 属性简写

JavaScript

```
1 //直接写入变量
2 const foo = 'bar';
3 const baz = {foo};
4 baz // {foo: "bar"}
5
6 // 等同于
7 const baz = {foo: foo};
8
9 //直接写入函数
10 const baz = {function(){}};
11 baz
12 // { function: [Function: function] }
```

除了属性，方法也可以简写

### 方法简写

JavaScript

```
1 const o = {
2   method() {
3     return "Hello!";
4   }
5 };
6
7 // 等同于
8
9 const o = {
10   method: function() {
11     return "Hello!";
12   }
13};
```

一般常用于模块输出、属性赋值器（getter、setter）

## CommonJS模块输出

JavaScript |

```
1 module.exports = { getItem,.setItem, clear };
2 // 等同于
3 module.exports = {
4   getItem: getItem,
5   setItem: setItem,
6   clear: clear
7 };
```

这里需要注意**简写对象方法没有prototype**, 所以使用new会报错。

```
1 const obj = {
2   f() {
3     this.foo = 'bar';
4   }
5 };
6
7 new obj.f() // 报错
8 console.log(obj.f.prototype) //undefined
9
10 const obj = {
11   f:function() {
12     this.foo = 'bar';
13   }
14 };
15
16 new obj.f() // 正确
17 console.log(obj.f.prototype) //{}
```

## 属性名表达式

JavaScript中定义对象属性有两种方式

### 定义对象属性

JavaScript |

```
1 // 方法一
2 obj.foo = true;
3
4 // 方法二
5 obj['a' + 'bc'] = 123;
```

但是如果使用字面量方式定义对象（大括号），ES5时只能用第一种，ES6允许字面量定义时，可以用表达式作为对象属性名

▼ 表达式定义对象属性名

JavaScript |

```
1 let propKey = 'foo';
2
3 //ES6
4 let obj = {
5   [propKey]: true,
6   ['a' + 'bc']: 123
7 };
```

方法名也可以

▼ 表达式定义对象方法名

JavaScript |

```
1 let obj = {
2   ['h' + 'ello']() {
3     return 'hi';
4   }
5 };
6
7 obj.hello() // hi
```

注意属性名表达式和简洁写法不能同时用

```
1 // 报错
2 const foo = 'bar';
3 const bar = 'abc';
4 const baz = { [foo] };
5
6 // 正确
7 const foo = 'bar';
8 const baz = { [foo]: 'abc' };
```

## 方法的name属性

函数的name属性，返回函数名。对象方法也是函数，因此也有name属性。

```
1 const person = {
2   sayName() {
3     console.log('hello!');
4   },
5 }
6
7 person.sayName.name // "sayName"
```

JavaScript |

如果对象的方法使用了取值函数（**getter**）和存值函数（**setter**），则**name**属性不是在该方法上面，而是该方法的属性的描述对象的**get**和**set**属性上面，返回值是方法名前加上**get**和**set**。

```
1 const obj = {
2   get foo() {},
3   set foo(x) {}
4 };
5
6 obj.foo.name
7 // TypeError: Cannot read property 'name' of undefined
8
9 const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');
10
11 descriptor.get.name // "get foo"
12 descriptor.set.name // "set foo"
```

JavaScript |

有两种特殊情况：

- **bind**方法创造的函数，**name**属性返回**bound**加上原函数的名字
- **Function**构造函数创造的函数，**name**属性返回**anonymous**

```
1 (new Function()).name // "anonymous"
2
3 var doSomething = function() {
4   // ...
5 };
6 doSomething.bind().name // "bound doSomething"
```

JavaScript |

## 属性的可枚举性和遍历

对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。

`Object.getOwnPropertyDescriptor`方法可以获取该属性的描述对象。

描述对象的`enumerable`属性，称为“可枚举性”，如果该属性为`false`，就表示某些操作会忽略当前属性。

目前，有四个操作会忽略`enumerable`为`false`的属性。

- `for...in`循环（ES5）：只遍历对象自身的和继承的可枚举的属性。
- `Object.keys()`（ES5）：返回对象自身的所有可枚举的属性的键名。
- `JSON.stringify()`（ES5）：只串行化对象自身的可枚举的属性。
- `Object.assign()`（ES6）：忽略`enumerable`为`false`的属性，只拷贝对象自身的可枚举的属性。

另外，ES6 规定，所有 `Class` 的原型的方法都是不可枚举的。

## 属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

(1) `for...in`

`for...in`循环遍历对象自身的和继承的可枚举属性（不含 `Symbol` 属性）。

(2) `Object.keys(obj)`

`Object.keys`返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 `Symbol` 属性）的键名。

(3) `Object.getOwnPropertyNames(obj)`

`Object.getOwnPropertyNames`返回一个数组，包含对象自身的所有属性（不含 `Symbol` 属性，但是包括不可枚举属性）的键名。

(4) `Object.getOwnPropertySymbols(obj)`

`Object.getOwnPropertySymbols`返回一个数组，包含对象自身的所有 `Symbol` 属性的键名。

(5) `Reflect.ownKeys(obj)`

`Reflect.ownKeys`返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 `Symbol` 或字符串，也不管是否可枚举。

以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

- 首先遍历所有数值键，按照数值升序排列。

- 其次遍历所有字符串键，按照加入时间升序排列。
- 最后遍历所有 Symbol 键，按照加入时间升序排列。

## super关键字

我们知道，`this`关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字`super`，指向当前对象的原型对象。

```
1 const proto = {  
2   foo: 'hello'  
3 };  
4  
5 const obj = {  
6   foo: 'world',  
7   find() {  
8     return super.foo;  
9   }  
10 };  
11  
12 Object.setPrototypeOf(obj, proto);  
13 obj.find() // "hello"
```

JavaScript

注意`super`关键字表示原型对象时，只能用在对象的方法中（简写对象方法）

```

1 // 报错
2 const obj = {
3   foo: super.foo
4 }
5
6 // 报错
7 const obj = {
8   foo: () => super.foo
9 }
10
11 // 报错
12 const obj = {
13   foo: function () {
14     return super.foo
15   }
16 }

```

以上几个报错

- 第一种方法是因为`super`用在了属性里面
- 第二种和第三种是因为`super`是在一个函数中，然后函数被赋值嘿`foo`属性。

JavaScript 引擎内部，`super.foo` 等同于 `Object.getPrototypeOf(this).foo` (属性) 或 `Object.getPrototypeOf(this).foo.call(this)` (方法)。

```

1 const proto = {
2   x: 'hello',
3   foo() {
4     console.log(this.x);
5   },
6 };
7
8 const obj = {
9   x: 'world',
10  foo() {
11    super.foo();
12  }
13 }
14
15 Object.setPrototypeOf(obj, proto);
16
17 obj.foo() // "world"

```

上面代码中，`super.foo`指向原型对象`proto`的`foo`方法，但是绑定的`this`却还是当前对象`obj`，因此输出的就是`world`。

## 对象的扩展运算符

### 解构赋值

对象的解构赋值用于从一个对象取值，相当于将目标**对象自身**的所有**可遍历的（enumerable）**、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

```
1 let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
2 x // 1
3 y // 2
4 z // { a: 3, b: 4 }
5
6 let { ...z } = null; // 运行时错误
7 let { ...z } = undefined; // 运行时错误
8
9 //解构赋值必须是第一个参数
10 let { ...x, y, z } = someObject; // 句法错误
11 let { x, ...y, ...z } = someObject; // 句法错误
```

扩展运算符的解构赋值，不能复制继承自原型对象的额属性。

```
1 let o1 = { a: 1 };
2 let o2 = { b: 2 };
3 o2.__proto__ = o1;
4 let { ...o3 } = o2;
5 o3 // { b: 2 }
6 o3.a // undefined
7
8 let { a } = o2;
9 a //1
```

所以这里就需要注意 `Object.create` 这个方法

```

1 const o = Object.create({ x: 1, y: 2 });
2 o.z = 3;
3
4 let { x, ...newObj } = o;
5 let { y, z } = newObj;
6 x // 1
7 y // undefined
8 z // 3

```

变量x是单纯的解构赋值，所以可以读取对象o继承的属性，但是newObj是扩展运算符解构，所以只能取到对象o自身的属性。

## 扩展运算符

对象的扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

```

1 let z = { a: 3, b: 4 };
2 let n = { ...z };
3 n // { a: 3, b: 4 }

```

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```

1 let foo = { ...['a', 'b', 'c'] };
2 foo
3 // {0: "a", 1: "b", 2: "c"}

```

如果扩展运算符后面不是对象，则会自动将其转为对象。

```

1 // 等同于 {...Object(1)}
2 {...1} // {}
3
4 // 等同于 {...Object(true)}
5 {...true} // {}
6
7 // 等同于 {...Object(undefined)}
8 {...undefined} // {}
9
10 // 等同于 {...Object(null)}
11 {...null} // {}
12
13 {...'hello'}
14 // {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}

```

对象的扩展运算符等同于使用 `Object.assign` 方法

```

1 let aClone = { ...a };
2 // 等同于
3 let aClone = Object.assign({}, a);

```

所以扩展运算符日常用于

## 合并对象

```

1 let ab = { ...a, ...b };
2 // 等同于
3 let ab = Object.assign({}, a, b);

```

## 修改现有对象部分属性

如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```

1 let newVersion = {
2   ...previousVersion,
3   name: 'New Name' // Override the name property
4 };

```

## AggregateError 错误对象

AggregateError 在一个错误对象里面，封装了多个错误。如果某个单一操作，同时引发了多个错误，需要同时抛出这些错误，那么就可以抛出一个 AggregateError 错误对象，把各种错误都放在这个对象里面。

`AggregateError()`构造函数可以接受两个参数。

- `errors`: 数组，它的每个成员都是一个错误对象。该参数是必须的。
- `message`: 字符串，表示 `AggregateError` 抛出时的提示信息。该参数是可选的。

`AggregateError` 的实例对象有三个属性。

- `name`: 错误名称，默認為“`AggregateError`”。
- `message`: 错误的提示信息。
- `errors`: 数组，每个成员都是一个错误对象。

```

1 try {
2   throw new AggregateError([
3     new Error('ERROR_11112'),
4     new TypeError('First name must be a string'),
5     ], 'Hello');
6 } catch (e) {
7   console.log(e instanceof AggregateError); // true
8   console.log(e.message); // "Hello"
9   console.log(e.name); // "AggregateError"
10  console.log(e.errors);
11    Error:"First name must be a string"]

```

## Error对象的cause属性

Error 对象用来表示代码运行时的异常情况，但是从这个对象拿到的上下文信息，有时很难解读，也不够充分。ES2022 为 Error 对象添加了一个**cause**属性，可以在生成错误时，添加报错原因的描述。

```
1 const actual = new Error('an error!', { cause: 'Error cause' });
2 actual.cause; // 'Error cause'
3
4 - try {
5     maybeWorks();
6 - } catch (err) {
7     throw new Error('maybeWorks failed!', { cause: err });
8 }
```

JavaScript

# 对象新增方法

## Object.is()

ES5中判断两个值是否相等，只有两个运算符：

- 相等运算符 (==)
  - 缺点是会自动转换数据类型
- 严格相等运算符 (===)
  - 缺点是 `NAN` 不等于自身，以及 `-0` 等于 `+0`

ES6提出“Same-value equality”（同值相等）算法，也就是 `Object.is`

```
1 Object.is('foo', 'foo')
2 // true
3 Object.is({}, {})
4 // false
```

与 `====` 行为基本一致，不同之处只有两个

- `NAN` 等于自身
- `+0` 不等于 `-0`

```
1 +0 === -0 //true
2 NaN === NaN // false
3
4 Object.is(+0, -0) // false
5 Object.is(NaN, NaN) // true
```

## Object.assign()

用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

```

1 const target = { a: 1, b: 1 };
2
3 const source1 = { b: 2, c: 2 };
4 const source2 = { c: 3 };
5
6 Object.assign(target, source1, source2);
7 target // {a:1, b:2, c:3}

```

## 不同参数

### 只有源对象参数

如果只有一个参数，会直接返回该参数

```

1 const obj = {a: 1};
2 Object.assign(obj) === obj // true

```

### 参数不是对象

会先转为对象

```

1 typeof Object.assign(2) // "object"

```

### 源对象参数是undefined/null

由于undefined和null无法转为对象，所以会报错

```

1 Object.assign(undefined) // 报错
2 Object.assign(null) // 报错

```

### 目标对象参数其他类型数据

如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果`undefined`和`null`不在首参数，就不会报错。

```
1 let obj = {a: 1};
2 Object.assign(obj, undefined) === obj // true
3 Object.assign(obj, null) === obj // true
```

其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。

```
1 const v1 = 'abc';
2 const v2 = true;
3 const v3 = 10;
4
5 const obj = Object.assign({}, v1, v2, v3);
6 console.log(obj); // { "0": "a", "1": "b", "2": "c" }
```

上面只有字符串被合入目标对象，因为字符串的包装对象有可枚举属性。

```
1 Object(true) // {[[PrimitiveValue]]: true}
2 Object(10) // {[[PrimitiveValue]]: 10}
3 Object('abc') // {0: "a", 1: "b", 2: "c", length: 3, {[[PrimitiveValue]]: "a
bc"}}
```

上面代码中，布尔值、数值、字符串分别转成对应的包装对象，可以看到它们的原始值都在包装对象的内部属性`[[[PrimitiveValue]]]`上面，这个属性是不会被`Object.assign()`拷贝的。只有字符串的包装对象，会产生可枚举的实义属性，那些属性则会被拷贝。

## 只拷贝源对象自身属性

`Object.assign()`拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（`enumerable: false`）。

```

1  Object.assign({b: 'c'},
2  Object.defineProperty({}, 'invisible', {
3      enumerable: false,
4      value: 'hello'
5  })
6 )
7 // { b: 'c' }

