

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.Symbol
- 11.Set 和 Map 数据结构
- 12.Proxy
- 13.Reflect
- 14.Promise 对象
- 15.Iterator 和 for...of 循环
- 16.Generator 函数的语法
- 17.Generator 函数的异步应用
- 18.async 函数
- 19.Class 的基本语法
- 20.Class 的继承
- 21.Decorator
- 22.Module 的语法
- 23.Module 的加载实现
- 24.编程风格
- 25.读懂规格
- 26.ArrayBuffer
- 27.参考链接

其他

- 源码
- 修订历史
- 反馈意见

Symbol

- 1.概述
- 2.作为属性名的 **Symbol**
- 3.实例：消除魔术字符串
- 4.属性名的遍历
- 5.**Symbol.for()**，**Symbol.keyFor()**
- 6.实例：模块的 **Singleton** 模式
- 7.内置的**Symbol**值

1. 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 // false

if (sym) {
  // ...
}

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

2. 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] // undefined
a['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。

```
log.levels = {
  DEBUG: Symbol('debug'),
  INFO: Symbol('info'),
  WARN: Symbol('warn')
};
log(log.levels.DEBUG, 'debug message');
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。

3. 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。

4. 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。

5. 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")
// true

Symbol("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。

6. 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.js
function 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。

```
const a = require('./mod.js');
global._foo = 123;
```

上面的代码，会使得别的脚本加载 `mod.js` 都失真。

为了防止这种情况出现，我们就可以使用`Symbol`。

```
// mod.js
const 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。

```
const a = require('./mod.js');  
global[Symbol.for('foo')] = 123;
```

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.js  
const 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，所以也不是完全可靠。

7. 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，判断左侧的运算符是否为 `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 // false  
2 instanceof Even // true  
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。

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

let 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，指向当前对象的构造函数。创造实例时，默认会调用这个方法，即使用这个属性返回的函数当作构造函数，来创造新的实例对象。

```
class MyArray extends Array {
  // 覆盖父类 Array 的构造函数
  static get [Symbol.species]() { return Array; }
}
```

上面代码中，子类 `MyArray` 继承了父类 `Array`。创建 `MyArray` 的实例对象时，本来会调用它自己的构造函数（本例中被省略了），但是由于定义了 `Symbol.species` 属性，所以会使用这个属性返回的的函数，创建 `MvArrav` 的实例。

这个例子也说明，定义 `Symbol.species` 属性要采用 `get` 读取器。默认的 `Symbol.species` 属性等同于下面的写法。

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

下面是一个例子。

```
class MyArray extends Array {  
  static get [Symbol.species]() { return Array; }  
}  
let a = new MyArray(1,2,3);  
let mapped = a.map(x => x * x);  
  
mapped instanceof MyArray // false  
mapped instanceof Array // true
```

上面代码中，由于构造函数被替换成了 `Array`。所以，`mapped` 对象不是 `MyArray` 的实例，而是 `Array` 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(regex)  
// 等同于  
regex[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(regex)
// 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 // 246
3 + obj // '3default'
obj == 'default' // true
String(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。

留言

