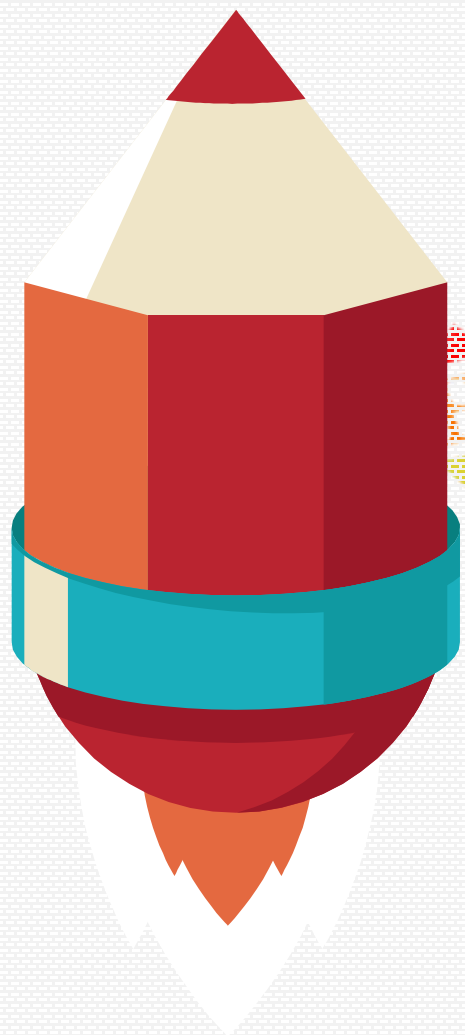




第26课：ES6初级

■ 主讲老师：万章



四知

数组的解构赋值

对象的解构赋值

字符串的解构赋值

数字和布尔值的解构赋值

函数的解构赋值

解构赋值的注意点辨析



数组的解构赋值

解构赋值

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）。

所谓的解构就是解析解构的意思

```
let [a,b,c]=[1,2,3];  
a;//1  
b;//2  
c;//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 // []  
  
let [a, ...b, c] = ['a',3,4,5,"c"];  
//直接报错
```

主要注意点：

1. 赋值等号的左右侧的数组结构需要一直,这个叫做”模式匹配”
2. 如果左右两侧的变量数与数值数的数量不匹配，那么就直接跳过缺失部分(谨记，左边的跳过了，右侧相应位置的也要跳过)
3. 如果左侧某个变量前有三个小点，那么意味着右侧对应位置及以后的所有数值将组合成一个数组,赋值给左侧的该变量,且左侧必须是最后一个变量才可以在前面加三个小点,否则报错

解构赋值

如果解构不成功，对应的变量的值就等于undefined。

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

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。

解构赋值

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），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），要么本身就不具备遍历接口（最后一个表达式）。

解构赋值

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，只有当右侧一个数组成员严格等于undefined，默认值才会生效。（因为要是采用==，那么就会产生数据类型转换了）

```
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。

解构赋值

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];  
}
```


解构赋值

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。



对象的解构赋值

解构赋值

解构不仅可以用于数组，还可以用于对象。

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

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。

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。

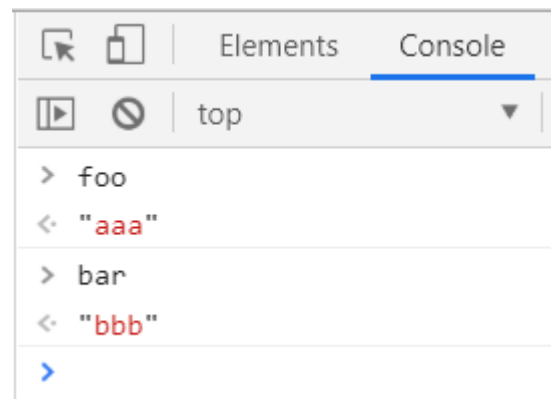
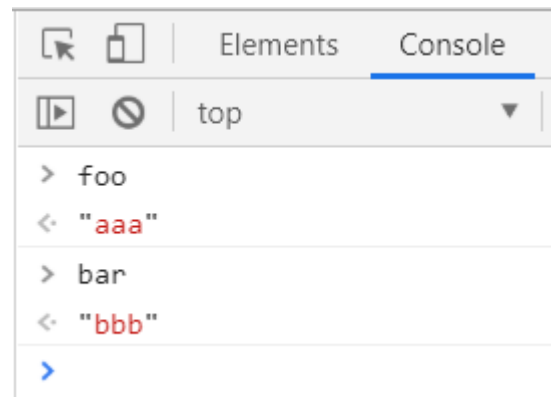
第二个例子的变量没有对应的同名属性，导致取不到值，最后等于undefined。