```

## 浅拷贝

`Object.assign()`方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```

1 const obj1 = {a: {b: 1}};
2 const obj2 = Object.assign({}, obj1);
3
4 obj1.a.b = 2;
5 obj2.a.b // 2

```

## 同名属性替换

对于这种嵌套的对象，一旦遇到同名属性，`Object.assign()`的处理方法是替换，而不是添加。

```

1 const target = { a: { b: 'c', d: 'e' } }
2 const source = { a: { b: 'hello' } }
3 Object.assign(target, source)
4 // { a: { b: 'hello' } }

```

## 数组处理

`Object.assign()`可以用来处理数组，但是会把数组视为对象。

```
1 Object.assign([1, 2, 3], [4, 5])
2 // [4, 5, 3]
```

## 取值函数处理

`Object.assign()`只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

```
1 const source = {
2   get foo() { return 1 }
3 };
4 const target = {};
5
6 Object.assign(target, source)
7 // { foo: 1 }
```

上面代码中，`source`对象的`foo`属性是一个取值函数，`Object.assign()`不会复制这个取值函数，只会拿到值以后，将这个值复制过去。

## Object.getOwnPropertyDescriptors()

ES5 的`Object.getOwnPropertyDescriptor()`方法会返回某个对象属性的描述对象

`(descriptor)`。ES2017 引入了`Object.getOwnPropertyDescriptors()`方法，返回指定对象所有自身属性（非继承属性）的描述对象。

```

1 const obj = {
2   foo: 123,
3   get bar() { return 'abc' }
4 };
5
6 Object.getOwnPropertyDescriptors(obj)
7 // { foo:
8 //   { value: 123,
9 //     writable: true,
10 //     enumerable: true,
11 //     configurable: true },
12 //   bar:
13 //   { get: [Function: get bar],
14 //     set: undefined,
15 //     enumerable: true,
16 //     configurable: true } }

```

## 解决Object.assign()无法拷贝get、set属性的问题

该方法的引入目的，主要是为了解决Object.assign()无法正确拷贝get属性和set属性的问题。

```

1 const source = {
2   set foo(value) {
3     console.log(value);
4   }
5 };
6
7 const target1 = {};
8 Object.assign(target1, source);
9
10 Object.getOwnPropertyDescriptor(target1, 'foo')
11 // { value: undefined,
12 //   writable: true,
13 //   enumerable: true,
14 //   configurable: true }

```

上面代码中，source对象的foo属性的值是一个赋值函数，Object.assign方法将这个属性拷贝给target1对象，结果该属性的值变成了undefined。这是因为Object.assign方法总是拷贝一个属性的值，而不会拷贝它背后的赋值方法或取值方法。

这时，`Object.getOwnPropertyDescriptors()`方法配合`Object.defineProperties()`方法，就可以实现正确拷贝。

```
1 const source = {
2   set foo(value) {
3     console.log(value);
4   }
5 };
6
7 const target2 = {};
8 Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source))
9;
10 Object.getOwnPropertyDescriptor(target2, 'foo')
11 // { get: undefined,
12 //   set: [Function: set foo],
13 //   enumerable: true,
14 //   configurable: true }
```

## 对象克隆

`Object.getOwnPropertyDescriptors()`方法的另一个用处，是配合`Object.create()`方法，将对象属性克隆到一个新对象。这属于浅拷贝。

```
1 const clone = Object.create(Object.getPrototypeOf(obj),
2   Object.getOwnPropertyDescriptors(obj));
3
4 // 或者
5
6 const shallowClone = (obj) => Object.create(
7   Object.getPrototypeOf(obj),
8   Object.getOwnPropertyDescriptors(obj)
9 );
```

上面代码会克隆对象`obj`。

## 对象继承

另外，`Object.getOwnPropertyDescriptors()`方法可以实现一个对象继承另一个对象。以前，继承另一个对象，常常写成下面这样。

```
1 const obj = {  
2   __proto__: prot,  
3   foo: 123,  
4 };
```

ES6 规定`__proto__`只有浏览器要部署，其他环境不用部署。如果去除`__proto__`，上面代码就要改成下面这样。

```
1 const obj = Object.create(prot);  
2 obj.foo = 123;  
3  
4 // 或者  
5  
6 const obj = Object.assign(  
7   Object.create(prot),  
8   {  
9     foo: 123,  
10    }  
11 );
```

有了`Object.getOwnPropertyDescriptors()`，我们就有了另一种写法。

```
1 const obj = Object.create(  
2   prot,  
3   Object.getOwnPropertyDescriptors({  
4     foo: 123,  
5   })  
6 );
```

`Object.getOwnPropertyDescriptors()`也可以用来实现 Mixin（混入）模式。

```

1 let mix = (object) => ({
2   with: (...mixins) => mixins.reduce(
3     (c, mixin) => Object.create(
4       c, Object.getOwnPropertyDescriptors(mixin)
5     ), object)
6   });
7
8 // multiple mixins example
9 let a = {a: 'a'};
10 let b = {b: 'b'};
11 let c = {c: 'c'};
12 let d = mix(c).with(a, b);
13
14 d.c // "c"
15 d.b // "b"
16 d.a // "a"

```

上面代码返回一个新的对象d，代表了对象a和b被混入了对象c的操作。

## \_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()

### \_\_proto\_\_属性

\_\_proto\_\_属性（前后各两个下划线），用来读取或设置当前对象的原型对象（prototype）。目前，所有浏览器（包括IE11）都部署了这个属性。

```

1 // es5 的写法
2 const obj = {
3   method: function() { ... }
4 };
5 obj.__proto__ = someOtherObj;
6
7 // es6 的写法
8 var obj = Object.create(someOtherObj);
9 obj.method = function() { ... };

```

该属性没有写入 ES6 的正文，而是写入了附录，原因是`__proto__`前后的双下划线，说明它本质上是一个内部属性，而不是一个正式的对外的 API，只是由于浏览器广泛支持，才被加入了 ES6。标准明确规定，只有浏览器必须部署这个属性，其他运行环境不一定需要部署，而且新的代码最好认为这个属性是不存在的。因此，无论从语义的角度，还是从兼容性的角度，都不要使用这个属性，而是使用下面的`Object.setPrototypeOf()`（写操作）、`Object.getPrototypeOf()`（读操作）、`Object.create()`（生成操作）代替。

实现上，`__proto__`调用的是`Object.prototype.__proto__`，具体实现如下。

```
JavaScript | 1  Object.defineProperty(Object.prototype, '__proto__', {
2    get() {
3      let _thisObj = Object(this);
4      return Object.getPrototypeOf(_thisObj);
5    },
6    set(proto) {
7      if (this === undefined || this === null) {
8        throw new TypeError();
9      }
10     if (!isObject(this)) {
11       return undefined;
12     }
13     if (!isObject(proto)) {
14       return undefined;
15     }
16     let status = Reflect.setPrototypeOf(this, proto);
17     if (!status) {
18       throw new TypeError();
19     }
20   },
21 });
22
23 function isObject(value) {
24   return Object(value) === value;
25 }
```

## Object.setPrototypeOf()

`Object.setPrototypeOf`方法的作用与`__proto__`相同，用来设置一个对象的原型对象（prototype），返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。

```

1 let proto = {};
2 let obj = { x: 10 };
3 Object.setPrototypeOf(obj, proto);
4
5 proto.y = 20;
6 proto.z = 40;
7
8 obj.x // 10
9 obj.y // 20
10 obj.z // 40

```

上面代码将`proto`对象设为`obj`对象的原型，所以从`obj`对象可以读取`proto`对象的属性。

如果第一个参数不是对象，会自动转为对象。但是由于返回的还是第一个参数，所以这个操作不会产生任何效果。

```

1 Object.setPrototypeOf(1, {}) === 1 // true
2 Object.setPrototypeOf('foo', {}) === 'foo' // true
3 Object.setPrototypeOf(true, {}) === true // true

```

由于`undefined`和`null`无法转为对象，所以如果第一个参数是`undefined`或`null`，就会报错。

```

1 Object.setPrototypeOf(undefined, {})
2 // TypeError: Object.setPrototypeOf called on null or undefined
3
4 Object.setPrototypeOf(null, {})
5 // TypeError: Object.setPrototypeOf called on null or undefined

```

## Object.getPrototypeOf()

该方法与`Object.setPrototypeOf`方法配套，用于读取一个对象的原型对象。

```

1  function Rectangle() {
2      // ...
3  }
4
5  const rec = new Rectangle();
6
7  Object.getPrototypeOf(rec) === Rectangle.prototype
8  // true
9
10 Object.setPrototypeOf(rec, Object.prototype);
11 Object.getPrototypeOf(rec) === Rectangle.prototype
12 // false

```

如果参数不是对象，会被自动转为对象。

```

1 // 等同于 Object.getPrototypeOf(Number(1))
2 Object.getPrototypeOf(1)
3 // Number {[[PrimitiveValue]]: 0}
4
5 // 等同于 Object.getPrototypeOf(String('foo'))
6 Object.getPrototypeOf('foo')
7 // String {length: 0, [[PrimitiveValue]]: ""}
8
9 // 等同于 Object.getPrototypeOf(Boolean(true))
10 Object.getPrototypeOf(true)
11 // Boolean {[[PrimitiveValue]]: false}
12
13 Object.getPrototypeOf(1) === Number.prototype // true
14 Object.getPrototypeOf('foo') === String.prototype // true
15 Object.getPrototypeOf(true) === Boolean.prototype // true

```

如果参数是`undefined`或`null`，它们无法转为对象，所以会报错。

```

1 Object.getPrototypeOf(null)
2 // TypeError: Cannot convert undefined or null to object
3
4 Object.getPrototypeOf(undefined)
5 // TypeError: Cannot convert undefined or null to object

```

# Object.keys(), Object.values(), Object.entries()

## Object.keys()

ES5 引入了 `Object.keys` 方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

```
1 var obj = { foo: 'bar', baz: 42 };
2 Object.keys(obj)
3 // ["foo", "baz"]
```

ES2017 引入了跟 `Object.keys` 配套的 `Object.values` 和 `Object.entries`，作为遍历一个对象的补充手段，供 `for...of` 循环使用。

```
1 let {keys, values, entries} = Object;
2 let obj = { a: 1, b: 2, c: 3 };
3
4 for (let key of keys(obj)) {
5   console.log(key); // 'a', 'b', 'c'
6 }
7
8 for (let value of values(obj)) {
9   console.log(value); // 1, 2, 3
10 }
11
12 for (let [key, value] of entries(obj)) {
13   console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]
```

## Object.values()

`Object.values` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。

```
1 const obj = { foo: 'bar', baz: 42 };
2 Object.values(obj)
3 // ["bar", 42]
```

返回数组的成员顺序，与本章的《属性的遍历》部分介绍的排列规则一致。

```
1 const obj = { 100: 'a', 2: 'b', 7: 'c' };
2 Object.values(obj)
3 // ["b", "c", "a"]
```

上面代码中，属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是b、c、a。

`Object.values`只返回对象自身的可遍历属性。

```
1 const obj = Object.create({}, {p: {value: 42}});
2 Object.values(obj) // []
```

上面代码中，`Object.create`方法的第二个参数添加的对象属性（属性p），如果不显式声明，默认是不可遍历的，因为p的属性描述对象的`enumerable`默认是`false`，`Object.values`不会返回这个属性。只要把`enumerable`改成`true`，`Object.values`就会返回属性p的值。

```
1 const obj = Object.create({}, {p:
2   {
3     value: 42,
4     enumerable: true
5   }
6 });
7 Object.values(obj) // [42]
```

`Object.values`会过滤属性名为`Symbol`值的属性。

```
1 Object.values({ [Symbol()]: 123, foo: 'abc' });
2 // ['abc']
```

如果`Object.values`方法的参数是一个字符串，会返回各个字符组成的一个数组。

```
1 Object.values('foo')
2 // ['f', 'o', 'o']
```

上面代码中，字符串会先转成一个类似数组的对象。字符串的每个字符，就是该对象的一个属性。因此，`Object.values`返回每个属性的键值，就是各个字符组成的一个数组。

如果参数不是对象，`Object.values`会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以，`Object.values`会返回空数组。

```
1 Object.values(42) // []
2 Object.values(true) // []
```

## Object.entries()

`Object.entries()`方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。

```
1 const obj = { foo: 'bar', baz: 42 };
2 Object.entries(obj)
3 // [ ["foo", "bar"], ["baz", 42] ]
```

除了返回值不一样，该方法的行为与`Object.values`基本一致。

如果原对象的属性名是一个 Symbol 值，该属性会被忽略。

```
1 Object.entries({ [Symbol()]: 123, foo: 'abc' });
2 // [ [ 'foo', 'abc' ] ]
```

上面代码中，原对象有两个属性，`Object.entries`只输出属性名非 Symbol 值的属性。将来可能会有`Reflect.ownEntries()`方法，返回对象自身的所有属性。

`Object.entries`的基本用途是遍历对象的属性。

```

1 let obj = { one: 1, two: 2 };
2 for (let [k, v] of Object.entries(obj)) {
3   console.log(
4     `${JSON.stringify(k)}: ${JSON.stringify(v)}`
5   );
6 }
7 // "one": 1
8 // "two": 2

```

Object.entries方法的另一个用处是，将对象转为真正的Map结构。

```

1 const obj = { foo: 'bar', baz: 42 };
2 const map = new Map(Object.entries(obj));
3 map // Map { foo: "bar", baz: 42 }

```

自己实现Object.entries方法，非常简单。

```

1 // Generator函数的版本
2 function* entries(obj) {
3   for (let key of Object.keys(obj)) {
4     yield [key, obj[key]];
5   }
6 }
7
8 // 非Generator函数的版本
9 function entries(obj) {
10   let arr = [];
11   for (let key of Object.keys(obj)) {
12     arr.push([key, obj[key]]);
13   }
14   return arr;
15 }

```

## Object.fromEntries()

Object.fromEntries()方法是Object.entries()的逆操作，用于将一个键值对数组转为对象。

```
1 ▼ Object.fromEntries([
2     ['foo', 'bar'],
3     ['baz', 42]
4 ])
5 // { foo: "bar", baz: 42 }
```

JavaScript

该方法的主要目的，是将键值对的数据结构还原为对象，因此特别适合将 Map 结构转为对象。

