ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

变量的解构赋值

---

1. 数组的解构赋值

基本用法

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

```
let a = 1;
let b = 2;
let c = 3;
```

ES6 允许写成下面这样。

```
let [a, b, c] = [1, 2, 3];
```

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [ , , third] = ["foo", "bar", "baz"];
third // "baz"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

let [head, ...tail] = [1, 2, 3, 4];
head // 1
tail // [2, 3, 4]

let [x, y, ...z] = ['a'];
x // "a"
y // undefined
z // []
```

如果解构不成功，变量的值就等于 `undefined` 。

```
let [foo] = [];
let [bar, foo] = [1];
```

以上两种情况都属于解构不成功， `foo` 的值都会等于 `undefined` 。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

```
let [x, y] = [1, 2, 3];
x // 1
y // 2

let [a, [b], d] = [1, [2, 3], 4];
a // 1
b // 2
d // 4
```

上面两个例子，都属于不完全解构，但是可以成功。

如果等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错。

```
// 报错
let [foo] = 1;
let [foo] = false;
let [foo] = NaN;
let [foo] = undefined;
let [foo] = null;
let [foo] = {};
```

上面的语句都会报错，因为等号右边的值，要么转为对象以后不具备 Iterator 接口（前五个表达式），要么本身就不具备 Iterator 接口（最后一个表达式）。

对于 Set 结构，也可以使用数组的解构赋值。

```
let [x, y, z] = new Set(['a', 'b', 'c']);
x // "a"
```

事实上，只要某种数据结构具有 Iterator 接口，都可以采用数组形式的解构赋值。

```
function* fibs() {
  let a = 0;
  let b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

let [first, second, third, fourth, fifth, sixth] = fibs();
sixth // 5
```

上面代码中， `fibs` 是一个 Generator 函数（参见《Generator 函数》一章），原生具有 Iterator 接口。解构赋值会依次从这个接口获取值。

默认值

解构赋值允许指定默认值。

```
let [foo = true] = [];
foo // true

let [x, y = 'b'] = ['a']; // x='a', y='b'
let [x, y = 'b'] = ['a', undefined]; // x='a', y='b'
```

注意，ES6 内部使用严格相等运算符（ `===` ），判断一个位置是否有值。所以，如果一个数组成员不严格等于 `undefined` ，默认值是不会生效的。

```
let [x = 1] = [undefined];
x // 1

let [x = 1] = [null];
x // null
```

上面代码中，如果一个数组成员是 `null` ，默认值就不会生效，因为 `null` 不严格等于 `undefined` 。

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

```
function f() {
  console.log('aaa');
}

let [x = f()] = [1];
```

上面代码中，因为 `x` 能取到值，所以函数 `f` 根本不会执行。上面的代码其实等价于下面的代码。

```
let x;
if ([1][0] === undefined) {
  x = f();
} else {
  x = [1][0];
}
```

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

```
let [x = 1, y = x] = [];     // x=1; y=1
let [x = 1, y = x] = [2];    // x=2; y=2
let [x = 1, y = x] = [1, 2]; // x=1; y=2
let [x = y, y = 1] = [];     // ReferenceError
```

上面最后一个表达式之所以会报错，是因为 x 用到默认值 y 时， y 还没有声明。

---

2. 对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

```
let { foo, bar } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
let { bar, foo } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"

let { baz } = { foo: "aaa", bar: "bbb" };
baz // undefined
```

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于 undefined 。

如果变量名与属性名不一致，必须写成下面这样。

```
let { foo: baz } = { foo: 'aaa', bar: 'bbb' };
baz // "aaa"

let obj = { first: 'hello', last: 'world' };
let { first: f, last: l } = obj;
f // 'hello'
l // 'world'
```

这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。

```
let { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };
```

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

```
let { foo: baz } = { foo: "aaa", bar: "bbb" };
baz // "aaa"
foo // error: foo is not defined
```

上面代码中， foo 是匹配的模式， baz 才是变量。真正被赋值的是变量 baz ，而不是模式 foo 。

与数组一样，解构也可以用于嵌套结构的对象。

```
let obj = {
  p: [
    'Hello',
    { y: 'World' }
  ]
};

let { p: [x, { y }] } = obj;
x // "Hello"
y // "World"
```

注意，这时 p 是模式，不是变量，因此不会被赋值。如果 p 也要作为变量赋值，可以写成下面这样。

