# Symbol

## 概述

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。

Symbol 值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

let s = Symbol();

typeof s

// "symbol"

上面代码中，变量s就是一个独一无二的值。typeof运算符的结果，表明变量s是 Symbol 数据类型，而不是字符串之类的其他类型。

注意，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。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

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，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。

let s1 = Symbol('foo');let s2 = Symbol('bar');

s1 // Symbol(foo)s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"s2.toString() // "Symbol(bar)"

上面代码中，s1和s2是两个 Symbol 值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。有了参数以后，就等于为它们加上了描述，输出的时候就能够分清，到底是哪一个值。

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。

const obj = {

toString() {

return 'abc';

}};

const sym = Symbol(obj);

sym // Symbol(abc)

注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的。

// 没有参数的情况let s1 = Symbol();let s2 = Symbol();

s1 === s2 // false

// 有参数的情况let s1 = Symbol('foo');let s2 = Symbol('foo');

s1 === s2 // false

上面代码中，s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

Symbol 值不能与其他类型的值进行运算，会报错。

let sym = Symbol('My symbol');

"your symbol is " + sym

// TypeError: can't convert symbol to string`your symbol is ${sym}`

// TypeError: can't convert symbol to string

但是，Symbol 值可以显式转为字符串。

let sym = Symbol('My symbol');

String(sym) // 'Symbol(My symbol)'sym.toString() // 'Symbol(My symbol)'

另外，Symbol 值也可以转为布尔值，但是不能转为数值。

let sym = Symbol();Boolean(sym) // true!sym // falseif (sym) {

// ...}

Number(sym) // TypeErrorsym + 2 // TypeError

## 作为属性名的 Symbol

由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

let mySymbol = Symbol();

// 第一种写法let a = {};

a[mySymbol] = 'Hello!';

// 第二种写法let a = {

[mySymbol]: 'Hello!'};

// 第三种写法let a = {};

Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上写法都得到同样结果a[mySymbol] // "Hello!"

上面代码通过方括号结构和Object.defineProperty，将对象的属性名指定为一个 Symbol 值。

注意，Symbol 值作为对象属性名时，不能用点运算符。

const mySymbol = Symbol();

const a = {};

a.mySymbol = 'Hello!';

a[mySymbol] // undefineda['mySymbol'] // "Hello!"

上面代码中，因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的那个值，导致a的属性名实际上是一个字符串，而不是一个 Symbol 值。

同理，在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在方括号之中。

let s = Symbol();

let obj = {

[s]: function (arg) { ... }};

obj[s](123);

上面代码中，如果s不放在方括号中，该属性的键名就是字符串s，而不是s所代表的那个 Symbol 值。

采用增强的对象写法，上面代码的obj对象可以写得更简洁一些。

let obj = {

[s](arg) { ... }};

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。

const log = {};

log.levels = {

DEBUG: Symbol('debug'),

INFO: Symbol('info'),

WARN: Symbol('warn')};

console.log(log.levels.DEBUG, 'debug message');

console.log(log.levels.INFO, 'info message');

下面是另外一个例子。

const COLOR\_RED = Symbol();

const COLOR\_GREEN = Symbol();

function getComplement(color) {

switch (color) {

case COLOR\_RED:

return COLOR\_GREEN;

case COLOR\_GREEN:

return COLOR\_RED;

default:

throw new Error('Undefined color');

}}

常量使用 Symbol 值最大的好处，就是其他任何值都不可能有相同的值了，因此可以保证上面的switch语句会按设计的方式工作。

还有一点需要注意，Symbol 值作为属性名时，该属性还是公开属性，不是私有属性。

## 实例：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。

function getArea(shape, options) {

let area = 0;

switch (shape) {

case 'Triangle': // 魔术字符串 area = .5 \* options.width \* options.height;

break;

/\* ... more code ... \*/

}

return area;}

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。

const shapeType = {

triangle: 'Triangle'};

function getArea(shape, options) {

let area = 0;

switch (shape) {

case shapeType.triangle:

area = .5 \* options.width \* options.height;

break;