```
1 // 例一
2 ▼ const entries = new Map([
3     ['foo', 'bar'],
4     ['baz', 42]
5 ]);
6
7 Object.fromEntries(entries)
8 // { foo: "bar", baz: 42 }
9
10 // 例二
11 const map = new Map().set('foo', true).set('bar', false);
12 Object.fromEntries(map)
13 // { foo: true, bar: false }
```

JavaScript

该方法的一个用处是配合[URLSearchParams](#)对象，将查询字符串转为对象。

```
1 Object.fromEntries(new URLSearchParams('foo=bar&baz=qux'))
2 // { foo: "bar", baz: "qux" }
```

JavaScript

## Object.hasOwnProperty()

JavaScript 对象的属性分成两种：自身的属性和继承的属性。对象实例有一个[hasOwnProperty\(\)](#)方法，可以判断某个属性是否为原生属性。ES2022 在[Object](#)对象上面新增了一个静态方法[Object.hasOwnProperty\(\)](#)，也可以判断是否为自身的属性。

[Object.hasOwnProperty\(\)](#)可以接受两个参数，第一个是所要判断的对象，第二个是属性名。

```
1 const foo = Object.create({ a: 123 });
2 foo.b = 456;
3
4 Object.hasOwn(foo, 'a') // false
5 Object.hasOwn(foo, 'b') // true
```

上面示例中，对象`foo`的属性`a`是继承属性，属性`b`是原生属性。`Object.hasOwn()`对属性`a`返回`false`，对属性`b`返回`true`。

`Object.hasOwn()`的一个好处是，对于不继承`Object.prototype`的对象不会报错，而`hasOwnProperty()`是会报错的。

```
1 const obj = Object.create(null);
2
3 obj.hasOwnProperty('foo') // 报错
4 Object.hasOwn(obj, 'foo') // false
```

上面示例中，`Object.create(null)`返回的对象`obj`是没有原型的，不继承任何属性，这导致调用`obj.hasOwnProperty()`会报错，但是`Object.hasOwn()`就能正确处理这种情况。

# 运算符扩展

## 指数运算符

ES2016 新增了一个指数运算符 (`**`)。

```
1 2 ** 2 // 4
2 2 ** 3 // 8
```

JavaScript |

这个运算符的一个特点是右结合，而不是常见的左结合。多个指数运算符连用时，是从最右边开始计算的。

```
1 // 相当于 2 ** (3 ** 2)
2 2 ** 3 ** 2
3 // 512
```

JavaScript |

## 链判断运算符

在开发过程中，对于多层对象，如果读取对象内部某个属性，往往需要逐级判断

```
1 // 错误的写法
2 const firstName = message.body.user.firstName || 'default';
3
4 // 正确的写法
5 const firstName = (message
6   && message.body
7   && message.body.user
8   && message.body.user.firstName) || 'default';
```

JavaScript |

这样的层层判断非常麻烦，因此 [ES2020](#) 引入了“链判断运算符” (optional chaining operator) `?.`，简化上面的写法。

```

1 const firstName = message?.body?.user?.firstName || 'default';
2 const fooValue = myForm.querySelector('input[name=foo]??.value

```

上面代码使用了`?.`运算符，直接在链式调用的时候判断，左侧的对象是否为`null`或`undefined`。如果是的，就不再往下运算，而是返回`undefined`。

下面是判断对象方法是否存在，如果存在就立即执行的例子。

```

1 iterator?.return??.()

```

上面代码中，`iterator.return`如果有定义，就会调用该方法，否则`iterator.return`直接返回`undefined`，不再执行`?.`后面的部分。

链判断运算符`?.`有三种写法。

- `obj?.prop` // 对象属性是否存在
- `obj?.[expr]` // 同上
- `func?(...args)` // 函数或对象方法是否存在

下面是`?.`运算符常见形式，以及不使用该运算符时的等价形式。

```

1 a?.b
2 // 等同于
3 a == null ? undefined : a.b
4
5 a?.[x]
6 // 等同于
7 a == null ? undefined : a[x]
8
9 a?.b()
10 // 等同于
11 a == null ? undefined : a.b()
12
13 a??.()
14 // 等同于
15 a == null ? undefined : a()

```

上面代码中，特别注意后两种形式，如果`a?.b()`和`a??.()`。如果`a?.b()`里面的`a.b`有值，但不是函数，不可调用，那么`a?.b()`是会报错的。`a??.()`也是如此，如果`a`不是`null`或`undefined`，但也不是函数，

那么`a?.()`会报错。

使用这个运算符，还有几个注意点

## 短路机制

本质上，`?.`运算符相当于一种短路机制，只要不满足条件，就不再往下执行。

```
1 a?.[++x]
2 // 等同于
3 a == null ? undefined : a[++x]
```

上面代码中，如果`a`是`undefined`或`null`，那么`x`不会进行递增运算。也就是说，链判断运算符一旦为真，右侧的表达式就不再求值。

## 括号影响

如果属性链有圆括号，链判断运算符对圆括号外部没有影响，只对圆括号内部有影响。

```
1 (a?.b).c
2 // 等价于
3 (a == null ? undefined : a.b).c
```

上面代码中，`?.`对圆括号外部没有影响，不管`a`对象是否存在，圆括号后面的`.c`总是会执行。

一般来说，使用`?.`运算符的场合，不应该使用圆括号。

## 报错场合

以下写法是禁止的，会报错。

```

1 // 构造函数
2 new a?.()
3 new a?.b()

4
5 // 链判断运算符的右侧有模板字符串
6 a?.`{b}`
7 a?.b`{c}`

8
9 // 链判断运算符的左侧是 super
10 super?.()
11 super?.foo

12
13 // 链运算符用于赋值运算符左侧
14 a?.b = c

```

## 右侧不得为十进制数值

为了保证兼容以前的代码，允许`foo?.3:0`被解析成`foo ? .3 : 0`，因此规定如果`?.`后面紧跟一个十进制数字，那么`?.`不再被看成是一个完整的运算符，而会按照三元运算符进行处理，也就是说，那个小数点会归属于后面的十进制数字，形成一个小数。

## Null判断运算符

读取对象属性的时候，如果某个属性的值是`null`或`undefined`，有时候需要为它们指定默认值。常见做法是通过`||`运算符指定默认值。

```

1 const headerText = response.settings.headerText || 'Hello, world!';
2 const animationDuration = response.settings.animationDuration || 300;
3 const showSplashScreen = response.settings.showSplashScreen || true;

```

上面的三行代码都通过`||`运算符指定默认值，但是这样写是错的。开发者的原意是，只要属性的值为`null`或`undefined`，默认值就会生效，但是属性的值如果为空字符串或`false`或`0`，默认值也会生效。

为了避免这种情况，[ES2020](#) 引入了一个新的 Null 判断运算符`??`。它的行为类似`||`，但是只有运算符左侧的值为`null`或`undefined`时，才会返回右侧的值。

```

1 const headerText = response.settings.headerText ?? 'Hello, world!';
2 const animationDuration = response.settings.animationDuration ?? 300;
3 const showSplashScreen = response.settings.showSplashScreen ?? true;

```

上面代码中，`默认值`只有在左侧属性值为`null`或`undefined`时，才会生效。

这个运算符的一个目的，就是跟链判断运算符`?.`配合使用，为`null`或`undefined`的值设置默认值。

```

1 const animationDuration = response.settings?.animationDuration ?? 300;

```

上面代码中，如果`response.settings`是`null`或`undefined`，或者`response.settings.animationDuration`是`null`或`undefined`，就会返回默认值`300`。也就是说，这一行代码包括了两级属性的判断。

## 逻辑赋值运算符

ES2021 引入了三个新的逻辑赋值运算符（logical assignment operators），将逻辑运算符与赋值运算符进行结合。

```

1 // 或赋值运算符
2 x ||= y
3 // 等同于
4 x || (x = y)
5
6 // 与赋值运算符
7 x &&= y
8 // 等同于
9 x && (x = y)
10
11 // Null 赋值运算符
12 x ??= y
13 // 等同于
14 x ?? (x = y)

```

它们的一个用途是，为变量或属性设置默认值。

```
1 // 老的写法
2 user.id = user.id || 1;
3
4 // 新的写法
5 user.id ||= 1;
```

JavaScript |

下面是另一个例子。

```
1 function example(opts) {
2     opts.foo = opts.foo ?? 'bar';
3     opts.baz ?? (opts.baz = 'qux');
4 }
```

JavaScript |

上面示例中，参数对象`opts`如果不存在属性`foo`和属性`baz`，则为这两个属性设置默认值。有了“Null 赋值运算符”以后，就可以统一写成下面这样。

```
1 function example(opts) {
2     opts.foo ??= 'bar';
3     opts.baz ??= 'qux';
```

JavaScript |

## #!命令

Unix 的命令行脚本都支持`#!`命令，又称为 Shebang 或 Hashbang。这个命令放在脚本的第一行，用来指定脚本的执行器。

比如 Bash 脚本的第一行。

```
1 #!/bin/sh
```

Bash |

Python 脚本的第一行。

```
1 #!/usr/bin/env python
```

Python |

ES2023 为 JavaScript 脚本引入了`#!`命令，写在脚本文件或者模块文件的第一行。

```
1 // 写在脚本文件第一行
2 #!/usr/bin/env node
3 'use strict';
4 console.log(1);
5
6 // 写在模块文件第一行
7 #!/usr/bin/env node
8 export {};
9 console.log(1);
```

有了这一行以后，Unix 命令行就可以直接执行脚本。

```
1 # 以前执行脚本的方式
2 $ node hello.js
3
4 # hashbang 的方式
5 $ ./hello.js
```

对于 JavaScript 引擎来说，会把`#!`理解成注释，忽略掉这一行。

# Symbol

## 概述

### 引入Symbol的原因

由于对象的属性名都是字符串，如果你引用了一个其他人提供的对象，想为这个对象添加方法，新方法的名字就有可能与对象现有方法产生冲突。

所以ES6引入了一中心的**原始数据类型**Symbol。现在对象除了可以用字符串做属性名，还可以用Symbol类型做属性名。

### Symbol声明

`Symbol()` 函数前不能使用 `new` 命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象，所以不能使用 `new` 命令来调用。另外，由于 Symbol 值不是对象，所以也不能添加属性。

`Symbol()` 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述。这主要是为了在控制台显示，或者转为字符串时，比较容易区分。

```
▼ JavaScript |  
1 let s1 = Symbol('foo');  
2 let s2 = Symbol('bar');  
3  
4 s1 // Symbol(foo)  
5 s2 // Symbol(bar)  
6  
7 s1.toString() // "Symbol(foo)"  
8 s2.toString() // "Symbol(bar)"
```

### Symbol的类型转换

如果 Symbol 的参数是一个对象，就会调用该对象的 `toString()` 方法，将其转为字符串，然后才生成一个 Symbol 值。

```
1 const obj = {  
2   toString() {  
3     return 'abc';  
4   }  
5 };  
6 const sym = Symbol(obj);  
7 sym // Symbol(abc)
```

JavaScript

Symbol 值不能与其他类型的值进行运算，会报错。

```
1 let sym = Symbol('My symbol');  
2  
3 "your symbol is " + sym  
4 // TypeError: can't convert symbol to string  
5 `your symbol is ${sym}`  
6 // TypeError: can't convert symbol to string
```

JavaScript

但是，Symbol 值可以显式转为字符串。

```
1 let sym = Symbol('My symbol');  
2  
3 String(sym) // 'Symbol(My symbol)'  
4 sym.toString() // 'Symbol(My symbol)'
```

JavaScript

另外，Symbol 值也可以转为布尔值，但是不能转为数值。

```
1 let sym = Symbol();  
2 Boolean(sym) // true  
3 !sym // false  
4  
5 if (sym) {  
6   // ...  
7 }  
8  
9 Number(sym) // TypeError  
10 sym + 2 // TypeError
```

JavaScript

# Symbol.prototype.description

前面我们说过创建Symbol值时可以用参数添加描述，但是读取描述需要转为字符串太麻烦。

```
1 const sym = Symbol('foo');
2
3 String(sym) // "Symbol(foo)"
4 sym.toString() // "Symbol(foo)"
```

所以ES2019给Symbol添加了一个实例属性 `description`，直接返回Symbol值的描述，

```
1 const sym = Symbol('foo');
2
3 sym.description // "foo"
```

## 作为属性名的 Symbol

Symbol作为对象属性时，不能用点运算符。

```
1 const mySymbol = Symbol();
2 const a = {};
3
4 a.mySymbol = 'Hello!';
5 a[mySymbol] // undefined
6 a['mySymbol'] // "Hello!"
```

上面因为用的点运算符，点运算符后面只能是字符串，所以其实是让字符串 `mySymbol` 作为对象属性名。

Symbol 类型还可以用于定义一组常量，保证这组常量的值都是不相等的。

```

1 const log = {};
2
3 log.levels = {
4   DEBUG: Symbol('debug'),
5   INFO: Symbol('info'),
6   WARN: Symbol('warn')
7 };
8 console.log(log.levels.DEBUG, 'debug message');
9 console.log(log.levels.INFO, 'info message');

```

## 消除魔术字符串

魔术字符串指的是，在代码之中多次出现、与代码形成强耦合的某一个具体的字符串或者数值。风格良好的代码，应该尽量消除魔术字符串，改由含义清晰的变量代替。

```

1 function getArea(shape, options) {
2   let area = 0;
3
4   switch (shape) {
5     case 'Triangle': // 魔术字符串
6       area = .5 * options.width * options.height;
7       break;
8     /* ... more code ... */
9   }
10
11   return area;
12 }
13
14 getArea('Triangle', { width: 100, height: 100 }); // 魔术字符串

```

上面代码中，字符串`Triangle`就是一个魔术字符串。它多次出现，与代码形成“强耦合”，不利于将来的修改和维护。

常用的消除魔术字符串的方法，就是把它写成一个变量。

