

# 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.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

## 其他

- 源码
- 修订历史
- 反馈意见

# 最新提案

- 1.do 表达式
- 2.throw 表达式
- 3.链判断运算符
- 4.直接输入 U+2028 和 U+2029
- 5.函数的部分执行
- 6.管道运算符
- 7.数值分隔符
- 8.BigInt 数据类型

本章介绍一些尚未进入标准、但很有希望的最新提案。

## 1. do 表达式

本质上，块级作用域是一个语句，将多个操作封装在一起，没有返回值。

```
{
  let t = f();
  t = t * t + 1;
}
```

上面代码中，块级作用域将两个语句封装在一起。但是，在块级作用域以外，没有办法得到 `t` 的值，因为块级作用域不返回值，除非 `t` 是全局变量。

现在有一个提案，使得块级作用域可以变为表达式，也就是说可以返回值，办法就是在块级作用域之前加上 `do`，使它变为 `do` 表达式，然后就会返回内部最后执行的表达式的值。

```
let x = do {
  let t = f();
  t * t + 1;
};
```

上面代码中，变量 `x` 会得到整个块级作用域的返回值（`t * t + 1`）。

`do` 表达式的逻辑非常简单：封装的是什么，就会返回什么。

```
// 等同于 <表达式>
do { <表达式>; }
```

```
// 等同于 <语句>
do { <语句> }
```

`do` 表达式的好处是可以封装多个语句，让程序更加模块化，就像乐高积木那样一块块拼装起来。

```
let x = do {
  if (foo()) { f() }
  else if (bar()) { g() }
  else { h() }
};
```

上面代码的本质，就是根据函数 `foo` 的执行结果，调用不同的函数，将返回结果赋给变量 `x`。使用 `do` 表达式，就将这个操作的意图表达得非常简洁清晰。而且，`do` 块级作用域提供了单独的作用域，内部操作可以与全局作用域隔绝。

值得一提的是，`do` 表达式在 `JSX` 语法中非常好用。

```
return (
  <nav>
    <Home />
    {
      do {
        if (loggedIn) {
          <LogoutButton />
        } else {
          <LoginButton />
        }
      }
    }
  </nav>
);
```

```
    }  
  }  
  </nav>  
)
```

上面代码中，如果不用 `do` 表达式，就只能用三元判断运算符（`?:`）。那样的话，一旦判断逻辑复杂，代码就会变得很不易读。

---

## 2. throw 表达式

JavaScript 语法规规定 `throw` 是一个命令，用来抛出错误，不能用于表达式之中。

```
// 报错  
console.log(throw new Error());
```

上面代码中，`console.log` 的参数必须是一个表达式，如果是一个 `throw` 语句就会报错。

现在有一个提案，允许 `throw` 用于表达式。

```
// 参数的默认值  
function save(filename = throw new TypeError("Argument required")) {  
  }  
  
// 箭头函数的返回值  
lint(ast, {  
  with: () => throw new Error("avoid using 'with' statements.")  
});  
  
// 条件表达式  
function getEncoder(encoding) {  
  const encoder = encoding === "utf8" ?  
    new UTF8Encoder() :  
    encoding === "utf16le" ?  
      new UTF16Encoder(false) :  
      encoding === "utf16be" ?  
        new UTF16Encoder(true) :  
        throw new Error("Unsupported encoding");  
}  
  
// 逻辑表达式  
class Product {  
  get id() {  
    return this._id;  
  }  
  set id(value) {  
    this._id = value || throw new Error("Invalid value");  
  }  
}
```

上面代码中，`throw` 都出现在表达式里面。

语法上，`throw` 表达式里面的 `throw` 不再是一个命令，而是一个运算符。为了避免与 `throw` 命令混淆，规定 `throw` 出现在行首，一律解释为 `throw` 语句，而不是 `throw` 表达式。

---

## 3. 链判断运算符

编程实务中，如果读取对象内部的某个属性，往往需要判断一下该对象是否存在。比如，要读取 `message.body.user.firstName`，安全的写法是写成下面这样。

```
const firstName = (message
  && message.body
  && message.body.user
  && message.body.user.firstName) || 'default';
```

这样的层层判断非常麻烦，因此现在有一个提案，引入了“链判断运算符”（optional chaining operator）`?.`，简化上面的写法。

```
const firstName = message?.body?.user?.firstName || 'default';
```