}

return area;}

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。

const shapeType = {

triangle: Symbol()};

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

Symbol 作为属性名，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。

Object.getOwnPropertySymbols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。

const obj = {};let a = Symbol('a');let b = Symbol('b');

obj[a] = 'Hello';

obj[b] = 'World';

const objectSymbols = Object.getOwnPropertySymbols(obj);

objectSymbols

// [Symbol(a), Symbol(b)]

下面是另一个例子，Object.getOwnPropertySymbols方法与for...in循环、Object.getOwnPropertyNames方法进行对比的例子。

const obj = {};

let foo = Symbol("foo");

Object.defineProperty(obj, foo, {

value: "foobar",});

for (let i in obj) {

console.log(i); // 无输出}

Object.getOwnPropertyNames(obj)

// []

Object.getOwnPropertySymbols(obj)

// [Symbol(foo)]

上面代码中，使用Object.getOwnPropertyNames方法得不到Symbol属性名，需要使用Object.getOwnPropertySymbols方法。

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。

let obj = {

[Symbol('my\_key')]: 1,

enum: 2,

nonEnum: 3};

Reflect.ownKeys(obj)

// ["enum", "nonEnum", Symbol(my\_key)]

由于以 Symbol 值作为名称的属性，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。

let size = Symbol('size');

class Collection {

constructor() {

this[size] = 0;

}

add(item) {

this[this[size]] = item;

this[size]++;

}

static sizeOf(instance) {

return instance[size];

}}

let x = new Collection();

Collection.sizeOf(x) // 0

x.add('foo');

Collection.sizeOf(x) // 1

Object.keys(x) // ['0']Object.getOwnPropertyNames(x) // ['0']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，否则就新建并返回一个以该字符串为名称的 Symbol 值。

let s1 = Symbol.for('foo');let s2 = Symbol.for('foo');

s1 === s2 // true

上面代码中，s1和s2都是 Symbol 值，但是它们都是同样参数的Symbol.for方法生成的，所以实际上是同一个值。

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。

Symbol.for("bar") === Symbol.for("bar")

// trueSymbol("bar") === Symbol("bar")

// false

上面代码中，由于Symbol()写法没有登记机制，所以每次调用都会返回一个不同的值。

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

let s1 = Symbol.for("foo");

Symbol.keyFor(s1) // "foo"let s2 = Symbol("foo");

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。

需要注意的是，Symbol.for为 Symbol 值登记的名字，是全局环境的，可以在不同的 iframe 或 service worker 中取到同一个值。

iframe = document.createElement('iframe');

iframe.src = String(window.location);

document.body.appendChild(iframe);

iframe.contentWindow.Symbol.for('foo') === Symbol.for('foo')

// true

上面代码中，iframe 窗口生成的 Symbol 值，可以在主页面得到。

## 实例：模块的 Singleton 模式

Singleton 模式指的是调用一个类，任何时候返回的都是同一个实例。

对于 Node 来说，模块文件可以看成是一个类。怎么保证每次执行这个模块文件，返回的都是同一个实例呢？

很容易想到，可以把实例放到顶层对象global。

// mod.jsfunction A() {

this.foo = 'hello';}

if (!global.\_foo) {

global.\_foo = new A();}

module.exports = global.\_foo;

然后，加载上面的mod.js。

const a = require('./mod.js');

console.log(a.foo);

上面代码中，变量a任何时候加载的都是A的同一个实例。

但是，这里有一个问题，全局变量global.\_foo是可写的，任何文件都可以修改。

global.\_foo = { foo: 'world' };

const a = require('./mod.js');

console.log(a.foo);

上面的代码，会使得加载mod.js的脚本都失真。

为了防止这种情况出现，我们就可以使用 Symbol。

// mod.jsconst FOO\_KEY = Symbol.for('foo');

function A() {

this.foo = 'hello';}

if (!global[FOO\_KEY]) {

global[FOO\_KEY] = new A();}

module.exports = global[FOO\_KEY];