```

1  const shapeType = {
2      triangle: 'Triangle'
3  };
4
5  function getArea(shape, options) {
6      let area = 0;
7      switch (shape) {
8          case shapeType.triangle:
9              area = .5 * options.width * options.height;
10         break;
11     }
12     return area;
13 }
14
15 getArea(shapeType.triangle, { width: 100, height: 100 });

```

上面代码中，我们把`Triangle`写成`shapeType`对象的`triangle`属性，这样就消除了强耦合。

如果仔细分析，可以发现`shapeType.triangle`等于哪个值并不重要，只要确保不会跟其他`shapeType`属性的值冲突即可。因此，这里就很适合改用`Symbol`值。

```

1  const shapeType = {
2      triangle: Symbol()
3  };

```

上面代码中，除了将`shapeType.triangle`的值设为一个`Symbol`，其他地方都不用修改。

## 属性名的遍历

由于`Symbol`值作为属性名，遍历对象的时候，该属性不会出现在`for...in`、`for...of`循环中，也不会被`Object.keys()`、`Object.getOwnPropertyNames()`、`JSON.stringify()`返回。

但是，它也不是私有属性，有一个`Object.getOwnPropertySymbols()`方法，可以获取指定对象的所有`Symbol`属性名。该方法返回一个数组，成员是当前对象的所有用作属性名的`Symbol`值。

```
1 const obj = {};
2 let a = Symbol('a');
3 let b = Symbol('b');
4
5 obj[a] = 'Hello';
6 obj[b] = 'World';
7
8 const objectSymbols = Object.getOwnPropertySymbols(obj);
9
10 objectSymbols
11 // [Symbol(a), Symbol(b)]
```

另一个新的 API, `Reflect.ownKeys()`方法可以返回所有类型的键名, 包括常规键名和 `Symbol` 键名。

```
1 let obj = {
2   [Symbol('my_key')]: 1,
3   enum: 2,
4   nonEnum: 3
5 };
6
7 Reflect.ownKeys(obj)
8 // ["enum", "nonEnum", Symbol(my_key)]
```

由于以 `Symbol` 值作为键名, 不会被常规方法遍历得到。我们可以利用这个特性, 为对象定义一些非私有的、但又希望只用于内部的方法。

```

1  let size = Symbol('size');
2
3  class Collection {
4    constructor() {
5      this[size] = 0;
6    }
7
8    add(item) {
9      this[this[size]] = item;
10     this[size]++;
11   }
12
13   static sizeOf(instance) {
14     return instance[size];
15   }
16 }
17
18 let x = new Collection();
19 Collection.sizeOf(x) // 0
20
21 x.add('foo');
22 Collection.sizeOf(x) // 1
23
24 Object.keys(x) // ['0']
25 Object.getOwnPropertyNames(x) // ['0']
26 Object.getOwnPropertySymbols(x) // [Symbol(size)]

```

上面代码中，对象x的size属性是一个Symbol值，所以Object.keys(x)、Object.getOwnPropertyNames(x)都无法获取它。这就造成了一种非私有的内部方法的效果。

## Symbol.for(), Symbol.keyFor()

有时，我们希望重新使用同一个Symbol值，`Symbol.for()`方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的Symbol值。如果有，就返回这个Symbol值，否则就新建一个以该字符串为名称的Symbol值，并将其[注册到全局](#)。

```

1 let s1 = Symbol.for('foo');
2 let s2 = Symbol.for('foo');
3
4 s1 === s2 // true

```

`Symbol.for()`与`Symbol()`这两种写法，都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，后者不会。`Symbol.for()`不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。比如，如果你调用`Symbol.for("cat")`30 次，每次都会返回同一个 Symbol 值，但是调用`Symbol("cat")`30 次，会返回 30 个不同的 Symbol 值。

```

1 Symbol.for("bar") === Symbol.for("bar")
2 // true
3
4 Symbol("bar") === Symbol("bar")
5 // false

```

`Symbol.keyFor()`方法返回一个已登记的 Symbol 类型值的key。

```

1 let s1 = Symbol.for("foo");
2 Symbol.keyFor(s1) // "foo"
3
4 let s2 = Symbol("foo");
5 Symbol.keyFor(s2) // undefined

```

上面代码中，变量`s2`属于未登记的 Symbol 值，所以返回`undefined`。

注意，`Symbol.for()`为 Symbol 值登记的名字，是全局环境的，不管有没有在全局环境运行。

```

1 function foo() {
2   return Symbol.for('bar');
3 }
4
5 const x = foo();
6 const y = Symbol.for('bar');
7 console.log(x === y); // true

```

上面代码中，`Symbol.for('bar')`是函数内部运行的，但是生成的 Symbol 值是登记在全局环境的。所以，第二次运行`Symbol.for('bar')`可以取到这个 Symbol 值。

`Symbol.for()`的这个全局登记特性，可以用在不同的 iframe 或 service worker 中取到同一个值。

JavaScript

```
1  iframe = document.createElement('iframe');
2  iframe.src = String(window.location);
3  document.body.appendChild(iframe);
4
5  iframe.contentWindow.Symbol.for('foo') === Symbol.for('foo')
6  // true
```

上面代码中，iframe 窗口生成的 Symbol 值，可以在主页面得到。

## 内置的Symbol值

### Symbol.hasInstance

对象的`Symbol.hasInstance`属性，指向一个内部方法。当其他对象使用`instanceof`运算符，判断是否为该对象的实例时，会调用这个方法。比如，`foo instanceof Foo`在语言内部，实际调用的是`Foo[Symbol.hasInstance](foo)`。

```

1  class MyClass {
2    [Symbol.hasInstance](foo) {
3      return foo instanceof Array;
4    }
5  }
6
7  [1, 2, 3] instanceof new MyClass() // true
8
9
10 class Even {
11   static [Symbol.hasInstance](obj) {
12     return Number(obj) % 2 === 0;
13   }
14 }
15
16 // 等同于
17 const Even = {
18   [Symbol.hasInstance](obj) {
19     return Number(obj) % 2 === 0;
20   }
21 };
22
23 1 instanceof Even // false
24 2 instanceof Even // true
25 12345 instanceof Even // false

```

## Symbol.isConcatSpreadable

对象的`Symbol.isConcatSpreadable`属性等于一个布尔值，表示该对象用于`Array.prototype.concat()`时，是否可以展开。

```

1 let arr1 = ['c', 'd'];
2 ['a', 'b'].concat(arr1, 'e') // ['a', 'b', 'c', 'd', 'e']
3 arr1[Symbol.isConcatSpreadable] // undefined
4
5 let arr2 = ['c', 'd'];
6 arr2[Symbol.isConcatSpreadable] = false;
7 ['a', 'b'].concat(arr2, 'e') // ['a', 'b', ['c', 'd'], 'e']

```

上面代码说明，数组的默认行为是可以展开，`Symbol.isConcatSpreadable`默认等于`undefined`。该属性等于`true`时，也有展开的效果。

类似数组的对象正好相反，默认不展开。它的`Symbol.isConcatSpreadable`属性设为`true`，才可以展开。

```
1 let obj = {length: 2, 0: 'c', 1: 'd'};
2 ['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']
3
4 obj[Symbol.isConcatSpreadable] = true;
5 ['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']
```

`Symbol.isConcatSpreadable`属性也可以定义在类里面。

```
1 class A1 extends Array {
2   constructor(args) {
3     super(args);
4     this[Symbol.isConcatSpreadable] = true;
5   }
6 }
7 class A2 extends Array {
8   constructor(args) {
9     super(args);
10 }
11 get [Symbol.isConcatSpreadable]() {
12   return false;
13 }
14 }
15 let a1 = new A1();
16 a1[0] = 3;
17 a1[1] = 4;
18 let a2 = new A2();
19 a2[0] = 5;
20 a2[1] = 6;
21 [1, 2].concat(a1).concat(a2)
22 // [1, 2, 3, 4, [5, 6]]
```

上面代码中，类`A1`是可展开的，类`A2`是不可展开的，所以使用`concat`时有不一样的结果。

注意，`Symbol.isConcatSpreadable`的位置差异，`A1`是定义在实例上，`A2`是定义在类本身，效果相同。

## Symbol.species

对象的Symbol.species属性，指向一个构造函数。创建衍生对象时，会使用该属性。

```
1 class MyArray extends Array {  
2 }  
3  
4 const a = new MyArray(1, 2, 3);  
5 const b = a.map(x => x);  
6 const c = a.filter(x => x > 1);  
7  
8 b instanceof MyArray // true  
9 c instanceof MyArray // true
```

上面代码中，子类MyArray继承了父类Array，a是MyArray的实例，b和c是a的衍生对象。你可能会认为，b和c都是调用数组方法生成的，所以应该是数组（Array的实例），但实际上它们也是MyArray的实例。

Symbol.species属性就是为了解决这个问题而提供的。现在，我们可以为MyArray设置Symbol.species属性。

```
1 class MyArray extends Array {  
2   static get [Symbol.species]() { return Array; }  
3 }
```

上面代码中，由于定义了Symbol.species属性，创建衍生对象时就会使用这个属性返回的函数，作为构造函数。这个例子也说明，定义Symbol.species属性要采用get取值器。默认的Symbol.species属性等同于下面的写法。

```
1 static get [Symbol.species]() {  
2   return this;  
3 }
```

现在，再来看前面的例子。

```

1 class MyArray extends Array {
2     static get [Symbol.species]() { return Array; }
3 }
4
5 const a = new MyArray();
6 const b = a.map(x => x);
7
8 b instanceof MyArray // false
9 b instanceof Array // true

```

上面代码中，`a.map(x => x)`生成的衍生对象，就不是`MyArray`的实例，而直接就是`Array`的实例。  
再看一个例子。

```

1 class T1 extends Promise {
2 }
3
4 class T2 extends Promise {
5     static get [Symbol.species]() {
6         return Promise;
7     }
8 }
9
10 new T1(r => r()).then(v => v) instanceof T1 // true
11 new T2(r => r()).then(v => v) instanceof T2 // false

```

上面代码中，`T2`定义了`Symbol.species`属性，`T1`没有。结果就导致了创建衍生对象时（`then`方法），`T1`调用的是自身的构造方法，而`T2`调用的是`Promise`的构造方法。

总之，`Symbol.species`的作用在于，实例对象在运行过程中，需要再次调用自身的构造函数时，会调用该属性指定的构造函数。它主要的用途是，有些类库是在基类的基础上修改的，那么子类使用继承的方法时，作者可能希望返回基类的实例，而不是子类的实例。

## Symbol.match

对象的`Symbol.match`属性，指向一个函数。当执行`str.match(myObject)`时，如果该属性存在，会调用它，返回该方法的返回值。

```

1 String.prototype.match(regex)
2 // 等同于
3 regexp[Symbol.match](this)
4
5 - class MyMatcher {
6 -   [Symbol.match](string) {
7     return 'hello world'.indexOf(string);
8   }
9 }
10
11 'e'.match(new MyMatcher()) // 1

```

## Symbol.replace

对象的Symbol.replace属性，指向一个方法，当该对象被String.prototype.replace方法调用时，会返回该方法的返回值。

```

1 String.prototype.replace(searchValue, replaceValue)
2 // 等同于
3 searchValue[Symbol.replace](this, replaceValue)

```

下面是一个例子。

```

1 const x = {};
2 x[Symbol.replace] = (...s) => console.log(s);
3
4 'Hello'.replace(x, 'World') // ["Hello", "World"]

```

Symbol.replace方法会收到两个参数，第一个参数是replace方法正在作用的对象，上面例子是Hello，第二个参数是替换后的值，上面例子是World。

## Symbol.search

对象的Symbol.search属性，指向一个方法，当该对象被String.prototype.search方法调用时，会返回该方法的返回值。

```
1 String.prototype.search(regexp)
2 // 等同于
3 regexp[Symbol.search](this)
4
5 - class MySearch {
6 -   constructor(value) {
7     this.value = value;
8   }
9 -   [Symbol.search](string) {
10    return string.indexOf(this.value);
11  }
12 }
13 'foobar'.search(new MySearch('foo')) // 0
```

## Symbol.split

对象的**Symbol.split**属性，指向一个方法，当该对象被**String.prototype.split**方法调用时，会返回该方法的返回值。

```
1 String.prototype.split(separator, limit)
2 // 等同于
3 separator[Symbol.split](this, limit)
```

下面是一个例子。

```

1  class MySplitter {
2    constructor(value) {
3      this.value = value;
4    }
5    [Symbol.split](string) {
6      let index = string.indexOf(this.value);
7      if (index === -1) {
8        return string;
9      }
10     return [
11       string.substr(0, index),
12       string.substr(index + this.value.length)
13     ];
14   }
15 }
16
17 'foobar'.split(new MySplitter('foo'))
18 // ['', 'bar']
19
20 'foobar'.split(new MySplitter('bar'))
21 // ['foo', '']
22
23 'foobar'.split(new MySplitter('baz'))
24 // 'foobar'

```

上面方法使用`Symbol.split`方法，重新定义了字符串对象的`split`方法的行为，

## Symbol.iterator

对象的`Symbol.iterator`属性，指向该对象的默认遍历器方法。

```

1  const myIterable = {};
2  myIterable[Symbol.iterator] = function* () {
3    yield 1;
4    yield 2;
5    yield 3;
6  };
7
8  [...myIterable] // [1, 2, 3]

```

对象进行`for...of`循环时，会调用`Symbol.iterator`方法，返回该对象的默认遍历器，详细介绍参见《Iterator 和 for...of 循环》一章。

The screenshot shows a code editor window with the title 'JavaScript'. The code in the editor is as follows:

```
1  class Collection {
2    *[Symbol.iterator]() {
3      let i = 0;
4      while(this[i] !== undefined) {
5        yield this[i];
6        ++i;
7      }
8    }
9  }
10
11 let myCollection = new Collection();
12 myCollection[0] = 1;
13 myCollection[1] = 2;
14
15 for(let value of myCollection) {
16   console.log(value);
17 }
18 // 1
19 // 2
```

## Symbol.toPrimitive

对象的`Symbol.toPrimitive`属性，指向一个方法。该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。

`Symbol.toPrimitive`被调用时，会接受一个字符串参数，表示当前运算的模式，一共有三种模式。

- Number：该场合需要转成数值
- String：该场合需要转成字符串
- Default：该场合可以转成数值，也可以转成字符串

