Es6 - ruanyifeng

let：

声明局部变量，声明过后不可再声明该变量

var会发生变量提升，let不会，所以一定要在声明之后使用

let会产生暂时性死区，即该变量在前面使用了var声明，又在后面使用let声明，并且在let声明之前使用了该变量，导致报错，这就是临时性死区

块级作用于内声明函数，类似于let声明。避免在块级作用域内声明函数，如果需要，最好写成let f = function() {...}

const：

const声明变量必须同时初始化。除此之外const同let，也存在临时性死区等。对于复合型变量，const变量不指向变量，而是指向变量地址，变量不可变，但是变量指向的值可变。所以如果要想变量真的无法改变，可以使用Object.freeze方法。

const foo = Object.freeze({});

// 常规模式时，下面一行不起作用；// 严格模式时，该行会报错

foo.prop = 123;

完全冻结对象，需要递归遍历对象的所有属性。

ES6中只有var和function声明的全局变量为全局对象的属性。let、const和class声明的全局变量不是全局对象的属性。

变量解构赋值

var [a, b, c] = [1, 2, 3];

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 // []

等号右边如果不是数组，严格来说是不可遍历的解构，则不能解构赋值。

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"

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() {

var a = 0;

var b = 1;

while (true) {

yield a;

[a, b] = [b, a + b];

}}

var [first, second, third, fourth, fifth, sixth] = fibs();

sixth // 5

解构赋值允许指定默认值。

var [foo = true] = [];

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

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

ES6内部使用===判断，如果默认值不严格等于undefined则默认值不会生效。

默认值如果是一个表达式，则表达式为惰性求值：

function f() {

console.log('aaa');}

let [x = f()] = [1]; // x能取到值，所以f()不会执行

对象的解构赋值：

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。冒号前是**模式**，冒号后才是变量。

var { bar, foo } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

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

baz // undefined

**上面的代码实际上是 var { bar: bar, foo: foo } = { 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"

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

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

f // 'hello'

l // 'world'

let foo;({foo} = {foo: 1}); // 成功

let baz;({bar: baz} = {bar: 1}); // 成功

上面代码中，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。

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"

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。

var arr = [1, 2, 3];

var {0 : first, [arr.length - 1] : last} = arr;

first // 1

last // 3

字符串解构赋值：

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

数值和布尔值的解构：

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，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

函数参数解构：

[[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]

解构中不能使用圆括号的问题：

1. 变量声明语句中，不能带有圆括号。
2. 函数参数中，模式不能带有圆括号。
3. 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。

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。

解构赋值用途：

****（1）交换变量的值****

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

1. ****从函数返回多个值****
2. ****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****

var jsonData = {

id: 42,

status: "OK",

data: [867, 5309]};

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

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};

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。

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

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，往往需要指定输入那些方法。解构赋值使得输入语句非常清晰。

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

# 字符串的扩展

字符的Unicode：

codePointAt():

ES6提供了codePointAt方法，能够正确处理4个字节储存的字符，返回一个字符的码点。

var s = ';

s.codePointAt(0) // 134071

s.codePointAt(1) // 57271

s.charCodeAt(2) // 97

根据编码返回字符：

ES5：String.fromCharCode不能识别32位的UTF-16字符（Unicode编号大于0xFFFF）。

String.fromCharCode(0x20BB7)

// "ஷ"

ES6：

ES6提供了String.fromCodePoint方法，可以识别0xFFFF的字符，弥补了String.fromCharCode方法的不足。在作用上，正好与codePointAt方法相反。

String.fromCodePoint(0x20BB7)

// "String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'

// true

上面代码中，如果String.fromCodePoint方法有多个参数，则它们会被合并成一个字符串返回。

注意，fromCodePoint方法定义在String对象上，而codePointAt方法定义在字符串的实例对象上。

for...of遍历字符串。它可以识别大于0xFFFF编码的文字。而一般的for循环无法满足，可能会把一个汉子拆分成两个其他文字。

ES5：charAt() ， ES6：at()

normalize()：在视觉上将两个字符合并：[http://es6.ruanyifeng.com/#docs/string#normalize](http://es6.ruanyifeng.com/#docs/string" \l "normalize)

includes() ：查看字符串是否包含参数字符串

startsWith()：

endsWith()：

repeat(n)：返回一个将原字符串重复n次的新字符串。

padStart(param1,param2)： param1是字符串最小长度，param2是用于填充的参数

padEnd：同padStart

字符串模板：

var x = 1;var y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

模板字符串之中还能调用函数。

function fn() {

return "Hello World";}

`foo ${fn()} bar`

// foo Hello World bar

**模板字符串的大括号内部，就是执行JavaScript代码**，因此如果大括号内部是一个字符串，将会原样输出。

`Hello ${'World'}`

// "Hello World"

**标签模板：**

放在函数后面作为参数，该函数会接收多个值，第一个参数是一个数组，这个数组包含了标签模板中的非参数部分，从第二个参数开始，就是这个标签模板的参数部分，

alert`123`

// 等同于alert(123)