上面代码中，可以保证global[FOO\_KEY]不会被无意间覆盖，但还是可以被改写。

global[Symbol.for('foo')] = { foo: 'world' };

const a = require('./mod.js');

如果键名使用Symbol方法生成，那么外部将无法引用这个值，当然也就无法改写。

// mod.jsconst FOO\_KEY = Symbol('foo');

// 后面代码相同 ……

上面代码将导致其他脚本都无法引用FOO\_KEY。但这样也有一个问题，就是如果多次执行这个脚本，每次得到的FOO\_KEY都是不一样的。虽然 Node 会将脚本的执行结果缓存，一般情况下，不会多次执行同一个脚本，但是用户可以手动清除缓存，所以也不是绝对可靠。

## 内置的 Symbol 值

除了定义自己使用的 Symbol 值以外，ES6 还提供了 11 个内置的 Symbol 值，指向语言内部使用的方法。

### 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)。

class MyClass {

[Symbol.hasInstance](foo) {

return foo instanceof Array;

}}

[1, 2, 3] instanceof new MyClass() // true

上面代码中，MyClass是一个类，new MyClass()会返回一个实例。该实例的Symbol.hasInstance方法，会在进行instanceof运算时自动调用，判断左侧的运算子是否为Array的实例。

下面是另一个例子。

class Even {

static [Symbol.hasInstance](obj) {

return Number(obj) % 2 === 0;

}}

// 等同于const Even = {

[Symbol.hasInstance](obj) {

return Number(obj) % 2 === 0;

}};

1 instanceof Even // false2 instanceof Even // true12345 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。

let arr1 = ['c', 'd'];['a', 'b'].concat(arr1, 'e') // ['a', 'b', 'c', 'd', 'e']arr1[Symbol.isConcatSpreadable] // undefinedlet arr2 = ['c', 'd'];

arr2[Symbol.isConcatSpreadable] = false;['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。

let obj = {length: 2, 0: 'c', 1: 'd'};['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']

obj[Symbol.isConcatSpreadable] = true;['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']

Symbol.isConcatSpreadable属性也可以定义在类里面。

class A1 extends Array {

constructor(args) {

super(args);

this[Symbol.isConcatSpreadable] = true;

}}

class A2 extends Array {

constructor(args) {

super(args);

}

get [Symbol.isConcatSpreadable] () {

return false;

}}let a1 = new A1();

a1[0] = 3;

a1[1] = 4;let a2 = new A2();

a2[0] = 5;

a2[1] = 6;[1, 2].concat(a1).concat(a2)

// [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。

class MyArray extends Array {}

const a = new MyArray(1, 2, 3);

const b = a.map(x => x);

const c = a.filter(x => x > 1);

b instanceof MyArray // truec 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。

class MyArray extends Array {

static get [Symbol.species]() { return Array; }}

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。

static get [Symbol.species]() {

return this;}

现在，再来看前面的例子。

class MyArray extends Array {

static get [Symbol.species]() { return Array; }}

const a = new MyArray();

const b = a.map(x => x);

b instanceof MyArray // falseb 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。

class T1 extends Promise {}

class T2 extends Promise {

static get [Symbol.species]() {

return Promise;

}}

new T1(r => r()).then(v => v) instanceof T1 // truenew 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。

String.prototype.match(regexp)

// 等同于regexp[Symbol.match](this)

class MyMatcher {

[Symbol.match](string) {

return 'hello world'.indexOf(string);

}}

'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。

String.prototype.replace(searchValue, replaceValue)

// 等同于searchValue[Symbol.replace](this, replaceValue)

下面是一个例子。

const x = {};

x[Symbol.replace] = (...s) => console.log(s);

'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。

String.prototype.search(regexp)

// 等同于regexp[Symbol.search](this)

class MySearch {

constructor(value) {

this.value = value;