```

1  let obj = {
2    [Symbol.toPrimitive](hint) {
3      switch (hint) {
4        case 'number':
5          return 123;
6        case 'string':
7          return 'str';
8        case 'default':
9          return 'default';
10       default:
11         throw new Error();
12     }
13   };
14 };
15
16 2 * obj // 246
17 3 + obj // '3default'
18 obj == 'default' // true
19 String(obj) // 'str'

```

## Symbol.toStringTag

这个属性可以用来定制 `[object Object]` 或 `[object Array]` 中 `object` 后面的那个大写字符串。

```

1 // 例一
2 ({[Symbol.toStringTag]: 'Foo'}.toString())
3 // "[object Foo]"
4
5 // 例二
6 class Collection {
7   get [Symbol.toStringTag]() {
8     return 'xxx';
9   }
10 }
11 let x = new Collection();
12 Object.prototype.toString.call(x) // "[object xxx]"

```

## Symbol.unscopables

对象的Symbol.unscopables属性，指向一个对象。该对象指定了使用with关键字时，哪些属性会被with环境排除。

```
▼ JavaScript

1  Array.prototype[Symbol.unscopables]
2  // {
3  //   copyWithin: true,
4  //   entries: true,
5  //   fill: true,
6  //   find: true,
7  //   findIndex: true,
8  //   includes: true,
9  //   keys: true
10 // }
11
12 Object.keys(Array.prototype[Symbol.unscopables])
13 // ['copyWithin', 'entries', 'fill', 'find', 'findIndex', 'includes', 'keys']
```

上面代码说明，数组有 7 个属性，会被with命令排除。

```
1 // 没有 unscopables 时
2 - class MyClass {
3     foo() { return 1; }
4 }
5
6 var foo = function () { return 2; };
7
8 - with (MyClass.prototype) {
9     foo(); // 1
10 }
11
12 // 有 unscopables 时
13 - class MyClass {
14     foo() { return 1; }
15 -     get [Symbol.unscopables]() {
16         return { foo: true };
17     }
18 }
19
20 var foo = function () { return 2; };
21
22 - with (MyClass.prototype) {
23     foo(); // 2
24 }
```

上面代码通过指定`Symbol.unscopables`属性，使得`with`语法块不会在当前作用域寻找`foo`属性，即`foo`将指向外层作用域的变量。

# Set 和 Map 数据结构

## Set

特点

- 成员的值都是唯一的，没有重复值
- `Set` 本身是一个构造函数，用来生成 `Set` 数据结构
- `Set` 可以接受数组/类数组为参数
- 向 `Set` 加入值时，不会发生类型转换
  - 比如 `5` 和 `"5"` 是两个值
  - `Set` 内部判断是否相等类似于 `==`，主要区别就是 `Set` 认为 `NAN` 等于自身
  - 两个对象总是不相等的

```
1 let set = new Set();
2
3 set.add({});
4 set.size // 1
5
6 set.add({});
7 set.size // 2
```

## 实例属性和方法

实例属性

- `Set.prototype.constructor`：构造函数，默认就是`Set`函数
- `Set.prototype.size`：返回`Set`实例的成员总数

`Set` 实例方法分为两类：操作方法（用于操作数据）和遍历方法（用于遍历成员）

## 操作方法

- `Set.prototype.add(value)`：添加某个值，返回 `Set` 结构本身。
- `Set.prototype.delete(value)`：删除某个值，返回一个布尔值，表示删除是否成

功。

- `Set.prototype.has(value)`：返回一个布尔值，表示该值是否为 `Set` 的成员。
- `Set.prototype.clear()`：清除所有成员，没有返回值。

`Array.from()` 方法可以将 `Set` 结构转为数组。

```
1 const items = new Set([1, 2, 3, 4, 5]);
2 const array = Array.from(items);
```

这就提供了去除数组重复成员的另一种方法。

```
1 function dedupe(array) {
2   return Array.from(new Set(array));
3 }
4
5 dedupe([1, 1, 2, 3]) // [1, 2, 3]
```

## 遍历操作

- `Set.prototype.keys()`：返回键名的遍历器
- `Set.prototype.values()`：返回键值的遍历器
- `Set.prototype.entries()`：返回键值对的遍历器
- `Set.prototype.forEach()`：使用回调函数遍历每个成员

### `keys()`, `values()`, `entries()`

`keys` 方法、`values` 方法、`entries` 方法返回的都是遍历器对象，由于 `Set` 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys` 方法和 `values` 方法的行为完全一致。

```

1  let set = new Set(['red', 'green', 'blue']);
2
3  for (let item of set.keys()) {
4      console.log(item);
5  }
6  // red
7  // green
8  // blue
9
10 for (let item of set.values()) {
11     console.log(item);
12 }
13 // red
14 // green
15 // blue
16
17 for (let item of set.entries()) {
18     console.log(item);
19 }
20 // ["red", "red"]
21 // ["green", "green"]
22 // ["blue", "blue"]

```

Set 结构的实例默认可遍历，它的默认遍历器生成函数就是它的 `values` 方法。

```

1  Set.prototype[Symbol.iterator] === Set.prototype.values
2  // true

```

这意味着，可以省略 `values` 方法，直接用 `for...of` 循环遍历 `Set`。

```

1  let set = new Set(['red', 'green', 'blue']);
2
3  for (let x of set) {
4      console.log(x);
5  }
6  // red
7  // green
8  // blue

```

## forEach()

Set 结构的实例与数组一样，也拥有 `forEach` 方法，用于对每个成员执行某种操作，没有返回值。

```
1 let set = new Set([1, 4, 9]);
2 set.forEach((value, key) => console.log(key + ' : ' + value))
3 // 1 : 1
4 // 4 : 4
5 // 9 : 9
```

JavaScript

## 遍历的应用

扩展运算符 (`...`) 内部使用 `for...of` 循环，所以也可以用于 `Set` 结构。

```
1 let set = new Set(['red', 'green', 'blue']);
2 let arr = [...set];
3 // ['red', 'green', 'blue']
```

JavaScript

扩展运算符就可以配合数组方法使用，比如实现并集 (Union)、交集 (Intersect) 和差集 (Difference)

```
1 let a = new Set([1, 2, 3]);
2 let b = new Set([4, 3, 2]);
3
4 // 并集
5 let union = new Set([...a, ...b]);
6 // Set {1, 2, 3, 4}
7
8 // 交集
9 let intersect = new Set([...a].filter(x => b.has(x)));
10 // set {2, 3}
11
12 // (a 相对于 b 的) 差集
13 let difference = new Set([...a].filter(x => !b.has(x)));
14 // Set {1}
```

JavaScript

如果想在遍历操作中，同步改变原来的 `Set` 结构，目前没有直接的方法，但有两种变通方法。一种是利用原 `Set` 结构映射出一个新的结构，然后赋值给原来的 `Set` 结构；另一种是利用

## Array.from 方法。

JavaScript |

```
1 // 方法一
2 let set = new Set([1, 2, 3]);
3 set = new Set([...set].map(val => val * 2));
4 // set的值是2, 4, 6
5
6 // 方法二
7 let set = new Set([1, 2, 3]);
8 set = new Set(Array.from(set, val => val * 2));
9 // set的值是2, 4, 6
```

# WeakSet

与 Set 类似，也是不重复的值的集合。但是，它与 Set 有两个区别。

## 语法

与 Set 方法基本类似，只是有以下区别：

- 没有 size 属性
- 没有办法遍历成员

WeakSet 不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet 的一个用处，是储存 DOM 节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

## 特性

### 成员只能是对象和Symbol值

JavaScript |

```
1 const ws = new WeakSet();
2 ws.add(1) // 报错
3 ws.add(Symbol()) // 不报错
```

作为构造函数，WeakSet 可以接受一个数组或类似数组的对象作为参数。（实际上，任何具有 Iterable 接口的对象，都可以作为 WeakSet 的参数。）该数组的所有成员，都会自动成为 WeakSet 实例对象的成员。

```
1 const a = [[1, 2], [3, 4]];
2 const ws = new WeakSet(a);
3 // WeakSet {[1, 2], [3, 4]}
```

上面代码中，`a` 是一个数组，它有两个成员，也都是数组。将 `a` 作为 WeakSet 构造函数的参数，`a` 的成员会自动成为 WeakSet 的成员。

注意，是 `a` 数组的成员成为 WeakSet 的成员，而不是 `a` 数组本身。这意味着，数组的成员只能是对象。

```
1 const b = [3, 4];
2 const ws = new WeakSet(b);
3 // Uncaught TypeError: Invalid value used in weak set(...)
```

上面代码中，数组 `b` 的成员不是对象，加入 WeakSet 就会报错。

## WeakSet 对象弱引用

WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

因此，WeakSet 适合临时存放一组对象，以及存放跟对象绑定的信息。只要这些对象在外部消失，它在 WeakSet 里面的引用就会自动消失。

# Map

## 基本用法

由于对象只能用字符串作为键，所以在很多情况无法使用。

```

1 const data = {};
2 const element = document.getElementById('myDiv');
3
4 data[element] = 'metadata';
5 data['[object HTMLDivElement]'] // "metadata"

```

上面代码原意是将一个 DOM 节点作为对象 `data` 的键，但是由于对象只接受字符串作为键名，所以 `element` 被自动转为字符串 `[object HTMLDivElement]`。

为了解决这个问题，ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串一值”的对应，Map 结构提供了“值一值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

```

1 const map = new Map([
2   ['name', '张三'],
3   ['title', 'Author']
4 ]);
5
6 map.size // 2
7 map.has('name') // true
8 map.get('name') // "张三"
9 map.has('title') // true
10 map.get('title') // "Author"

```

## Map接受数组作为参数

`Map` 构造函数接受数组作为参数，实际上执行的是下面的算法。

```

1 const items = [
2   ['name', '张三'],
3   ['title', 'Author']
4 ];
5
6 const map = new Map();
7
8 items.forEach(
9   ([key, value]) => map.set(key, value)
10 );

```

事实上，不仅仅是数组，任何具有 `Iterator` 接口、且每个成员都是一个双元素的数组的数据结构都可以当作 `Map` 构造函数的参数。这就是说，`Set` 和 `Map` 都可以用来生成新的 `Map`。

```

1 const set = new Set([
2   ['foo', 1],
3   ['bar', 2]
4 ]);
5 const m1 = new Map(set);
6 m1.get('foo') // 1
7
8 const m2 = new Map([['baz', 3]]);
9 const m3 = new Map(m2);
10 m3.get('baz') // 3

```

## Map的键与内存地址绑定

如果对同一个键多次赋值，后面的值将覆盖前面的值。

对于对象作为键，只有同一个引用的才会被视为同一个键。

```

1 const map = new Map();
2
3 map.set(['a'], 555);
4 map.get(['a']) // undefined

```

对于简单类型，`Map` 也是采用 `Set` 的判断方式

- `0` 和 `-0` 是一个键
- 布尔值 `true` 和字符串 `true` 是两个不同的键
- `undefined` 和 `null` 也是两个不同的键
- 虽然 `Nan` 不严格相等于自身，但 `Map` 将其视为同一个键

## 实例属性和操作方法

### size 属性

`size` 属性返回 `Map` 结构的成员总数。

```
1 const map = new Map();
2 map.set('foo', true);
3 map.set('bar', false);
4
5 map.size // 2
```

JavaScript |

### Map.prototype.set(key, value)

`set` 方法设置键名 `key` 对应的键值为 `value`，然后返回整个 `Map` 结构。如果 `key` 已经有值，则键值会被更新，否则就新生成该键。

```
1 const m = new Map();
2
3 m.set('edition', 6)           // 键是字符串
4 m.set(262, 'standard')       // 键是数值
5 m.set(undefined, 'nah')      // 键是 undefined
```

JavaScript |

`set` 方法返回的是当前的 `Map` 对象，因此可以采用链式写法。

```
1 let map = new Map()
2   .set(1, 'a')
3   .set(2, 'b')
4   .set(3, 'c');
```

JavaScript |

## Map.prototype.get(key)

get 方法读取 key 对应的键值，如果找不到 key，返回 undefined。

```
1 const m = new Map();
2
3 const hello = function() {console.log('hello');};
4 m.set(hello, 'Hello ES6!') // 键是函数
5
6 m.get(hello) // Hello ES6!
```

JavaScript

## Map.prototype.has(key)

has 方法返回一个布尔值，表示某个键是否在当前 Map 对象之中。

```
1 const m = new Map();
2
3 m.set('edition', 6);
4 m.set(262, 'standard');
5 m.set(undefined, 'nah');
6
7 m.has('edition') // true
8 m.has('years') // false
9 m.has(262) // true
10 m.has(undefined) // true
```

JavaScript

## Map.prototype.delete(key)

delete() 方法删除某个键，返回 true。如果删除失败，返回 false。

```
1 const m = new Map();
2 m.set(undefined, 'nah');
3 m.has(undefined) // true
4
5 m.delete(undefined)
6 m.has(undefined) // false
```

JavaScript

## Map.prototype.clear()

clear() 方法清除所有成员，没有返回值。

```
▼ JavaScript |  
1 let map = new Map();  
2 map.set('foo', true);  
3 map.set('bar', false);  
4  
5 map.size // 2  
6 map.clear()  
7 map.size // 0
```

## 遍历方法

- Map.prototype.keys() : 返回键名的遍历器。
- Map.prototype.values() : 返回键值的遍历器。
- Map.prototype.entries() : 返回所有成员的遍历器。
- Map.prototype.forEach() : 遍历 Map 的所有成员。

这里需要注意的就是Map的遍历顺序就是插入顺序。