```
let obj = {
  p: [
    'Hello',
    { y: 'World' }
  ]
};

let { p, p: [x, { y }] } = obj;
x // "Hello"
y // "World"
p // ["Hello", {y: "World"}]
```

下面是另一个例子。

```
const node = {
  loc: {
    start: {
      line: 1,
      column: 5
    }
  }
};

let { loc, loc: { start }, loc: { start: { line }} } = node;
line // 1
loc  // Object {start: Object}
start // Object {line: 1, column: 5}
```

上面代码有三次解构赋值，分别是对 loc 、 start 、 line 三个属性的解构赋值。注意，最后一次对 line 属性的解构赋值之中，只有 line 是变量， loc 和 start 都是模式，不是变量。

下面是嵌套赋值的例子。

```
let obj = {};
let arr = [];

({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });

obj // {prop:123}
arr // [true]
```

对象的解构也可以指定默认值。

```
var {x = 3} = {};
x // 3

var {x, y = 5} = {x: 1};
x // 1
y // 5

var {x: y = 3} = {};
y // 3

var {x: y = 3} = {x: 5};
y // 5

var { message: msg = 'Something went wrong' } = {};
msg // "Something went wrong"
```

默认值生效的条件是，对象的属性值严格等于 undefined 。

```
var {x = 3} = {x: undefined};
x // 3

var {x = 3} = {x: null};
x // null
```

上面代码中，如果 x 属性等于 null ，就不严格相等于 undefined ，

如果解构失败，变量的值等于 `undefined` 。

```
let {foo} = {bar: 'baz'};
foo // undefined
```

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

```
// 报错
let {foo: {bar}} = {baz: 'baz'};
```

上面代码中，等号左边对象的 `foo` 属性，对应一个子对象。该子对象的 `bar` 属性，解构时会报错。原因很简单，因为 `foo` 这时等于 `undefined` ，再取子属性就会报错，请看下面的代码。

```
let _tmp = {baz: 'baz'};
_tmp.foo.bar // 报错
```

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

```
// 错误的写法
let x;
{x} = {x: 1};
// SyntaxError: syntax error
```

上面代码的写法会报错，因为 JavaScript 引擎会将 `{x}` 理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免 JavaScript 将其解释为代码块，才能解决这个问题。

```
// 正确的写法
let x;
({x} = {x: 1});
```

上面代码将整个解构赋值语句，放在一个圆括号里面，就可以正确执行。关于圆括号与解构赋值的关系，参见下文。

解构赋值允许等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式。

```
({} = [true, false]);
({} = 'abc');
({} = []);
```

上面的表达式虽然毫无意义，但是语法是合法的，可以执行。

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

```
let { log, sin, cos } = Math;
```

上面代码将 `Math` 对象的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。

由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

```
let arr = [1, 2, 3];
let {0 : first, [arr.length - 1] : last} = arr;
first // 1
last // 3
```

上面代码对数组进行对象解构。数组 `arr` 的 `0` 键对应的值是 `1` ， `[arr.length - 1]` 就是 `2` 键，对应的值是 `3` 。方括号这种写法，属于“属性名表达式”，参见《对象的扩展》一章。

---

3. 字符串的解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

```
const [a, b, c, d, e] = 'hello';
a // "h"
b // "e"
c // "l"
d // "l"
e // "o"
```

类似数组的对象都有一个 `length` 属性，因此还可以对这个属性解构赋值。

```
let {length : len} = 'hello';
len // 5
```

---

4. 数值和布尔值的解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

```
let {toString: s} = 123;
s === Number.prototype.toString // true

let {toString: s} = true;
s === Boolean.prototype.toString // true
```

上面代码中，数值和布尔值的包装对象都有 `toString` 属性，因此变量 `s` 都能取到值。

解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于 `undefined` 和 `null` 无法转为对象，所以对它们进行解构赋值，都会报错。

```
let { prop: x } = undefined; // TypeError
let { prop: y } = null; // TypeError
```

---

5. 函数参数的解构赋值

函数的参数也可以使用解构赋值。

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

add([1, 2]); // 3
```

上面代码中，函数 `add` 的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量 `x` 和 `y` 。对于函数内部的代码来说，它们能感受到的参数就是 `x` 和 `y` 。

