**第3章 变量的解构赋值**

3.1 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，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

1. var a = 1;

2. var b = 2;

3. var c = 3;

4.

而ES6允许写成下面这样。

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

2.

上面的代码表示，可以从数组中提取值，按照位置的对应关系对变量赋值。

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。下面是使用一些嵌套数组进行解构的例子。

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

2. foo; // 1

3. bar; // 2

4. baz; // 3

5.

6. let [,,thrid] = ["foo", "bar", "baz"];

7. thrid; // "baz"

8.

9. let [x, ,y] = [1, 2, 3];

10. x; // 1

11. y; // 3

12.

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

14. head; // 1

15. tail; // [2, 3, 4]

16.

17. let [x, y, ...z] = ['a'];

18. x; // "a"

19. y; // undefined

20. z; // []

21.

如果解构不成功，变量的值就等于undefined。

1. var [foo] = [];

2. var [bar, foo] = [1];

3.

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等候右边数组的一部分。这种情况下，解构依然可以成功。

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

2. x // 1

3. y // 2

4.

5. let [a, [b], c] = [1, [2, 3], 4];

6. a // 1

7. b // 2

8. c // 4

9.

上面的两个例子都属于不完全解构，但是可以成功。

如果等候的右边不是数组（或者严格地说，不是可遍历的结构），那么将会报错。

1. // 报错

2. let [foo] = 1;

3. let [foo] = false;

4. let [foo] = NaN;

5. let [foo] = undefined;

6. let [foo] = null;

7. let [foo] = {};

8.

上面的表达式都会报错，因为等号右边的值，要么转为对象以后不具备Iterator接口（前5个表达式），要么本身就不具备Iterator接口（最后一个表达式）。

解构赋值不仅适用于var命令，也适用于let和const命令。

1. var [v1, v2, v3, ..., vN] = array;

2. let [v1, v2, v3, ..., vN] = array;

3. const [v1, v2, v3, ..., vN] = array;

4.

对于set结构，也可以使用数组的解构赋值。

1. let [x, y, z] = new Set(["a", "b", "c"]);

2. x // "a"

3.

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。

1. function\* fibs() {

2.     var a = 0;

3.     var b = 1;

4.     while (true) {

5.         yield a;

6.         [a, b] = [b, a + b];

7.     }

8. }

9.

10. var [first, second, thrid, fourth, sixth] = fibs();

11. sixth // 5

12.

上面的代码中，fibs是一个Generator函数，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。

1. var [foo = true] = [];

2. foo // true

3.

4. [x, y = 'b'] = ['a']; // x='a' y='b'

5. [x, y = 'b'] = ['a', undefined] // x='a' y='b'

6.

注意，ES6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。

1. var [x = 1] = [undefined];

2. x // 1

3.

4. var [x = 1] = [null];

5. x // null

6.

上面的代码中，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。

1. function f() {

2. console.log(‘aaa’);

3. }

4. let [x = f()] = [1];

5.

上面的代码中，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以下代码。

1. let x;

2. if ([1][0] === undefined) {

3.     x = f();

4. } else {

5.     x = [1][0];

6. }

7.

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。

1. let [x = 1, y = x] = [];        // x = 1, y = 1

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

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

4. let [x = y, y = 1] = [];        // ReferenceError

5.

上面的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。

3.2 对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

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

2. foo // "aaa"

3. bar // "bbb"

4.

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。

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

2. foo // "aaa"

3. bar // "bbb"

4.

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

6. baz // undefined

7.

以上代码中的第一个例子，等号左边两个变量的次序与等号右边两个同名属性不一致，但是对取值完全没有影响。第二个例子中，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。

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

2. baz // "aaa"

3.

4. let obj = { first: "hello", last: "world" };

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

6. f // "hello"

7. l // "world"

8.

这实际上说明，对象的解构赋值是以下形式的简写（参见第9章）。

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

2.

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

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

2. baz // "aaa"

3. foo // error: foo is not defined

4.