```
1 const map = new Map([
2   ['F', 'no'],
3   ['T', 'yes'],
4 ]);
5
6 for (let key of map.keys()) {
7   console.log(key);
8 }
9 // "F"
10 // "T"
11
12 for (let value of map.values()) {
13   console.log(value);
14 }
15 // "no"
16 // "yes"
17
18 for (let item of map.entries()) {
19   console.log(item[0], item[1]);
20 }
21 // "F" "no"
22 // "T" "yes"
23
24 // 或者
25 for (let [key, value] of map.entries()) {
26   console.log(key, value);
27 }
28 // "F" "no"
29 // "T" "yes"
30
31 // 等同于使用map.entries()
32 for (let [key, value] of map) {
33   console.log(key, value);
34 }
35 // "F" "no"
36 // "T" "yes"
37
38 map.forEach(function(value, key, map) {
39   console.log("Key: %s, Value: %s", key, value);
40 });
```

## 与其他数据结构互相转换

## Map转为数组

Map 转为数组最方便的方法，就是使用扩展运算符（`...`）。

```
1 const myMap = new Map()
2   .set(true, 7)
3   .set({foo: 3}, ['abc']);
4 [...myMap]
5 // [ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]
```

JavaScript |

## 数组转为Map

将数组传入 Map 构造函数，就可以转为 Map。

```
1 new Map([
2   [true, 7],
3   [{foo: 3}, ['abc']]
4 ])
5 // Map {
6 //   true => 7,
7 //   Object {foo: 3} => ['abc']
8 // }
```

JavaScript |

## Map转为对象

如果所有 Map 的键都是字符串，它可以无损地转为对象。

```

1  function strMapToObj(strMap) {
2      let obj = Object.create(null);
3      for (let [k,v] of strMap) {
4          obj[k] = v;
5      }
6      return obj;
7  }
8
9  const myMap = new Map()
10 .set('yes', true)
11 .set('no', false);
12 strMapToObj(myMap)
13 // { yes: true, no: false }

```

如果有非字符串的键名，那么这个键名会被转成字符串，再作为对象的键名。

## 对象转为Map

对象转为 Map 可以通过 `Object.entries()`。

```

1  let obj = {"a":1, "b":2};
2  let map = new Map(Object.entries(obj));

```

此外，也可以自己实现一个转换函数。

```

1  function objToStrMap(obj) {
2      let strMap = new Map();
3      for (let k of Object.keys(obj)) {
4          strMap.set(k, obj[k]);
5      }
6      return strMap;
7  }
8
9  objToStrMap({yes: true, no: false})
10 // Map {"yes" => true, "no" => false}

```

## Map 转为 JSON

Map 转为 JSON 要区分两种情况。一种情况是，Map 的键名都是字符串，这时可以选择转为对象 JSON。

```
1 function strMapToJson(strMap) {  
2     return JSON.stringify(strMapToObj(strMap));  
3 }  
4  
5 let myMap = new Map().set('yes', true).set('no', false);  
6 strMapToJson(myMap)  
7 // '{"yes":true,"no":false}'
```

另一种情况是，Map 的键名有非字符串，这时可以选择转为数组 JSON。

```
1 function mapToArrayJson(map) {  
2     return JSON.stringify([...map]);  
3 }  
4  
5 let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);  
6 mapToArrayJson(myMap)  
7 // [[[true,7],[{"foo":3},["abc"]]]]
```

## JSON 转为 Map

JSON 转为 Map，正常情况下，所有键名都是字符串。

```
1 function jsonToStrMap(jsonStr) {  
2     return objToStrMap(JSON.parse(jsonStr));  
3 }  
4  
5 jsonToStrMap('{"yes": true, "no": false}')  
6 // Map {'yes' => true, 'no' => false}
```

但是，有一种特殊情况，整个 JSON 就是一个数组，且每个数组成员本身，又是一个有两个成员的数组。这时，它可以一一对应地转为 Map。这往往是 Map 转为数组 JSON 的逆操作。

```

1 function jsonToMap(jsonStr) {
2     return new Map(JSON.parse(jsonStr));
3 }
4
5 jsonToMap('[[true,7],[{"foo":3},["abc"]]]')
6 // Map {true => 7, Object {foo: 3} => ['abc']}

```

## WeakMap

### 含义

与Map结构类似，`WeakMap` 与 `Map` 的区别有两点。

- `WeakMap` 只接受对象（null除外）和Symbol值作为键名
- `WeakMap` 的键名所指向的对象不计入垃圾回收

所以 `WeakMap` 得使用场景就是它的键所对应的对象可能消失。比如DOM操作。

```

1 const wm = new WeakMap();
2
3 const element = document.getElementById('example');
4
5 wm.set(element, 'some information');
6 wm.get(element) // "some information"

```

上面代码中，先新建一个 `WeakMap` 实例。然后，将一个 DOM 节点作为键名存入该实例，并将一些附加信息作为键值，一起存放在 `WeakMap` 里面。这时，`WeakMap` 里面对 `element` 的引用就是弱引用，不会被计入垃圾回收机制。

也就是说，上面的 `DOM` 节点对象除了 `WeakMap` 的弱引用外，其他位置对该对象的引用一旦消除，该对象占用的内存就会被垃圾回收机制释放。`WeakMap` 保存的这个键值对，也会自动消失。

注意，`WeakMap` 弱引用的只是键名，而不是键值。键值依然是正常引用。

```

1 const wm = new WeakMap();
2 let key = {};
3 let obj = {foo: 1};
4
5 wm.set(key, obj);
6 obj = null;
7 wm.get(key)
8 // Object {foo: 1}

```

上面代码中，键值 `obj` 是正常引用。所以，即使在 `WeakMap` 外部消除了 `obj` 的引用，`WeakMap` 内部的引用依然存在。

## 语法

`WeakMap` 与 `Map` 在 API 上的区别主要是两个

- 没有遍历操作，没有 `size` 属性
- 不支持 `clear` 方法

## 使用场景

### DOM操作

```

1 let myWeakmap = new WeakMap();
2
3 myWeakmap.set(
4   document.getElementById('logo'),
5   {timesClicked: 0})
6 ;
7
8 document.getElementById('logo').addEventListener('click', function() {
9   let logoData = myWeakmap.get(document.getElementById('logo'));
10  logoData.timesClicked++;
11 }, false);

```

上面代码中，`document.getElementById('logo')` 是一个 DOM 节点，每当发生 `click` 事件，就更新一下状态。我们将这个状态作为键值放在 `WeakMap` 里，对应的键名就

是这个节点对象。一旦这个 DOM 节点删除，该状态就会自动消失，不存在内存泄漏风险。

## 部署私有属性

JavaScript |

```
1  const _counter = new WeakMap();
2  const _action = new WeakMap();
3
4  class Countdown {
5    constructor(counter, action) {
6      _counter.set(this, counter);
7      _action.set(this, action);
8    }
9    dec() {
10      let counter = _counter.get(this);
11      if (counter < 1) return;
12      counter--;
13      _counter.set(this, counter);
14      if (counter === 0) {
15        _action.get(this)();
16      }
17    }
18  }
19
20  const c = new Countdown(2, () => console.log('DONE'));
21
22  c.dec()
23  c.dec()
24  // DONE
```

上面代码中，`Countdown` 类的两个内部属性 `_counter` 和 `_action`，是实例的弱引用，所以如果删除实例，它们也就随之消失，不会造成内存泄漏。

## WeakRef

`WeakSet` 和 `WeakMap` 是基于弱引用的数据结构，[ES2021](#) 更进一步，提供了 `WeakRef` 对象，用于直接创建对象的弱引用。

```

1 let target = {};
2 let wr = new WeakRef(target);

```

上面示例中，target是原始对象，构造函数WeakRef()创建了一个基于target的新对象wr。这里，wr就是一个 WeakRef 的实例，属于对target的弱引用，垃圾回收机制不会计入这个引用，也就是说，wr的引用不会妨碍原始对象target被垃圾回收机制清除。

WeakRef 实例对象有一个deref()方法，如果原始对象存在，该方法返回原始对象；如果原始对象已经被垃圾回收机制清除，该方法返回undefined。

```

1 let target = {};
2 let wr = new WeakRef(target);
3
4 let obj = wr.deref();
5 if (obj) { // target 未被垃圾回收机制清除
6   // ...
7 }

```

上面示例中，`deref()` 方法可以判断原始对象是否已被清除。

弱引用对象的一大用处，就是作为缓存，未被清除时可以从缓存取值，一旦清除缓存就自动失效。

```

1 function makeWeakCached(f) {
2   const cache = new Map();
3   return key => {
4     const ref = cache.get(key);
5     if (ref) {
6       const cached = ref.deref();
7       if (cached !== undefined) return cached;
8     }
9
10    const fresh = f(key);
11    cache.set(key, new WeakRef(fresh));
12    return fresh;
13  };
14}
15
16 const getImageCached = makeWeakCached(getImage);

```

上面示例中，`makeWeakCached()` 用于建立一个缓存，缓存里面保存对原始文件的弱引用。

注意，标准规定，一旦使用 `WeakRef()` 创建了原始对象的弱引用，那么在本轮事件循环（event loop），原始对象肯定不会被清除，只会在后面的事件循环才会被清除。

## FinalizationRegistry

ES2021 引入了清理器注册表功能 `FinalizationRegistry`，用来指定目标对象被垃圾回收机制清除以后，所要执行的回调函数。

首先，新建一个注册表实例。

```
1 const registry = new FinalizationRegistry(heldValue => {  
2     // ....  
3});
```

上面代码中，`FinalizationRegistry()` 是系统提供的构造函数，返回一个清理器注册表实例，里面登记了所要执行的回调函数。回调函数作为 `FinalizationRegistry()` 的参数传入，它本身有一个参数 `heldValue`。

然后，注册表实例的 `register()` 方法，用来注册所要观察的目标对象。

```
1 registry.register(theObject, "some value");
```

上面示例中，`theObject` 就是所要观察的目标对象，一旦该对象被垃圾回收机制清除，注册表就会在清除完成后，调用早前注册的回调函数，并将 `some value` 作为参数（前面的 `heldValue`）传入回调函数。

注意，注册表不对目标对象 `theObject` 构成强引用，属于弱引用。因为强引用的话，原始对象就不会被垃圾回收机制清除，这就失去使用注册表的意义了。

回调函数的参数 `heldValue` 可以是任意类型的值，字符串、数值、布尔值、对象，甚至可以是 `undefined`。

最后，如果以后还想取消已经注册的回调函数，则要向 `register()` 传入第三个参数，作为标记值。这个标记值必须是对象，一般都用原始对象。接着，再使用注册表实例对象的 `unregister()` 方法取消注册。

```
1 registry.register(theObject, "some value", theObject);
2 // ...其他操作...
3 registry.unregister(theObject);
```

上面代码中，`register()` 方法的第三个参数就是标记值 `theObject`。取消回调函数时，要使用  `unregister()` 方法，并将标记值作为该方法的参数。这里 `register()` 方法对第三个参数的引用，也属于弱引用。如果没有这个参数，则回调函数无法取消。

由于回调函数被调用以后，就不再存在于注册表之中了，所以执行 `unregister()` 应该是在回调函数还没被调用之前。

# Proxy

## 概述

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

```
1 //target表示代理的目标对象  
2 //handler表示拦截方法  
3 var proxy = new Proxy(target, handler);
```

Proxy接受两个参数：

- 第一个是要代理的目标对象，如果为空对象则表示对象本身
- 第二个参数是配置对象，用于指定代理的操作对应的处理函数
  - 参数为空的话表示不设置拦截，则返回的对象指向目标对象

```
1 //第一个参数为空  
2 var proxy = new Proxy({}, {  
3   get: function(target, propKey) {  
4     return 35;  
5   }  
6 });  
7  
8 proxy.time // 35  
9 proxy.name // 35  
10 proxy.title // 35  
11  
12 //第二个参数为空  
13 var target = {};  
14 var handler = {};  
15 var proxy = new Proxy(target, handler);  
16 proxy.a = 'b';  
17 target.a // "b"
```

## 拦截器操作

以下是Proxy支持的拦截器操作

## get(target, propKey, receiver)

拦截对象属性的读取，比如 `proxy.foo` 和 `proxy['foo']`。

`get` 方法接受三个参数

- 目标对象
- 属性名
- proxy实例本身（操作行为所针对的对象）

```
▼ JavaScript |  
1 var pipe = function (value) {  
2     var funcStack = [];  
3     var oproxy = new Proxy({}, {  
4         get : function (pipeObject, fnName) {  
5             if (fnName === 'get') {  
6                 return funcStack.reduce(function (val, fn) {  
7                     return fn(val);  
8                 }, value);  
9             }  
10            funcStack.push(window[fnName]);  
11            return oproxy;  
12        }  
13    });  
14  
15    return oproxy;  
16}  
17  
18 var double = n => n * 2;  
19 var pow    = n => n * n;  
20 var reverseInt = n => n.toString().split("").reverse().join("") | 0;  
21  
22 pipe(3).double.pow.reverseInt.get; // 63
```

以上是使用读取属性的操作（`get`），实现的链式操作。

## set(target, propKey, value, receiver)

拦截对象属性的设置，比如 `proxy.foo = v` 或 `proxy['foo'] = v`，返回一个布尔值。

`set` 方法用于拦截某个属性的赋值操作，接受四个参数

- 目标对象
- 属性名
- 属性值
- proxy实例本身

## has(target, propKey)

拦截 `propKey in proxy` 的操作，返回一个布尔值。

`has()` 方法用来拦截 `HasProperty` 操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是`in`运算符。

值得注意的是，`has()` 方法拦截的是 `HasProperty` 操作，而不是 `HasOwnProperty` 操作，即`has()`方法不判断一个属性是对象自身的属性，还是继承的属性。

```
1 var handler = {
2   has (target, key) {
3     if (key[0] === '_') {
4       return false;
5     }
6     return key in target;
7   }
8 };
9 var target = { _prop: 'foo', prop: 'foo' };
10 var proxy = new Proxy(target, handler);
11 '_prop' in proxy // false
```

上面代码中，如果原对象的属性名的第一个字符是下划线，`proxy.has()` 就会返回 `false`，从而不会被 `in` 运算符发现。

另外，虽然 `for...in` 循环也用到了 `in` 运算符，但是 `has()` 拦截对 `for...in` 循环不生效。