解构赋值

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

等价于

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



解构赋值

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

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

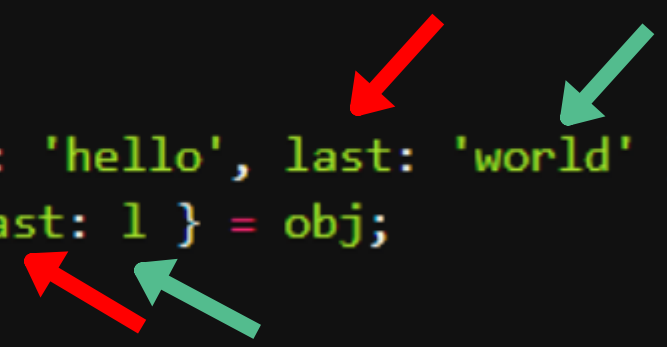
对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

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。

```
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'
```



如果变量名与属性名不一致，必须写成这样。

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。

解构赋值

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

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

与数组一样，解构也可以用于嵌套结构的对象。注意，这时p是模式，不是变量，因此不会被赋值

```
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}
```

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

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

如果p也要作为变量赋值，可以写成这样。
记住数组的解构赋值和对象的解构赋值的不同

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

解构赋值

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

嵌套赋值

```
function Person(){  
    this.name="万章";  
}  
  
Person.prototype.age=18;  
  
let o=new Person();  
  
let {age:age}=o;  
  
console.log(age);//18
```

注意，对象的解构赋值可以取到继承的属性
左侧代码中o的原型中有一个属性名称是age，值是18

所在解构赋值的过程中在进行模式匹配时，左侧的age
会和右侧o的原型中age属性进行匹配
匹配了之后就执行赋值操作

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

上面代码中，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。先要父属性先匹配，再匹配子属性

```
> o  
< ▼ Person {name: "万章"} ⓘ  
  name: "万章"  
  __proto__:  
    age: 18  
    ▶ constructor: f Person()  
    ▶ __proto__: Object  
>
```

对象解构赋值的注意点

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

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

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

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

(1) 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。左侧代码的写法会报错，因为 JavaScript 引擎会将{x}理解成一个代码块(最简单的块级作用域)，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。

(2) 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。上面的表达式虽然毫无意义，但是语法是合法的，可以执行。

(3) 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



字符串的解构赋值

字符串解构赋值

```
const [a, b, c, d, e] = 'hello';  
a // "h"  
b // "e"  
c // "l"  
d // "l"  
e // "o"  
  
let {length : len} = 'hello';  
len // 5
```

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。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



数值和布尔值的解构赋值

数值和布尔值的解构赋值

```
> let {s}=123;
< undefined

> let {toString:str}=123;
< undefined

> str
< f toString() { [native code] }

> let {toString:st}=true;
< undefined

> st
< f toString() { [native code] }

> let {t}=true;
< undefined

> t
< undefined
```

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(临时的包装对象)

左侧代码中，数值和布尔值的包装对象都有toString属性，因此左侧的变量str/st都能取到值。

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

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

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。



函数参数的解构赋值

函数参数的解构赋值

```
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。

```
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]
```

函数参数的解构也可以使用默认值。

这里有两个步骤：

1. 对函数里面的参数进行解构赋值，得到的结果就是{x=0, y=0}
2. 函数调用时进行数据传参，如果传入的参数可以正常进行解构赋值，那么就使用传进的参数否则，就调用默认值

函数参数的解构赋值

```
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]
```

左侧代码是为函数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。

所以函数的默认参数是{x=0, y=0}

一旦这个函数在执行时传入了其他参数，那么这个默认的参数就不起作用

只有当函数在执行不传入任何参数，这个默认参数才有作用



解构赋值的注意点辨析

圆括号的问题

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。

```
// 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) 变量**声明语句**

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。

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

(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。

不能使用圆括号的情况

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

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

(3) **赋值语句的模式**

上面代码将整个模式放在圆括号之中，导致报错。

下面代码将一部分模式放在圆括号之中，导致报错。

圆括号的问题

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。

解构赋值的常见应用场景

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

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

(1) 交换变量的值

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。

```
// 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();
```

(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。

```
// 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});
```

(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。

解构赋值的常见应用场景

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

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

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

（4）提取 JSON 数据

解构赋值对提取 JSON 对象中的数据，尤其有用。

JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parse **JSON**-format data. The official Internet media type for **JSON** is `application/json`. **JSON** filenames use the extension `.json`.

JSON - Wikipedia
<https://en.wikipedia.org/wiki/JSON>



JSON

编程语言



JSON是一种由道格拉斯·克洛克福特构想和设计、轻量级的数据交换语言，该语言以易于让人阅读的文字为基础，用来传输由属性值或者序列性的值组成的数据对象。尽管JSON是JavaScript的一个子集，但JSON是独立于语言的文本格式，并且采用了类似于C语言家族的一些习惯。 [维基百科](#)