上面代码有三个 `?.` 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`。

链判断运算符有三种用法。

- `obj?.prop` // 读取对象属性
- `obj?.[expr]` // 同上
- `func?.(...args)` // 函数或对象方法的调用

下面是判断函数是否存在的例子。

```
iterator.return?.()
```

上面代码中，`iterator.return` 如果有定义，就会调用该方法，否则直接返回 `undefined`。

下面是更多的例子。

```
a?.b
// 等同于
a == null ? undefined : a.b

a?.[x]
// 等同于
a == null ? undefined : a[x]

a?.b()
// 等同于
a == null ? undefined : a.b()

a?.()
// 等同于
a == null ? undefined : a()
```

使用这个运算符，有几个注意点。

#### （1）短路机制

```
a?.[++x]
// 等同于
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。

#### （2）delete 运算符

```
delete a?.b
// 等同于
a == null ? undefined : delete a.b
```

上面代码中，如果 `a` 是 `undefined` 或 `null`，会直接返回 `undefined`，而不会进行 `delete` 运算。

### （3）报错场合

以下写法是禁止，会报错。

```
// 构造函数判断
new a?.()

// 运算符右侧是模板字符串
a?.`{b}`

// 链判断运算符前后有构造函数或模板字符串
new a?.b()
a?.b`{c}`

// 链运算符用于赋值运算符左侧
a?.b = c
```

### （4）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。

---

## 4. 直接输入 U+2028 和 U+2029

JavaScript 字符串允许直接输入字符，以及输入字符的转义形式。举例来说，“中”的 Unicode 码点是 U+4e2d，你可以直接在字符串里面输入这个汉字，也可以输入它的转义形式 `\u4e2d`，两者是等价的。

```
'中' === '\u4e2d' // true
```

但是，JavaScript 规定有5个字符，不能在字符串里面直接使用，只能使用转义形式。

- U+005C：反斜杠（reverse solidus）
- U+000D：回车（carriage return）
- U+2028：行分隔符（line separator）
- U+2029：段分隔符（paragraph separator）
- U+000A：换行符（line feed）

举例来说，字符串里面不能直接包含反斜杠，一定要转义写成 `\\` 或者 `\u005c`。

这个规定本身没有问题，麻烦在于 JSON 格式允许字符串里面直接使用 U+2028（行分隔符）和 U+2029（段分隔符）。这样一来，服务器输出的 JSON 被 `JSON.parse` 解析，就有可能直接报错。

JSON 格式已经冻结（RFC 7159），没法修改了。为了消除这个报错，现在有一个提案，允许 JavaScript 字符串直接输入 U+2028（行分隔符）和 U+2029（段分隔符）。

```
const PS = eval('"\"u2029"');
```

根据这个提案，上面的代码不会报错。

[上一章](#)

[下一章](#)

注意，模板字符串现在就允许直接输入这两个字符。另外，正则表达式依然不允许直接输入这两个字符，这是没有问题的，因为 JSON 本来就不允许直接包含正则表达式。

## 5. 函数的部分执行

### 语法

多参数的函数有时需要绑定其中的一个或多个参数，然后返回一个新函数。

```
function add(x, y) { return x + y; }
function add7(x) { return x + 7; }
```

上面代码中，`add7` 函数其实是 `add` 函数的一个特殊版本，通过将一个参数绑定为 `7`，就可以从 `add` 得到 `add7`。

```
// bind 方法
const add7 = add.bind(null, 7);

// 箭头函数
const add7 = x => add(x, 7);
```

上面两种写法都有些冗余。其中，`bind` 方法的局限更加明显，它必须提供 `this`，并且只能从前到后一个个绑定参数，无法只绑定非头部的参数。

现在有一个提案，使得绑定参数并返回一个新函数更加容易。这叫做函数的部分执行（`partial application`）。

```
const add = (x, y) => x + y;
const addOne = add(1, ?);

const maxGreaterThanZero = Math.max(0, ...);
```

根据新提案，`?` 是单个参数的占位符，`...` 是多个参数的占位符。以下的形式都属于函数的部分执行。

```
f(x, ?)
f(x, ...)
f(?, x)
f(..., x)
f(?, x, ?)
f(..., x, ...)
```

`?` 和 `...` 只能出现在函数的调用之中，并且会返回一个新函数。