```

1  let stu1 = {name: '张三', score: 59};
2  let stu2 = {name: '李四', score: 99};
3
4  let handler = {
5    has(target, prop) {
6      if (prop === 'score' && target[prop] < 60) {
7        console.log(`${target.name} 不及格`);
8        return false;
9      }
10     return prop in target;
11   }
12 }
13
14 let oproxy1 = new Proxy(stu1, handler);
15 let oproxy2 = new Proxy(stu2, handler);
16
17 'score' in oproxy1
18 // 张三 不及格
19 // false
20
21 'score' in oproxy2
22 // true
23
24 for (let a in oproxy1) {
25   console.log(oproxy1[a]);
26 }
27 // 张三
28 // 59
29
30 for (let b in oproxy2) {
31   console.log(oproxy2[b]);
32 }
33 // 李四
34 // 99

```

## deleteProperty(target, propKey)

拦截 `delete proxy[propKey]` 的操作，返回一个布尔值。

`deleteProperty` 方法用于拦截 `delete` 操作，如果这个方法抛出错误或者返回 `false`，当前属性就无法被 `delete` 命令删除。

## ownKeys(target)

拦截 `Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`、`for...in` 循环，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而 `Object.keys()` 的返回结果仅包括目标对象自身的可遍历属性。

`ownKeys()`方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。

- `Object.getOwnPropertyNames()`
- `Object.getOwnPropertySymbols()`
- `Object.keys()`
- `for...in` 循环

## getOwnPropertyDescriptor(target, propKey)

拦截 `Object.getOwnPropertyDescriptor(proxy, propKey)`，返回属性的描述对象。

```
1 var handler = {
2   getOwnPropertyDescriptor (target, key) {
3     if (key[0] === '_') {
4       return;
5     }
6     return Object.getOwnPropertyDescriptor(target, key);
7   }
8 };
9 var target = { _foo: 'bar', baz: 'tar' };
10 var proxy = new Proxy(target, handler);
11 Object.getOwnPropertyDescriptor(proxy, 'wat')
12 // undefined
13 Object.getOwnPropertyDescriptor(proxy, '_foo')
14 // undefined
15 Object.getOwnPropertyDescriptor(proxy, 'baz')
16 // { value: 'tar', writable: true, enumerable: true, configurable: true }
```

## defineProperty(target, propKey, propDesc)

拦截 `Object.defineProperty(proxy, propKey, propDesc)`、`Object.defineProperties(proxy, propDescs)`，返回一个布尔值。

```

1 var handler = {
2   defineProperty (target, key, descriptor) {
3     return false;
4   }
5 };
6 var target = {};
7 var proxy = new Proxy(target, handler);
8 proxy.foo = 'bar' // 不会生效

```

## preventExtensions(target)

拦截 `Object.preventExtensions(proxy)`，返回一个布尔值。

## getPrototypeOf(target)

拦截 `Object.getPrototypeOf(proxy)`，返回一个对象。

`getPrototypeOf()` 方法主要用来拦截获取对象原型。具体来说，拦截下面这些操作。

- `Object.prototype.__proto__`
- `Object.prototype.isPrototypeOf()`
- `Object.getPrototypeOf()`
- `Reflect.getPrototypeOf()`
- `instanceof`

```

1 var proto = {};
2 var p = new Proxy({}, {
3   getPrototypeOf(target) {
4     return proto;
5   }
6 });
7 Object.getPrototypeOf(p) === proto // true

```

## isExtensible(target)

拦截 `Object.isExtensible(proxy)`，返回一个布尔值。

## setPrototypeOf(target, proto)

拦截 `Object.setPrototypeOf(proxy, proto)`，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。

## apply(target, object, args)

拦截 `Proxy` 实例作为函数调用的操作，比如 `proxy(...args)`、`proxy.call(object, ...args)`、`proxy.apply(...)`。

```
1 var target = function () { return 'I am the target'; };
2 var handler = {
3   apply: function () {
4     return 'I am the proxy';
5   }
6 };
7
8 var p = new Proxy(target, handler);
9
10 p()
11 // "I am the proxy"
```

变量p是Proxy的实例，当作为函数调用时，就会被拦截。

## construct(target, args)

拦截 `Proxy` 实例作为构造函数调用的操作，比如 `new proxy(...args)`。

`construct()` 方法用于拦截 `new` 命令，可以接受三个参数：

- 目标对象
- 构造函数的参数数组
- 创造实例对象时，`new` 命令作用的构造函数

```

1 const p = new Proxy(function () {}, {
2   construct: function(target, args) {
3     console.log('called: ' + args.join(', '));
4     return { value: args[0] * 10 };
5   }
6 });
7
8 (new p(1)).value
9 // "called: 1"
10 // 10

```

## Proxy.revocable()

`Proxy.revocable()` 方法返回一个可取消的 Proxy 实例。

```

1 let target = {};
2 let handler = {};
3
4 let {proxy, revoke} = Proxy.revocable(target, handler);
5
6 proxy.foo = 123;
7 proxy.foo // 123
8
9 revoke();
10 proxy.foo // TypeError: Revoked

```

## this问题

关于 proxy 的 this 指向有两个问题

- 目标对象内部的 `this` 指向 `proxy` 代理
- 拦截函数内部 `this` 指向 `handler` 对象

```

1 const target = {
2   m: function () {
3     console.log(this === proxy);
4   }
5 };
6 const handler = {};
7
8 const proxy = new Proxy(target, handler);
9
10 target.m() // false
11 proxy.m() // true

```

## Date对象this指向错误

```

1 const target = new Date();
2 const handler = {};
3 const proxy = new Proxy(target, handler);
4
5 proxy.getDate();
6 // TypeError: this is not a Date object.
7
8 const target = new Date('2015-01-01');
9 const handler = {
10   get(target, prop) {
11     if (prop === 'getDate') {
12       return target.getDate.bind(target);
13     }
14     return Reflect.get(target, prop);
15   }
16 };
17 const proxy = new Proxy(target, handler);
18
19 proxy.getDate() // 1

```

▼ 拦截函数内部this指向handler对象

JavaScript |

```
1 const handler = {
2   get: function (target, key, receiver) {
3     console.log(this === handler);
4     return 'Hello, ' + key;
5   },
6   set: function (target, key, value) {
7     console.log(this === handler);
8     target[key] = value;
9     return true;
10  }
11};
12
13 const proxy = new Proxy({}, handler);
14
15 proxy.foo
16 // true
17 // Hello, foo
18
19 proxy.foo = 1
20 // true
```

# Reflect

---

## 概述

`Reflect` 对象与 `Proxy` 对象一样，也是 ES6 为了操作对象而提供的新 API。`Reflect` 对象的设计目的有这样几个。

- 将 `Object` 对象的一些明显属于语言内部的方法（比如 `Object.defineProperty`），放到 `Reflect` 对象上。现阶段，某些方法同时在 `Object` 和 `Reflect` 对象上部署，未来的新方法将只部署在 `Reflect` 对象上。也就是说，从 `Reflect` 对象上可以拿到语言内部的方法。
- 修改某些 `Object` 方法的返回结果，让其变得更合理。比如，`Object.defineProperty(obj, name, desc)` 在无法定义属性时，会抛出一个错误，而 `Reflect.defineProperty(obj, name, desc)` 则会返回`false`。
- 让 `Object` 操作都变成函数行为。某些 `Object` 操作是命令式，比如 `name in obj` 和 `delete obj[name]`，而 `Reflect.has(obj, name)` 和 `Reflect.deleteProperty(obj, name)` 让它们变成了函数行为。
- `Reflect` 对象的方法与 `Proxy` 对象的方法一一对应，只要是 `Proxy` 对象的方法，就能在 `Reflect` 对象上找到对应的方法。这就让 `Proxy` 对象可以方便地调用对应的 `Reflect` 方法，完成默认行为，作为修改行为的基础。也就是说，不管 `Proxy` 怎么修改默认行为，你总可以在 `Reflect` 上获取默认行为。

## 静态方法

对应proxy可以拦截的13个方法，`Reflect`也有这13个方法：

- `Reflect.apply(target, thisArg, args)`
- `Reflect.construct(target, args)`
- `Reflect.get(target, name, receiver)`
- `Reflect.set(target, name, value, receiver)`
- `Reflect.defineProperty(target, name, desc)`
- `Reflect.deleteProperty(target, name)`
- `Reflect.has(target, name)`

- `Reflect.ownKeys(target)`
- `Reflect.isExtensible(target)`
- `Reflect.preventExtensions(target)`
- `Reflect.getOwnPropertyDescriptor(target, name)`
- `Reflect.getPrototypeOf(target)`
- `Reflect.setPrototypeOf(target, prototype)`

# Promise

## 概述

`Promise` 是异步编程的一种解决方案，`Promise` 对象有以下特点：

- 对象状态不受外界影响
- 状态一旦改变，就不会再变。
  - pending → fulfilled
  - pending → rejected

## 用法

`Promise` 是一个构造函数，用来生成 `Promise` 实例。

```
▼                                     JavaScript |  
1 const promise = new Promise(function(resolve, reject) {  
2     // ... some code  
3  
4     if /* 异步操作成功 */{  
5         resolve(value);  
6     } else {  
7         reject(error);  
8     }  
9});
```

如果调用 `resolve` 函数和 `reject` 函数时带有参数，那么它们的参数会被传递给回调函数。`reject` 函数的参数通常是 `Error` 对象的实例，表示抛出的错误；`resolve` 函数的参数除了正常的值以外，还可能是另一个 `Promise` 实例。

```

1 const p1 = new Promise(function (resolve, reject) {
2   setTimeout(() => reject(new Error('fail')), 3000)
3 }
4
5 const p2 = new Promise(function (resolve, reject) {
6   setTimeout(() => resolve(p1), 1000)
7 }
8
9 p2
10 .then(result => console.log(result))
11 .catch(error => console.log(error))
12 // Error: fail

```

上面代码中，`p1`是一个 `Promise`，3 秒之后变为 `rejected`。`p2`的状态在 1 秒之后改变，`resolve` 方法返回的是`p1`。由于`p2`返回的是另一个 `Promise`，导致`p2`自己的状态无效了，由`p1`的状态决定`p2`的状态。所以，后面的 `then` 语句都变成针对后者（`p1`）。又过了 2 秒，`p1`变为 `rejected`，导致触发 `catch` 方法指定的回调函数。

## Promise.prototype.then()

`Promise` 实例具有 `then` 方法，也就是说，`then` 方法是定义在原型对象 `Promise.prototype` 上的。它的作用是为 `Promise` 实例添加状态改变时的回调函数。

```

1getJSON("/post/1.json").then(function(post) {
2  return getJSON(post.commentURL);
3}).then(function (comments) {
4  console.log("resolved: ", comments);
5}, function (err){
6  console.log("rejected: ", err);
7});

```

上面代码中，第一个 `then` 方法指定的回调函数，返回的是另一个 `Promise` 对象。这时，第二个 `then` 方法指定的回调函数，就会等待这个新的 `Promise` 对象状态发生变化。如果变为 `resolved`，就调用第一个回调函数，如果状态变为 `rejected`，就调用第二个回调函数。

## Promise.prototype.catch()

`Promise.prototype.catch()` 方法是 `.then(null, rejection)` 或 `.then(undefined, rejection)` 的别名，用于指定发生错误时的回调函数。

```
1 p.then((val) => console.log('fulfilled:', val))
2   .catch((err) => console.log('rejected', err));
3
4 // 等同于
5 p.then((val) => console.log('fulfilled:', val))
6   .then(null, (err) => console.log("rejected:", err));
```

`promise` 抛出错误就会被 `catch` 捕获。

```
1 // 写法一
2 const promise = new Promise(function(resolve, reject) {
3   try {
4     throw new Error('test');
5   } catch(e) {
6     reject(e);
7   }
8 });
9 promise.catch(function(error) {
10   console.log(error);
11 });
12
13 // 写法二
14 const promise = new Promise(function(resolve, reject) {
15   reject(new Error('test'));
16 });
17 promise.catch(function(error) {
18   console.log(error);
19 });
```

一般来说，不要在 `then()` 方法里面定义 `Reject` 状态的回调函数（即 `then` 的第二个参数），总是使用 `catch` 方法。

```

1 // bad
2 promise
3 .then(function(data) {
4     // success
5 }, function(err) {
6     // error
7 });
8
9 // good
10 promise
11 .then(function(data) { //cb
12     // success
13 })
14 .catch(function(err) {
15     // error
16 });

```

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面`then`方法执行中的错误，也更接近同步的写法（`try/catch`）。因此，建议总是使用`catch()`方法，而不使用`then()`方法的第二个参数。

注意：如果没有`catch`方法，`promise`抛出的错误不会传递到外层代码。

```

1 const someAsyncThing = function() {
2     return new Promise(function(resolve, reject) {
3         // 下面一行会报错，因为x没有声明
4         resolve(x + 2);
5     });
6 }
7
8 someAsyncThing().then(function() {
9     console.log('everything is great');
10 });
11
12 setTimeout(() => { console.log(123) }, 2000);
13 // Uncaught (in promise) ReferenceError: x is not defined
14 // 123

```

## Promise.prototype.finally()

`finally()` 方法用于指定不管 `Promise` 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

```
1 promise
2 .finally(() => {
3     // 语句
4 });
5
6 // 等同于
7 promise
8 .then(
9   result => {
10     // 语句
11     return result;
12 },
13   error => {
14     // 语句
15     throw error;
16 }
17 );
```

JavaScript

finally实现

JavaScript

```
1 Promise.prototype.finally = function (callback) {
2     let P = this.constructor;
3     return this.then(
4         value => P.resolve(callback()).then(() => value),
5         reason => P.resolve(callback()).then(() => { throw reason })
6     );
7 }
```

上面代码中，不管前面的 `Promise` 是 `fulfilled` 还是 `rejected`，都会执行回调函数 `callback`。

根据上面代码可以知道 `finally` 总是返回原来的值。

