ECMAScript 6 入门

作者：阮一峰

目录

其他

最新提案

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>
)
```

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';
```

或者使用三元运算符 `?:`，判断一个对象是否存在。

```
const fooInput = myForm.querySelector('input[name=foo]')
const fooValue = fooInput ? fooInput.value : undefined
```

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';
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`。

链判断运算符有三种用法。

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

下面是判断对象方法是否存在，如果存在就立即执行的例子。

```
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`。

对于那些可能没有实现的方法，这个运算符尤其有用。

```
if (myForm.checkValidity?.() === false) {
  // 表单校验失败
  return;
}
```

上面代码中，老式浏览器的表单可能没有 `checkValidity` 这个方法，这时 `?.` 运算符就会返回 `undefined`，判断语句就变成了 `undefined === false`，所以就会跳过下面的代码。

下面是这个运算符常见的使用形式，以及不使用该运算符时的等价形式。

```
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()
```

上面代码中，特别注意后两种形式，如果 `a?.b()` 里面的 `a.b` 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，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）括号不改变运算顺序

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

上面代码中，`?.` 对圆括号没有影响，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。

（4）报错场合

以下写法是禁止的，会报错。

```
// 构造函数
new a?.()
new a?.b()
```

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

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

// 链运算符用于赋值运算符左侧
a?.b = c
```

（5）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. 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。

```
const headerText = response.settings.headerText || 'Hello, world!';
const animationDuration = response.settings.animationDuration || 300;
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，现在有一个提案，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。

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

上面代码中，默认值只有在属性值为 `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。

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

上面代码中，`response.settings` 如果是 `null` 或 `undefined`，就会返回默认值300。

这个运算符很适合判断函数参数是否赋值。

```
function Component(props) {
  const enable = props.enabled ?? true;
  // …
}
```

上面代码判断 `props` 参数的 `enabled` 属性是否赋值，等同于下面的写法。

```
function Component(props) {
  const {
```

```
    enabled: enable = true,
  } = props;
  // …
}
```

`??` 有一个运算优先级问题，它与 `&&` 和 `||` 的优先级孰高孰低。现在的规则是，如果多个逻辑运算符一起使用，必须用括号表明优先级，否则会报错。

```
// 报错
lhs && middle ?? rhs
lhs ?? middle && rhs
lhs || middle ?? rhs
lhs ?? middle || rhs
```

上面四个表达式都会报错，必须加入表明优先级的括号。

```
(lhs && middle) ?? rhs;
lhs && (middle ?? rhs);

(lhs ?? middle) && rhs;
lhs ?? (middle && rhs);

(lhs || middle) ?? rhs;
lhs || (middle ?? rhs);

(lhs ?? middle) || rhs;
lhs ?? (middle || rhs);
```

---

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。

---

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
```

注意，分隔符不能紧跟着进制的前缀 `0b`、`0B`、`0o`、`0O`、`0x`、`0X`。

```
// 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

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 转为 0 到 $2^{width} - 1$ 之间对应的值。
- `BigInt.asIntN(width, BigInt)`：给定的 BigInt 转为 $-2^{width - 1}$ 到 $2^{width - 1} - 1$ 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 与其他类型的值混合计算，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`

---

10. 双冒号运算符

箭头函数可以绑定 `this` 对象，大大减少了显式绑定 `this` 对象的写法（`call`、`apply`、`bind`）。但是，箭头函数并不适用于所有场合，所以现在有一个提案，提出了“函数绑定”（function bind）运算符，用来取代 `call`、`apply`、`bind` 调用。

函数绑定运算符是并排的两个冒号（`::`），双冒号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即 `this` 对象），绑定到右边的函数上面。

```
foo::bar;
// 等同于
bar.bind(foo);

foo::bar(...arguments);
// 等同于
bar.apply(foo, arguments);

const hasOwnProperty = Object.prototype.hasOwnProperty;
function hasOwn(obj, key) {
  return obj::hasOwnProperty(key);
}
```

如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上面。

```
var method = obj::obj.foo;
// 等同于
var method = ::obj.foo;

let log = ::console.log;
```

```
// 等同于
var log = console.log.bind(console);
```

如果双冒号运算符的运算结果，还是一个对象，就可以采用链式写法。

```
import { map, takeWhile, forEach } from "iterlib";

getPlayers()
::map(x => x.character())
::takeWhile(x => x.strength > 100)
::forEach(x => console.log(x));
```

---

11. Realm API

Realm API 提供沙箱功能（sandbox），允许隔离代码，防止那些被隔离的代码拿到全局对象。

以前，经常使用 `<iframe>` 作为沙箱。

```
const globalOne = window;
let iframe = document.createElement('iframe');
document.body.appendChild(iframe);
const globalTwo = iframe.contentWindow;
```

上面代码中， `<iframe>` 的全局对象是独立的（ `iframe.contentWindow` ）。Realm API 可以取代这个功能。

