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 上一章 作 下一章 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. 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。

---

4. 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));
```

---

5. 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
```

---

6. 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`

---

7. 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));
```

---

8. 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` 。

---

9. `#!` 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。

---

10. import.meta

开发者使用一个模块时，有时需要知道模板本身的一些信息（比如模块的路径）。现在有一个提案，为 import 命令添加了一个元属性 `import.beta`，返回当前模块的元信息。

`import.meta` 只能在模块内部使用，如果在模块外部使用会报错。

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，`import.meta` 至少会有下面两个属性。

（1）import.meta.url

`import.meta.url` 返回当前模块的 URL 路径。举例来说，当前模块主文件的路径是 `https://foo.com/main.js`，`import.meta.url` 就返回这个路径。如果模块里面还有一个数据文件 `data.txt`，那么就可以用下面的代码，获取这个数据文件的路径。

```
new URL('data.txt', import.meta.url)
```

注意，Node.js 环境中，`import.meta.url` 返回的总是本地路径，即是 `file:URL` 协议的字符串，比如 `file:///home/user/foo.js`。

（2）import.meta.scriptElement

`import.meta.scriptElement` 是浏览器特有的元属性，返回加载模块的那个 `<script>` 元素，相当于 `document.currentScript` 属性。

```
// HTML 代码为
// <script type="module" src="my-module.js" data-foo="abc"></script>

// my-module.js 内部执行下面的代码
import.meta.scriptElement.dataset.foo
// "abc"
```

留言