```

1 // resolve 的值是 undefined
2 Promise.resolve(2).then(() => {}, () => {})
3
4 // resolve 的值是 2
5 Promise.resolve(2).finally(() => {})
6
7 // reject 的值是 undefined
8 Promise.reject(3).then(() => {}, () => {})
9
10 // reject 的值是 3
11 Promise.reject(3).finally(() => {})

```

## Promise.all()

用于将多个Promise实例包装成一个新的Promise实例。

```
1 const p = Promise.all([p1, p2, p3]);
```

上面代码中，`Promise.all()`方法接受一个数组作为参数，`p1`、`p2`、`p3`都是`Promise`实例，如果不是，就会先调用下面讲到的`Promise.resolve`方法，将参数转为`Promise`实例，再进一步处理。另外，`Promise.all()`方法的参数可以不是数组，但必须具有`Iterator`接口，且返回的每个成员都是`Promise`实例。

`p`的状态由`p1`、`p2`、`p3`决定，分成两种情况。

- (1) 只有`p1`、`p2`、`p3`的状态都变成`fulfilled`，`p`的状态才会变成`fulfilled`，此时`p1`、`p2`、`p3`的返回值组成一个数组，传递给`p`的回调函数。
- (2) 只要`p1`、`p2`、`p3`之中有一个被`rejected`，`p`的状态就变成`rejected`，此时第一个被`reject`的实例的返回值，会传递给`p`的回调函数。

```

1 // 生成一个Promise对象的数组
2 const promises = [2, 3, 5, 7, 11, 13].map(function (id) {
3   return getJSON('/post/' + id + ".json");
4 });
5
6 Promise.all(promises).then(function (posts) {
7   // ...
8 }).catch(function(reason){
9   // ...
10 });

```

注意，如果作为参数的 `Promise` 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected`，并不会触发 `Promise.all()` 的 `catch` 方法。

```

1 const p1 = new Promise((resolve, reject) => {
2   resolve('hello');
3 })
4 .then(result => result)
5 .catch(e => e);
6
7 const p2 = new Promise((resolve, reject) => {
8   throw new Error('报错了');
9 })
10 .then(result => result)
11 .catch(e => e);
12
13 Promise.all([p1, p2])
14 .then(result => console.log(result))
15 .catch(e => console.log(e));
16 // ["hello", Error: 报错了"]

```

上面代码中，`p1`会 `resolved`，`p2`首先会 `rejected`，但是`p2`有自己的 `catch` 方法，该方法返回的是一个新的 `Promise` 实例，`p2`指向的实际上是这个实例。该实例执行完 `catch` 方法后，也会变成 `resolved`，导致 `Promise.all()` 方法参数里面的两个实例都会 `resolved`，因此会调用 `then` 方法指定的回调函数，而不会调用 `catch` 方法指定的回调函数。

## Promise.race()

Promise.race()方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

```
1 const p = Promise.race([p1, p2, p3]);
```

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

## Promise.allSettled()

Promise.allSettled()方法，用来确定一组异步操作是否都结束了（不管成功或失败）。所以，它的名字叫做”Settled“，包含了”fulfilled“和”rejected“两种情况。

该方法返回的新的 Promise 实例，一旦发生状态变更，状态总是fulfilled，不会变成rejected。状态变成fulfilled后，它的回调函数会接收到一个数组作为参数，该数组的每个成员对应前面数组的每个 Promise 对象。

```
1 const resolved = Promise.resolve(42);
2 const rejected = Promise.reject(-1);
3
4 const allSettledPromise = Promise.allSettled([resolved, rejected]);
5
6 allSettledPromise.then(function (results) {
7   console.log(results);
8 });
9 // [
10 //   { status: 'fulfilled', value: 42 },
11 //   { status: 'rejected', reason: -1 }
12 // ]
```

## Promise.any()

ES2021 引入了Promise.any()方法。该方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例返回。

```

1 Promise.any([
2   fetch('https://v8.dev/').then(() => 'home'),
3   fetch('https://v8.dev/blog').then(() => 'blog'),
4   fetch('https://v8.dev/docs').then(() => 'docs')
5 ]).then((first) => { // 只要有一个 fetch() 请求成功
6   console.log(first);
7 }).catch((error) => { // 所有三个 fetch() 全部请求失败
8   console.log(error);
9 });

```

只要参数实例有一个变成`fulfilled`状态，包装实例就会变成`fulfilled`状态；如果所有参数实例都变成`rejected`状态，包装实例就会变成`rejected`状态。

`Promise.any()`跟`Promise.race()`方法很像，只有一点不同，就是`Promise.any()`不会因为某个`Promise`变成`rejected`状态而结束，必须等到所有参数`Promise`变成`rejected`状态才会结束。

## Promise.resolve()

有时需要将现有对象转为`Promise`对象，`Promise.resolve()`方法就起到这个作用。

```

1 Promise.resolve('foo')
2 // 等价于
3 new Promise(resolve => resolve('foo'))

```

`Promise.resolve()`方法的参数分成四种情况。

### 参数是一个`Promise`实例

如果参数是`Promise`实例，那么`Promise.resolve`将不做任何修改、原封不动地返回这个实例。

### 参数是一个`thenable`对象

`thenable`对象指的是具有`then`方法的对象，比如下面这个对象。

```
1 let thenable = {  
2   then: function(resolve, reject) {  
3     resolve(42);  
4   }  
5};
```

JavaScript |

Promise.resolve()方法会将这个对象转为 Promise 对象，然后就立即执行thenable对象的then()方法。

```
1 let thenable = {  
2   then: function(resolve, reject) {  
3     resolve(42);  
4   }  
5};  
6  
7 let p1 = Promise.resolve(thenable);  
8 p1.then(function (value) {  
9   console.log(value); // 42  
10});
```

JavaScript |

上面代码中，thenable对象的then()方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then()方法指定的回调函数，输出42。

### 参数不是具有then()方法的对象，或根本就不是对象

如果参数是一个原始值，或者是一个不具有then()方法的对象，则Promise.resolve()方法返回一个新的 Promise 对象，状态为resolved。

```
1 const p = Promise.resolve('Hello');  
2  
3 p.then(function (s) {  
4   console.log(s)  
5});  
6 // Hello
```

JavaScript |

上面代码生成一个新的 Promise 对象的实例p。由于字符串Hello不属于异步操作（判断方法是字符串对象不具有 then 方法），返回 Promise 实例的状态从一生成就是resolved，所以回调函数会立即执行。Promise.resolve()方法的参数，会同时传给回调函数。

## 不带有任何参数

Promise.resolve()方法允许调用时不带参数，直接返回一个resolved状态的 Promise 对象。

所以，如果希望得到一个 Promise 对象，比较方便的方法就是直接调用Promise.resolve()方法。

```
1 const p = Promise.resolve();
2
3 p.then(function () {
4   // ...
5});
```

上面代码的变量p就是一个 Promise 对象。

## Promise.reject()

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

```
1 const p = Promise.reject('出错了');
2 // 等同于
3 const p = new Promise((resolve, reject) => reject('出错了'))
4
5 p.then(null, function (s) {
6   console.log(s)
7 });
8 // 出错了
```

上面代码生成一个 Promise 对象的实例p，状态为rejected，回调函数会立即执行。

Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。

```
1 Promise.reject('出错了')
2 .catch(e => {
3   console.log(e === '出错了')
4 })
5 // true
```

## Promise.try() 【提案状态】

对于不确定的函数，让同步函数同步执行，异步函数异步执行。

JavaScript |

```
1 const f = () => console.log('now');
2 Promise.try(f);
3 console.log('next');
4 // now
5 // next
```

# Iterator 和 for...of 循环

## Iterator（遍历器）的概念

JavaScript 原有的表示“集合”的数据结构

- 数组 (Array)
- 对象 (Object)
- Map
- Set

需要一种统一的接口机制来处理不用的数据结构，遍历器 (Iterator) 就是这种机制。

Iterator 的作用有三个：

- 一是为各种数据结构，提供一个统一的、简便的访问接口；
- 二是使得数据结构的成员能够按某种次序排列；
- 三是ES6创造了一种新的遍历命令for...of循环，**Iterator**接口主要供**for...of**消费。

```
1 interface Iterable {
2     [Symbol.iterator](): Iterator,
3 }
4
5 interface Iterator {
6     next(value?: any): IterationResult,
7 }
8
9 interface IterationResult {
10    value: any,
11    done: boolean,
12 }
```

TypeScript

## 默认 Iterator 接口

一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是“可遍历的” (iterable)。这种数据结构就可以被**for...of**循环遍历。

原生具备 Iterator 接口的数据结构如下，这些数据结构都原生部署了Symbol.iterator属性。

- Array
- Map
- Set
- String
- TypedArray
- 函数的 arguments 对象
- NodeList 对象

## 调用 Iterator 接口的场合

### 解构赋值

对数组和 Set 结构进行解构赋值时，会默认调用Symbol.iterator方法。

```
1 let set = new Set().add('a').add('b').add('c');
2
3 let [x,y] = set;
4 // x='a'; y='b'
5
6 let [first, ...rest] = set;
7 // first='a'; rest=['b','c'];
```

TypeScript |

### 扩展运算符

扩展运算符 (...) 也会调用默认的 Iterator 接口。

```
TypeScript |
```

```
1 // 例一
2 var str = 'hello';
3 [...str] // ['h','e','l','l','o']
4
5 // 例二
6 let arr = ['b', 'c'];
7 ['a', ...arr, 'd']
8 // ['a', 'b', 'c', 'd']
```

## yield\*

```
TypeScript |
```

```
1 let generator = function* () {
2     yield 1;
3     yield* [2,3,4];
4     yield 5;
5 };
6
7 var iterator = generator();
8
9 iterator.next() // { value: 1, done: false }
10 iterator.next() // { value: 2, done: false }
11 iterator.next() // { value: 3, done: false }
12 iterator.next() // { value: 4, done: false }
13 iterator.next() // { value: 5, done: false }
14 iterator.next() // { value: undefined, done: true }
```

## 其他

- for...of
- Array.from()
- Map(), Set(), WeakMap(), WeakSet() (比如new Map([['a',1],['b',2]]))
- Promise.all()
- Promise.race()

# Iterator 接口与 Generator 函数

Symbol.iterator()方法的最简单实现

```
1 let myIterable = {
2   [Symbol.iterator]: function* () {
3     yield 1;
4     yield 2;
5     yield 3;
6   }
7 };
8 [...myIterable] // [1, 2, 3]
9
10 // 或者采用下面的简洁写法
11
12 let obj = {
13   * [Symbol.iterator]() {
14     yield 'hello';
15     yield 'world';
16   }
17 };
18
19 for (let x of obj) {
20   console.log(x);
21 }
22 // "hello"
23 // "world"
```

# React中，不要使用renderXXX,要使用函数式组件

```
▼ JavaScript |  
1 // bad  
2 -  renderHeader = () => {  
3     return (<div />)  
4 }  
5 -  renderBody = () => {  
6     return (<div />)  
7 }  
8 -  renderFooter = () => {  
9     return (<div />)  
10 }  
11 - render(){  
12     return(  
13         <div>  
14             renderHeader()  
15             renderBody()  
16             renderFooter()  
17         </div>  
18     )  
19 }  
20
```

```

1 // good
2 function RenderHeader(props) = {
3     return (<div />)
4 }
5 function RenderBody(props) = {
6     return (<div />)
7 }
8 function RenderFooter(props) = {
9     return (<div />)
10 }
11 class Component extends React.Component<iProps, iState>{
12     render () {
13         return(
14             <div>
15                 <RenderHeader />
16                 <RenderBody />
17                 <RenderFooter />
18             </div>
19         )
20     }
21 }
22

```

在React中，第二段代码更规范。原因如下：

- 组件拆分：**第二段代码将不同的部分（Header、Body、Footer）拆分成独立的函数组件。这样做好处是提高了代码的可读性和可维护性。每个组件负责自己的部分，使得代码更加模块化。
- 复用性：**将组件拆分后，这些组件可以在其他地方被重用。如果其他地方需要渲染头部、主体或底部，可以直接使用这些组件，而不需要重新编写代码。
- 类型注解：**第二段代码中使用了类型注解（iProps 和 iState），这表明这个类组件是类型安全的。类型注解有助于减少运行时错误，并使代码更易于理解和维护。
- 性能优化：**在第一段代码中，renderHeader、renderBody 和 renderFooter 是作为类方法调用的，这意味着每次组件渲染时都会创建这些函数的实例。而在第二段代码中，这些函数组件只会被创建一次，然后在需要的地方被引用，这样可以减少不必要的渲染和内存使用。
- 符合React设计哲学：**React鼓励开发者使用组件的方式来构建用户界面。第二段代码更符合这一设计哲学，它将界面拆分成多个小组件，而不是将所有逻辑放在一个类组件中。

# 少用undefined

在JavaScript中，`undefined` 并不是一个保留字，这意味着你可以在代码中给它赋值，从而改变它的原始意义。例如：

javascript

```
1 var undefined = 'hello';
2 console.log(undefined); // 输出: 'hello'
```

在这段代码中，`undefined` 被重新定义为一个字符串，这显然会破坏代码的预期行为，因为它不再代表原始的“未定义”概念。

为了确保我们能够获取到原始的“未定义”值，而不受到外部干扰，JavaScript社区提出了一种使用`void`操作符的做法。`void`操作符会执行一个表达式，并始终返回`undefined`，无论该表达式的值是什么。因此，`void 0`成为了获取原始`undefined`值的一种常见做法，因为`void 0`总是返回`undefined`，而`0`是一个简单且执行成本很低的表达式。

使用`void 0`而不是`undefined`可以确保你的代码不会受到环境中`undefined`被重新定义的影响，从而使代码更加健壮和可靠。这是在JavaScript社区中推崇的一种最佳实践。