```
const g = f(?, 1, ...);
// 等同于
const g = (x, ...y) => f(x, 1, ...y);
```

函数的部分执行，也可以用于对象的方法。

```
let obj = {
  f(x, y) { return x + y; },
};
```

```
const g = obj.f(?, 3);  
g(1) // 4
```

---

## 注意点

函数的部分执行有一些特别注意的地方。

(1) 函数的部分执行是基于原函数的。如果原函数发生变化，部分执行生成的新函数也会立即反映这种变化。

```
let f = (x, y) => x + y;  
  
const g = f(?, 3);  
g(1); // 4  
  
// 替换函数 f  
f = (x, y) => x * y;  
  
g(1); // 3
```

上面代码中，定义了函数的部分执行以后，更换原函数会立即影响到新函数。

(2) 如果预先提供的那个值是一个表达式，那么这个表达式并不会在定义时求值，而是在每次调用时求值。

```
let a = 3;  
const f = (x, y) => x + y;  
  
const g = f(?, a);  
g(1); // 4  
  
// 改变 a 的值  
a = 10;  
g(1); // 11
```

上面代码中，预先提供的参数是变量 **a**，那么每次调用函数 **g** 的时候，才会对 **a** 进行求值。

(3) 如果新函数的参数多于占位符的数量，那么多余的参数将被忽略。

```
const f = (x, ...y) => [x, ...y];  
const g = f(?, 1);  
g(2, 3, 4); // [2, 1]
```

上面代码中，函数 **g** 只有一个占位符，也就意味着它只能接受一个参数，多余的参数都会被忽略。

写成下面这样，多余的参数就没有问题。

```
const f = (x, ...y) => [x, ...y];  
const g = f(?, 1, ...);  
g(2, 3, 4); // [2, 1, 3, 4];
```

(4) **...** 只会被采集一次，如果函数的部分执行使用了多个 **...**，那么每个 **...** 的值都将相同。

```
const f = (...x) => x;  
const g = f(..., 9, ...);  
g(1, 2, 3); // [1, 2, 3, 9, 1, 2, 3]
```

上面代码中，**g** 定义了两个 **...** 占位符，真正执行的时候，它们的值是一样的

[上一章](#)

[下一章](#)

## 6. 管道运算符

Unix 操作系统有一个管道机制（**pipeline**），可以把前一个操作的值传给后一个操作。这个机制非常有用，使得简单的操作可以组合成为复杂的操作。许多语言都有管道的实现，现在有一个提案，让 **JavaScript** 也拥有管道机制。

**JavaScript** 的管道是一个运算符，写作 `|>`。它的左边是一个表达式，右边是一个函数。管道运算符把左边表达式的值，传入右边的函数进行求值。

```
x |> f
// 等同于
f(x)
```

管道运算符最大的好处，就是可以把嵌套的函数，写成从左到右的链式表达式。

```
function doubleSay (str) {
  return str + ", " + str;
}

function capitalize (str) {
  return str[0].toUpperCase() + str.substring(1);
}

function exclaim (str) {
  return str + '!';
}
```

上面是三个简单的函数。如果要嵌套执行，传统的写法和管道的写法分别如下。

```
// 传统的写法
exclaim(capitalize(doubleSay('hello')))
// "Hello, hello!"

// 管道的写法
'hello'
  |> doubleSay
  |> capitalize
  |> exclaim
// "Hello, hello!"
```

管道运算符只能传递一个值，这意味着它右边的函数必须是一个单参数函数。如果是多参数函数，就必须进行柯里化，改成单参数的版本。

```
function double (x) { return x + x; }
function add (x, y) { return x + y; }

let person = { score: 25 };
person.score
  |> double
  |> (_ => add(7, _))
// 57
```

上面代码中，`add` 函数需要两个参数。但是，管道运算符只能传入一个值，因此需要事先提供另一个参数，并将其改成单参数的箭头函数 `_ => add(7, _)`。这个函数里面的下划线并没有特别的含义，可以用其他符号代替，使用下划线只是因为，它能够形象地表示这里是占位符。

管道运算符对于 `await` 函数也适用。