上面的代码中，真正被赋值的是变量baz，而不是模式foo。

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而言，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。

1. let foo;

2. let {foo} = { foo: 1 }; // Syntax: Duplicate declaration “foo”

3.

4. let baz;

5. let { bar: baz } = { bar: 1 }; // SyntaxError: Duplicate declaration “foo”

6.

上面的代码中，解构赋值的变量都会重新声明，所以报错了。不过，因为var命令允许重新声明，所以这个错误只会在使用let和const命令时出现。如果没有第二个let命令，上面的代码就不会报错。

1. let foo;

2. ({foo} = {foo: 1}); // success

3.

4. let baz;

5. ({bar: baz} = {bar: 1}); // success

6.

和数组一样，解构也可以用于嵌套结构的对象。

1. var obj = {

2.     p: [

3.         "Hello",

4.         {

5.             y: "World"

6.         }

7.     ]

8. };

9.

10. var { p: [x, { y }]} = obj;

11.

12. x // Hello

13. y // World

14.

注意，这时p是模式，不是变量，因此不会被赋值。

1. var node = {

2.     loc: {

3.         start: {

4.             line: 1,

5.             column: 5

6.         }

7.     }

8. };

9.

10. var { loc: { start: { line }}} = node;

11. line // 1

12. loc // Error: loc is undefined

13. start // Error: start is undefined

14.

上面的代码中，只有line是变量，loc和start都是模式，不会被赋值。

下面是嵌套赋值的例子。

1. let obj = {};

2. let arr = [];

3.

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

5.

6. obj // {prop: 123}

7. arr // [true]

8.

对象的解构也可以指定默认值。

1. var {x = 3} = {};

2. x // 3

3.

4. var {x, y = 1} = {x: 5};

5. x // 5

6. y // 1

7.

8. var {message: msg = "Something went wrong"} = {};

9. msg // "Something went wrong"

10.

默认值生效的条件是，对象的属性值严格等于undefined。

1. var {x = 3} = { x: undefined };

2. x // 3

3.

4. var {x = 3} = { x: null };

5. x // null

6.

上面的代码中，如果x属性等于null，就不严格等于undefined，导致默认值不会生效。

如果解构失败，变量的值等于undefined。

1. var {foo} = {bar: "baz"};

2. foo // undefined

3.

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。

1. // 报错

2. var { foo: { bar } } = { baz: 'baz'};

3.

上面的代码中，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。

1. var \_tmp = { baz: "baz" };

2. \_tmp.foo.bar // 报错

3.

如果将一个已声明的变量用于解构赋值，必须非常小心。

1. // 错误的写法

2. var x;

3. {x} = {x: 1};

4. // SyntaxError: syntax error

5.

上面的代码写法会报错，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。

1. // 正确的写法

2. ({x} = { x: 1 });

3.

上面的代码将整个解构赋值语句放在了一个圆括号里面，就可以正确执行。关于圆括号与解构赋值的关系，参见上下文。

解构赋值允许等号左边的模式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。

1. ({} = [true, false]);

2. ({} = "abc");

3. ({} = []);

4.

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。

1. let { log, sin, cos } = Math;

2.

上面的代码将Math对象的取对数、正弦、余弦3个方法赋值到了对应的变量上，这样使用起来会方便很多。

3.3 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。

1. const [a, b, c, d, e] = "Hello";

2.

类似数组的对象都有length属性，因此还可以对这个属性解构赋值。

1. let {length: len} = "Hello";

2. len // 5

3.

3.4 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或布尔值，则会先转为对象。

1. let {toString: s} = 123;

2. s === Number.prototype.toString // true

3.

4. let {toString: s} = true;

5. s === Boolean.prototype.toString // true

6.

上面的代码中，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。

1. let {prop: x} = undefined; // TypeError

2. let {prop: y} = null; // TypeError

3.

3.5 函数参数的解构赋值

函数的参数也可以解构赋值。

1. function add([x, y]) {

2.     return x + y;

3. }

4.

5. add([1, 3]); // 4

6.

上面的代码中，函数add的参数实际上不是一个数组，而是通过解构得到的变量x和y。

下面的另一个例子。

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

2. // [3, 5]

3.

