**1. 数组的解构赋值**

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];

如果解构不成功，变量的值就等于undefined。

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

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 // 1y // 2

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

a // 1b // 2d // 4

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，或第15点），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（第11点），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（第16点），原生具有 Iterator 接口。解构赋值会依次从这个接口获取值。

1. **默认值**

解构赋值允许指定默认值。

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 [foo = true] = [];

foo // truelet [x, y = 'b'] = ['a']; // x='a', y='b'

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

let [x = 1] = [null];

x // null

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();//会运行函数f

let [x = f()] = [1];//不会运行函数f

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报错理由同上一点函数里使用没有定义过的变量

**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。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 { foo, bar } = { foo: "aaa", bar: "bbb" };

foo // "aaa"bar // "bbb"

如果变量名与属性名不一致，必须写成下面这样。

var { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

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。数组也是如此。

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

注意，采用这种写法时，变量的声明和赋值是一体的。对于let和const来说，变量不能重新声明，所以一旦赋值的变量以前声明过，就会报错。

let foo;

({foo} = {foo: 1}); // 成功

let {foo} = {foo: 1}; // SyntaxError: Duplicate declaration "foo"

let命令下面一行的圆括号是必须的，否则会报错。因为解析器会将起首的大括号{，理解成一个代码块，而不是赋值语句。

对象的解构也可以指定默认值。

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'};

//因为foo这时等于undefined，再取子属性就会报错就像下面这个例子

let \_tmp = {baz: 'baz'};

\_tmp.foo.bar // 报错

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;

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

方括号这种写法，属于“属性名表达式”，参见《对象的扩展》一章（第9点）。

**3. 字符串的解构赋值**

字符串被转换成了一个类似数组的对象。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。

const [a, b, c, d, e] = 'hello';

a // "h"

b // "e"

c // "l"

d // "l"

e // "o"

let {length : len,slice} = 'hello';

len // 5

slice //function slice() { [native code] }

slice === String.prototype.slice //true

**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

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**

function add([x, y]){

return x + y;

}

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

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。

函数参数的解构也可以使用默认值。

//第一个方法是，传入的对象没有x，y的属性时，使用自定义，而传入为空的时候，使用默认的x，y。

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]

//第二个方法没有指定默认x，y，传入的参数全等于undefined的时候才会使用后面的默认值。

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]

//下面这个例子，就是上面那个函数的简化版。

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

// [ 1, 'yes', 3 ]

1. **圆括号问题**

以下三种解构赋值不得使用圆括号。

1. 变量声明语句中，不能带有圆括号。

// 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 } };

1. 函数参数中，模式不能带有圆括号。

// 报错function f([(z)]) { return z; }

1. 赋值语句中，不能将整个模式，或嵌套模式中的一层，放在圆括号之中。

// 全部报错

({ p: a }) = { p: 42 };

([a]) = [5];

可以使用情况

[(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。

1. **用途**
2. ****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。

1. ****从函数返回多个值****

// 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();

1. ****函数参数的定义****

// 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});

1. ****提取JSON数据****

let jsonData = {

id: 42,

status: "OK",

data: [867, 5309]};

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

console.log(id, status, number);

// 42, "OK", [867, 5309]

1. ****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};

1. ****遍历Map结构****

var 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

// 获取键名for (let [key] of map) {

// ...}

// 获取键值for (let [,value] of map) {

// ...}

1. ****输入模块的指定方法****

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