```
const globalOne = window;
const globalTwo = new Realm().global;
```

上面代码中， `Realm API` 单独提供了一个全局对象 `new Realm().global` 。

Realm API 提供一个 `Realm()` 构造函数，用来生成一个 Realm 对象。该对象的 `global` 属性指向一个新的顶层对象，这个顶层对象跟原始的顶层对象类似。

```
const globalOne = window;
const globalTwo = new Realm().global;

globalOne.evaluate('1 + 2') // 3
globalTwo.evaluate('1 + 2') // 3
```

上面代码中，Realm 生成的顶层对象的 `evaluate()` 方法，可以运行代码。

下面的代码可以证明，Realm 顶层对象与原始顶层对象是两个对象。

```
let a1 = globalOne.evaluate('[1,2,3]');
let a2 = globalTwo.evaluate('[1,2,3]');
a1.prototype === a2.prototype; // false
a1 instanceof globalTwo.Array; // false
a2 instanceof globalOne.Array; // false
```

上面代码中，Realm 沙箱里面的数组的原型对象，跟原始环境里面的数组是不一样的。

Realm 沙箱里面只能运行 ECMAScript 语法提供的 API，不能运行宿主环境提供的 API。

```
globalTwo.evaluate('console.log(1)')
// throw an error: console is undefined
```

上面代码中，Realm 沙箱里面没有 `console` 对象，导致报错。因为 `console` 不是语法标准，是宿主环境提供的。

如果要解决这个问题，可以使用下面的代码。

```
globalTwo.console = globalOne.console;
```

`Realm()` 构造函数可以接受一个参数对象，该参数对象的 `intrinsics` 属性可以指定 Realm 沙箱继承原始顶层对象的方法。

```
const r1 = new Realm();
r1.global === this;
r1.global.JSON === JSON; // false

const r2 = new Realm({ intrinsics: 'inherit' });
r2.global === this; // false
r2.global.JSON === JSON; // true
```

上面代码中，正常情况下，沙箱的 `JSON` 方法不同于原始的 `JSON` 对象。但是， `Realm()` 构造函数接受 `{ intrinsics: 'inherit' }` 作为参数以后，就会继承原始顶层对象的方法。

用户可以自己定义 `Realm` 的子类，用来定制自己的沙箱。

```
class FakeWindow extends Realm {
  init() {
    super.init();
    let global = this.global;

    global.document = new FakeDocument(...);
    global.alert = new Proxy(fakeAlert, { ... });
    // ...
  }
}
```

上面代码中， `FakeWindow` 模拟了一个假的顶层对象 `window` 。

---

12. `#!` 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。

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

Python 脚本的第一行。

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

现在有一个提案，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。

```
// 写在脚本文件第一行
#!/usr/bin/env node
```

```
'use strict';
console.log(1);

// 写在模块文件第一行
#!/usr/bin/env node
export {};
console.log(1);
```

有了这一行以后，Unix 命令行就可以直接执行脚本。

```
# 以前执行脚本的方式
$ node hello.js

# hashbang 的方式
$ hello.js
```

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

---

13. import.meta

加载 JavaScript 脚本的时候，有时候需要知道脚本的元信息。Node.js 提供了两个特殊变量 `__filename` 和 `__dirname`，用来获取脚本的文件名和所在路径。

```
const fs = require('fs');
const path = require('path');
const bytes = fs.readFileSync(path.resolve(__dirname, 'data.bin'));
```

上面代码中，`__dirname` 用于加载与脚本同一个目录的数据文件 `data.bin`。

但是，浏览器没有这两个特殊变量。如果需要知道脚本的元信息，就只有手动提供。

```
<script data-option="value" src="library.js"></script>
```

上面这一行 HTML 代码加载 JavaScript 脚本，使用 `data-` 属性放入元信息。如果脚本内部要获知元信息，可以像下面这样写。

```
const theOption = document.currentScript.dataset.option;
```

上面代码中，`document.currentScript` 属性可以拿到当前脚本的 DOM 节点。

由于 Node.js 和浏览器做法的不统一，现在有一个提案，提出统一使用 `import.meta` 属性在脚本内部获取元信息。这个属性返回一个对象，该对象的各种属性就是当前运行的脚本的元信息。具体包含哪些属性，标准没有规定，由各个运行环境自行决定。

一般来说，浏览器的 `import.meta` 至少会有两个属性。

- `import.meta.url`：脚本的 URL。
- `import.meta.scriptElement`：加载脚本的那个 `<script>` 的 DOM 节点，用来替代 `document.currentScript`。

```
<script type="module" src="path/to/hamster-displayer.js" data-size="500"></script>
```

上面这行代码加载的脚本内部，就可以使用 `import.meta` 获取元信息。

```
(async () => {
  const response = await fetch(new URL("../hamsters.jpg", import.meta.url));
  const blob = await response.blob();

  const size = import.meta.scriptElement.dataset.size || 300;

  const image = new Image();
  image.src = URL.createObjectURL(blob);
  image.width = image.height = size;

  document.body.appendChild(image);
})();
```

上面代码中， `import.meta` 用来获取所加载的图片的尺寸。

留言