下面是另一个例子。

```
[[1, 2], [3, 4]].map(([a, b]) => a + b);
// [ 3, 7 ]
```

函数参数的解构也可以使用默认值。

```
function move({x = 0, y = 0} = {}) {
  return [x, y];
}

move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, 0]
move({}); // [0, 0]
move(); // [0, 0]
```

上面代码中，函数 `move` 的参数是一个对象，通过对这个对象进行解构，得到变量 `x` 和 `y` 的值。如果解构失败， `x` 和 `y` 等于默认值。

注意，下面的写法会得到不一样的结果。

```
function move({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}

move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, undefined]
move({}); // [undefined, undefined]
move(); // [0, 0]
```

上面代码是为函数 `move` 的参数指定默认值，而不是为变量 `x` 和 `y` 指定默认值，所以会得到与前一种写法不同的结果。

`undefined` 就会触发函数参数的默认值。

```
[1, undefined, 3].map((x = 'yes') => x);
// [ 1, 'yes', 3 ]
```

---

6. 圆括号问题

解构赋值虽然很方便，但是解析起来并不容易。对于编译器来说，一个式子到底是模式，还是表达式，没有办法从一开始就知道，必须解析到（或解析不到）等号才能知道。

由此带来的问题是，如果模式中出现圆括号怎么处理。ES6 的规则是，只要有可能导致解构的歧义，就不得使用圆括号。

但是，这条规则实际上不那么容易辨别，处理起来相当麻烦。因此，建议只要有可能，就不要在模式中放置圆括号。

不能使用圆括号的情况

以下三种解构赋值不得使用圆括号。

（1）变量声明语句

```
// 全部报错
let [(a)] = [1];

let {x: (c)} = {};
let ({x: c}) = {};
let {(x: c)} = {};
let {(x): c} = {};

let { o: ({ p: p }) } = { o: { p: 2 } };
```

上面6个语句都会报错，因为它们都是变量声明语句，模式不能使用圆括号。

（2）函数参数

函数参数也属于变量声明，因此不能带有圆括号。

```
// 报错
function f([(z)]) { return z; }
// 报错
function f([z,(x)]) { return x; }
```

（3）赋值语句的模式

```
// 全部报错
({ p: a }) = { p: 42 };
([a]) = [5];
```

上面代码将整个模式放在圆括号之中，导致报错。

```
// 报错
[({ p: a }), { x: c }] = [{}, {}];
```

上面代码将一部分模式放在圆括号之中，导致报错。

可以使用圆括号的情况

可以使用圆括号的情况只有一种：赋值语句的非模式部分，可以使用圆括号。

```
[(b)] = [3]; // 正确
({ p: (d) } = {}); // 正确
[(parseInt.prop)] = [3]; // 正确
```

上面三行语句都可以正确执行，因为首先它们都是赋值语句，而不是声明语句；其次它们的圆括号都不属于模式的一部分。第一行语句中，模式是取数组的第一个成员，跟圆括号无关；第二行语句中，模式是 `p`，而不是 `d`；第三行语句与第一行语句的性质一致。

---

7. 用途

变量的解构赋值用途很多。

（1）交换变量的值

```
let x = 1;
let y = 2;

[x, y] = [y, x];
```

上面代码交换变量 `x` 和 `y` 的值，这样的写法不仅简洁，而且易读，语义非常清晰。

（2）从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

```
// 返回一个数组

function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象

function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
```

（3）函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
```

```
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

（4）提取JSON数据

解构赋值对提取JSON对象中的数据，尤其有用。

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id, status, number);
// 42, "OK", [867, 5309]
```

上面代码可以快速提取 JSON 数据的值。

（5）函数参数的默认值

```
jQuery.ajax = function (url, {
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
  crossDomain = false,
  global = true,
  // ... more config
}) {
  // ... do stuff
};
```

指定参数的默认值，就避免了在函数体内部再写 `var foo = config.foo || 'default foo';` 这样的语句。

（6）遍历Map结构

任何部署了Iterator接口的对象，都可以用 `for...of` 循环遍历。Map结构原生支持Iterator接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
const map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
  console.log(key + " is " + value);
}
// first is hello
// second is world
```

如果只想获取键名，或者只想获取键值，可以写成下面这样。

```
// 获取键名
for (let [key] of map) {
  // ...
}

// 获取键值
for (let [,value] of map) {
  // ...
}
```

（7）输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

```
const { SourceMapConsumer, SourceNode } = require("source-map");
```

留言