}

[Symbol.search](string) {

return string.indexOf(this.value);

}}'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。

String.prototype.split(separator, limit)

// 等同于separator[Symbol.split](this, limit)

下面是一个例子。

class MySplitter {

constructor(value) {

this.value = value;

}

[Symbol.split](string) {

let index = string.indexOf(this.value);

if (index === -1) {

return string;

}

return [

string.substr(0, index),

string.substr(index + this.value.length)

];

}}

'foobar'.split(new MySplitter('foo'))

// ['', 'bar']'foobar'.split(new MySplitter('bar'))

// ['foo', '']'foobar'.split(new MySplitter('baz'))

// '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。

const myIterable = {};

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;};

[...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。

class Collection {

\*[Symbol.iterator]() {

let i = 0;

while(this[i] !== undefined) {

yield this[i];

++i;

}

}}

let myCollection = new Collection();

myCollection[0] = 1;

myCollection[1] = 2;

for(let value of myCollection) {

console.log(value);}

// 1// 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

let obj = {

[Symbol.toPrimitive](hint) {

switch (hint) {

case 'number':

return 123;

case 'string':

return 'str';

case 'default':

return 'default';

default:

throw new Error();

}

}};

2 \* obj // 2463 + obj // '3default'obj == 'default' // trueString(obj) // 'str'

### Symbol.toStringTag

对象的Symbol.toStringTag属性，指向一个方法。在该对象上面调用Object.prototype.toString方法时，如果这个属性存在，它的返回值会出现在toString方法返回的字符串之中，表示对象的类型。也就是说，这个属性可以用来定制[object Object]或[object Array]中object后面的那个字符串。

// 例一({[Symbol.toStringTag]: 'Foo'}.toString())

// "[object Foo]"

// 例二class Collection {

get [Symbol.toStringTag]() {

return 'xxx';

}}let x = new Collection();

Object.prototype.toString.call(x) // "[object xxx]"

ES6 新增内置对象的Symbol.toStringTag属性值如下。

* JSON[Symbol.toStringTag]：'JSON'
* Math[Symbol.toStringTag]：'Math'
* Module 对象M[Symbol.toStringTag]：'Module'
* ArrayBuffer.prototype[Symbol.toStringTag]：'ArrayBuffer'
* DataView.prototype[Symbol.toStringTag]：'DataView'
* Map.prototype[Symbol.toStringTag]：'Map'
* Promise.prototype[Symbol.toStringTag]：'Promise'
* Set.prototype[Symbol.toStringTag]：'Set'
* %TypedArray%.prototype[Symbol.toStringTag]：'Uint8Array'等
* WeakMap.prototype[Symbol.toStringTag]：'WeakMap'
* WeakSet.prototype[Symbol.toStringTag]：'WeakSet'
* %MapIteratorPrototype%[Symbol.toStringTag]：'Map Iterator'
* %SetIteratorPrototype%[Symbol.toStringTag]：'Set Iterator'
* %StringIteratorPrototype%[Symbol.toStringTag]：'String Iterator'
* Symbol.prototype[Symbol.toStringTag]：'Symbol'
* Generator.prototype[Symbol.toStringTag]：'Generator'
* GeneratorFunction.prototype[Symbol.toStringTag]：'GeneratorFunction'

### 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。

Array.prototype[Symbol.unscopables]

// {// copyWithin: true,// entries: true,// fill: true,// find: true,// findIndex: true,// includes: true,// keys: true// }

Object.keys(Array.prototype[Symbol.unscopables])

// ['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。

// 没有 unscopables 时class MyClass {

foo() { return 1; }}

var foo = function () { return 2; };

with (MyClass.prototype) {

foo(); // 1}

// 有 unscopables 时class MyClass {

foo() { return 1; }

get [Symbol.unscopables]() {

return { foo: true };

}}

var foo = function () { return 2; };

with (MyClass.prototype) {

foo(); // 2}

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。