```
x |> await f
// 等同于
```



```
await f(x)

const userAge = userId |> await fetchUserById |> getAgeFromUser;
// 等同于
const userAge = getAgeFromUser(await fetchUserById(userId));
```

---

## 7. 数值分隔符

欧美语言中，较长的数值允许每三位添加一个分隔符（通常是一个逗号），增加数值的可读性。比如，`1000` 可以写作 `1,000`。

现在有一个提案，允许 JavaScript 的数值使用下划线（`_`）作为分隔符。

```
let budget = 1_000_000_000_000;
budget === 10 ** 12 // true
```

JavaScript 的数值分隔符没有指定间隔的位数，也就是说，可以每三位添加一个分隔符，也可以每一位、每两位、每四位添加一个。

```
123_00 === 12_300 // true

12345_00 === 123_4500 // true
12345_00 === 1_234_500 // true
```

小数和科学计数法也可以使用数值分隔符。

```
// 小数
0.000_001
// 科学计数法
1e10_000
```

数值分隔符有几个使用注意点。

- 不能在数值的最前面（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。

下面的写法都会报错。

```
// 全部报错
3_.141
3._141
1_e12
1e_12
123__456
_1464301
1464301_
```

除了十进制，其他进制的数值也可以使用分隔符。

```
// 二进制
0b1010_0001_1000_0101
// 十六进制
0xA0_B0_C0
```

```
// 报错
0_b111111000
0b_111111000
```

下面三个将字符串转成数值的函数，不支持数值分隔符。主要原因是提案的设计者认为，数值分隔符主要是为了编码时书写数值的方便，而不是为了处理外部输入的数据。

- Number()
- parseInt()
- parseFloat()

```
Number('123_456') // NaN
parseInt('123_456') // 123
```

---

## 8. BigInt 数据类型

---

### 简介

JavaScript 所有数字都保存成 64 位浮点数，这给数值的表示带来了两大限制。一是数值的精度只能到 53 个二进制位（相当于 16 个十进制位），大于这个范围的整数，JavaScript 是无法精确表示的，这使得 JavaScript 不适合进行科学和金融方面的精确计算。二是大于或等于 2 的 1024 次方的数值，JavaScript 无法表示，会返回 **Infinity**。

```
// 超过 53 个二进制位的数值，无法保持精度
Math.pow(2, 53) === Math.pow(2, 53) + 1 // true

// 超过 2 的 1024 次方的数值，无法表示
Math.pow(2, 1024) // Infinity
```

现在有一个提案，引入了一种新的数据类型 **BigInt**（大整数），来解决这个问题。**BigInt** 只用来表示整数，没有位数的限制，任何位数的整数都可以精确表示。

```
const a = 2172141653n;
const b = 15346349309n;

// BigInt 可以保持精度
a * b // 33334444555566667777n

// 普通整数无法保持精度
Number(a) * Number(b) // 33334444555566670000
```

为了与 **Number** 类型区别，**BigInt** 类型的数据必须添加后缀 **n**。

```
1234 // 普通整数
1234n // BigInt

// BigInt 的运算
1n + 2n // 3n
```

**BigInt** 同样可以使用各种进制表示，都要加上后缀 **n**。

```
0b1101n // 二进制
0o777n // 八进制
0xFFn // 十六进制
```

BigInt 与普通整数是两种值，它们之间并不相等。

```
42n === 42 // false
```

typeof 运算符对于 BigInt 类型的数据返回 bigint 。

```
typeof 123n // 'bigint'
```

BigInt 可以使用负号（-），但是不能使用正号（+），因为会与 asm.js 冲突。

```
-42n // 正确
+42n // 报错
```

---

## BigInt 对象

JavaScript 原生提供 BigInt 对象，可以用作构造函数生成 BigInt 类型的数值。转换规则基本与 Number() 一致，将其他类型的值转为 BigInt。

```
BigInt(123) // 123n
BigInt('123') // 123n
BigInt(false) // 0n
BigInt(true) // 1n
```

BigInt() 构造函数必须有参数，而且参数必须可以正常转为数值，下面的用法都会报错。

```
new BigInt() // TypeError
BigInt(undefined) //TypeError
BigInt(null) // TypeError
BigInt('123n') // SyntaxError
BigInt('abc') // SyntaxError
```

上面代码中，尤其值得注意字符串 123n 无法解析成 Number 类型，所以会报错。

参数如果是小数，也会报错。

