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）

结构有3种情况

解构不成功（左边多，右边好），解构成功，不完全解构（左边少于右边）。

有数组的解构赋值，也有对象的解构赋值

一：数据的解构

```
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

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

let [a, [b], d] = [1, [2, 3], 4];
a // 1
b // 2
d // 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），那么将会报错。

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，只有当一个数组成员严格等于`undefined`，默认值才会生效。

```
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: y is not
defined
```

上面最后一个表达式之所以会报错，是因为`x`用`y`做默认值时，`y`还没有声明

二：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。

{ log, sin, cos } 是3个对象

```
// 例一
let { log, sin, cos } = Math;

// 例二
const { log } = console;
log('hello') // hello
```

上面代码的例一将`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。例二将`console.log`赋值到`log`变量。

对象的解构也可以指定默认值。默认值生效的条件是，对象的属性值严格等于`undefined`。

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

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

对象解构赋值的应用

比如：处理后端返回的json数据，取出自己想要的字段值

```
let dataJson = {
  title:'abc',
  name:'winne',
  test:[{
    title:"ggg",
    desc:"对象的解构赋值"
  }]
}
//取两个数据
let{title:oneTitle,test:[{title:twoTitle}]}= dataJson;
console.log(oneTitle,twoTitle);
// let{name}=dataJson;   //取一个数据  相当于 let name = dataJson.name

console.log(name);
```

3:字符串的结构赋值

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
```

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 ]
```

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。

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");
```