```
BigInt(1.5) // RangeError
BigInt('1.5') // SyntaxError
```

BigInt 对象继承了 Object 提供的实例方法。

- BigInt.prototype.toLocaleString()
- BigInt.prototype.toString()
- BigInt.prototype.valueOf()

此外，还提供了三个静态方法。

- BigInt.asUintN(width, BigInt)：给定的 BigInt 转换成 2<sup>width</sup> 之间对应的值。

- `BigInt.asIntN(width, BigInt)`：给定的 `BigInt` 转为  $-2^{\text{width}-1}$  到  $2^{\text{width}-1}-1$  之间对应的值。
- `BigInt.parseInt(string[, radix])`：近似于 `Number.parseInt()`，将一个字符串转换成指定进制的 `BigInt`。

```
const max = 2n ** (64n - 1n) - 1n;

BigInt.asIntN(64, max)
// 9223372036854775807n
BigInt.asIntN(64, max + 1n)
// -9223372036854775808n
BigInt.asUintN(64, max + 1n)
// 9223372036854775808n
```

上面代码中，`max` 是64位带符号的 `BigInt` 所能表示的最大值。如果对这个值加 `1n`，`BigInt.asIntN()` 将会返回一个负值，因为这时新增的一位将被解释为符号位。而 `BigInt.asUintN()` 方法由于不存在符号位，所以可以正确返回结果。

如果 `BigInt.asIntN()` 和 `BigInt.asUintN()` 指定的位数，小于数值本身的位数，那么头部的位将被舍弃。

```
const max = 2n ** (64n - 1n) - 1n;

BigInt.asIntN(32, max) // -1n
BigInt.asUintN(32, max) // 4294967295n
```

上面代码中，`max` 是一个64位的 `BigInt`，如果转为32位，前面的32位都会被舍弃。

下面是 `BigInt.parseInt()` 的例子。

```
// Number.parseInt() 与 BigInt.parseInt() 的对比
Number.parseInt('9007199254740993', 10)
// 9007199254740992
BigInt.parseInt('9007199254740993', 10)
// 9007199254740993n
```

上面代码中，由于有效数字超出了最大限度，`Number.parseInt` 方法返回的结果是不精确的，而 `BigInt.parseInt` 方法正确返回了对应的 `BigInt`。

对于二进制数组，`BigInt` 新增了两个类型 `BigUint64Array` 和 `BigInt64Array`，这两种数据类型返回的都是64位 `BigInt`。`DataView` 对象的实例方法 `DataView.prototype.getBigInt64()` 和 `DataView.prototype.getBigUint64()`，返回的也是 `BigInt`。

---

## 转换规则

可以使用 `Boolean()`、`Number()` 和 `String()` 这三个方法，将 `BigInt` 可以转为布尔值、数值和字符串类型。

```
Boolean(0n) // false
Boolean(1n) // true
Number(1n)  // 1
String(1n)  // "1"
```

上面代码中，注意最后一个例子，转为字符串时后缀 `n` 会消失。

另外，取反运算符（`!`）也可以将 `BigInt` 转为布尔值。

```
!0n // true
!1n // false
```

## 数学运算

数学运算方面，`BigInt` 类型的 `+`、`-`、`*` 和 `**` 这四个二元运算符，与 `Number` 类型的行为一致。除法运算 `/` 会舍去小数部分，返回一个整数。

```
9n / 5n
// 1n
```

几乎所有的数值运算符都可以用在 `BigInt`，但是有两个例外。

- 不带符号的右移位运算符 `>>>`
- 一元的求正运算符 `+`

上面两个运算符用在 `BigInt` 会报错。前者是因为 `>>>` 运算符是不带符号的，但是 `BigInt` 总是带有符号的，导致该运算无意义，完全等同于右移运算符 `>>`。后者是因为一元运算符 `+` 在 `asm.js` 里面总是返回 `Number` 类型，为了不破坏 `asm.js` 就规定 `+1n` 会报错。

`BigInt` 不能与普通数值进行混合运算。

```
1n + 1.3 // 报错
```

上面代码报错是因为无论返回的是 `BigInt` 或 `Number`，都会导致丢失精度信息。比如 `(2n**53n + 1n) + 0.5` 这个表达式，如果返回 `BigInt` 类型，`0.5` 这个小数部分会丢失；如果返回 `Number` 类型，有效精度只能保持 53 位，导致精度下降。

同样的原因，如果一个标准库函数的参数预期是 `Number` 类型，但是得到的是一个 `BigInt`，就会报错。

```
// 错误的写法
Math.sqrt(4n) // 报错

// 正确的写法
Math.sqrt(Number(4n)) // 2
```

上面代码中，`Math.sqrt` 的参数预期是 `Number` 类型，如果是 `BigInt` 就会报错，必须先用 `Number` 方法转一下类型，才能进行计算。

`asm.js` 里面，`|0` 跟在一个数值的后面会返回一个32位整数。根据不能与 `Number` 类型混合运算的规则，`BigInt` 如果与 `|0` 进行运算会报错。

```
1n | 0 // 报错
```

---

## 其他运算

`BigInt` 对应的布尔值，与 `Number` 类型一致，即 `0n` 会转为 `false`，其他值转为 `true`。

```
if (0n) {
  console.log('if');
} else {
  console.log('else');
}
// else
```

上面代码中，`0n` 对应 `false`，所以会进入 `else` 子句。

比较运算符（比如 `>`）和相等运算符（`==`）允许 `BigInt` [上一章](#) [下一章](#) 与 `Number` 类型的数值混合运算，因为这样做不会损失精度。

```
0n < 1 // true
0n < true // true
0n == 0 // true
0n == false // true
0n === 0 // false
```

BigInt 与字符串混合运算时，会先转为字符串，再进行运算。

```
' ' + 123n // "123"
```

---

## 9. Math.signbit()

`Math.sign()` 用来判断一个值的正负，但是如果参数是 `-0`，它会返回 `-0`。

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

这导致对于判断符号位的正负，`Math.sign()` 不是很有用。JavaScript 内部使用 64 位浮点数（国际标准 IEEE 754）表示数值，IEEE 754 规定第一位是符号位，`0` 表示正数，`1` 表示负数。所以会有两种零，`+0` 是符号位为 `0` 时的零值，`-0` 是符号位为 `1` 时的零值。实际编程中，判断一个值是 `+0` 还是 `-0` 非常麻烦，因为它们是相等的。

```
+0 === -0 // true
```

目前，有一个提案，引入了 `Math.signbit()` 方法判断一个数的符号位是否设置了。

```
Math.signbit(2) //false
Math.signbit(-2) //true
Math.signbit(0) //false
Math.signbit(-0) //true
```

可以看到，该方法正确返回了 `-0` 的符号位是设置了的。

该方法的算法如下。

- 如果参数是 `NaN`，返回 `false`
- 如果参数是 `-0`，返回 `true`
- 如果参数是负值，返回 `true`
- 其他情况返回 `false`

---

留言

6 Comments

ECMAScript 6 入门

 Login ▾

 Recommend 1

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH

上一章

下一章

**hao wangg** • 4 months ago

觉得链判断运算符很好用

^ | v • Reply • Share ›

**image72** • a year ago

建议增加这个

proposal-pipeline-operator

<https://github.com/tc39/pro...> pipe

^ | v • Reply • Share ›

**ruanyf** Mod → **image72** • a year ago

你刷新一下页面，已经加上去了。

^ | v • Reply • Share ›

**image72** → **ruanyf** • a year ago

😊 更新好快

^ | v • Reply • Share ›

**chase butterfly** • a year ago

有没有babel来支持最新的提案呢？

^ | v • Reply • Share ›

**faichou** • a year ago

Null 传导运算符 很方便, 支持. 特别像swift

^ | v • Reply • Share ›

ALSO ON ECMAScript 6 入门

## Module 的加载实现

18 comments • 2 years ago

**ruanyf** — 你说的对，这里是有错，我已经把缓存这部分删掉了。import() 可以用 Babel 执行。```javascript//

...

## Generator 函数：异步操作

30 comments • 2 years ago

**ruanyf** — 理解正确。Promise 和 Thunk 本质都是对回调函数的包装，只是 API 更友好了。2018-05-19 0:31

...

## 对象和函数的扩展

90 comments • 5 years ago

**yunnysunny** — 可惜Null传导运算符现在还不支持

## SIMD 的用法

3 comments • 3 years ago

**dou4cc** — Firefox Nightly 支持！