函数参数的解构也可以使用默认值。

1. function move({x = 0, y = 0} = {}) {

2.     return [x, y];

3. }

4.

5. move({x: 3, y: 8}); // [3, 8]

6. move({x: 3}); // [3, 0]

7. move({}); // [0, 0]

8. move(); // [0, 0]

9.

上面的代码中，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，则x和y等于默认值。

注意，下面的写法会得到不一样的结果。

1. function move({x, y} = {x: 0, y: 0}) {

2.     return [x, y];

3. }

4.

5. move({x: 3, y: 8}); // [3, 8]

6. move({x: 3}); // [3, undefined]

7. move({}); // [undefined, undefined]

8. move(); // [0, 0]

9.

上面的代码是为函数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会触发函数参数的默认值。

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

2. // [1, “yes”, 3]

3.

3.6 圆括号问题

解构赋值虽然很方便，但是解析并不容易。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3中解构赋值不得使用圆括号。

1. 变量声明语句中，模式不能带有圆括号。

1. // 全部报错

2. var [(a)] = [1];

3. var { x: (c) } = {};

4. var { o: ({ p: p}) } = {o: {p: 2}};

5.

上面三条语句都会报错，因为它们都是变量声明语句，模式不能带有圆括号。

1. 函数参数中，模式不能带有圆括号。

函数参数也属于变量声明，因此不能带有圆括号。

1. // 报错

2. function f([(z)]) {

3.     return z;

4. }

5.

1. 不能将整个模式或嵌套模式中的一层放在圆括号中。

1. // 全部报错

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

3. ([a]) = [5];

4.

5.

上面的代码将整个模式放在圆括号中，导致报错。

1. // 报错

2. [({ p: a }), { x: c }] = [{}, {}];

3.

上面的代码将嵌套模式的一层放在了圆括号中，导致报错。

**可以使用圆括号**

可以使用圆括号的请阔只有一种：赋值语句的非模式部分可以使用圆括号。

1. [(b)] = [2]; // 正确

2. ({ p: (d) } = {}); // 正确

3. [(parseInt.prop)] = [3]; // 正确

4.

上面3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。

3.7 用途

变量的解构赋值用途很多。

**交换变量值**

1. [x, y] = [y, x];

2.

上面的代码交换了变量x和有的值，这样的写法不仅简洁，而且易读，语义非常清晰。

**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，只能将其放在数组或对象中返回。有了解构赋值，取出这些值就非常方便。

1. // 返回一个数组

2. function example() {

3.     return [1, 2, 3];

4. }

5. var [a, b, c] = example();

6.

7. // 返回一个对象

8. function example() {

9.     return {

10.         foo: 1,

11.         bar: 2

12.     };

13. }

14. var { foo, bar } = example();

15.

**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。

1. // 参数是一组有次序的值

2. function f([x, y, z]) { ... }

3. f([1, 2, 3]);

4.

1. // 参数是一组无次序的值

2. function f({x, y, z}) { ... }

3. 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。

1. var jsonData = {

2.     id: 42,

3.     status: "ok",

4.     data: [867, 5309]

5. }

6.

7. let {id, status, number} = jsonData;

8. console.log(id, status, number);

9. // 42, ok, [867, 5309]

10.

上面的代码可以快速提取JSON数据的值。

**函数参数的默认值**

1. jQuery.ajax = function (url, {

2.     async = true,

3.     beforeSend = function () {},

4.     cache = true,

5.     complete = function () {},

6.     crossDoamin = false,

7.     global = true,

8.     // ...more config

9. }) {

10.     // ...do stuff

11. }

12.

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。

**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。

1. var map = new Map();

2. map.set("first", "hello");

3. map.set("second", "world");

4.

5. for (let [key, value] of map) {

6.     console.log(key + " is " +value);

7. }

8.

9. // first is hello

10. // second is world

11.

如果指向获取键名，或者只想获取键值，可以写成下面这样。

1. // 获取键名

2. for (let [key] of map) { // … }

3. // 获取键值

4. for (let [value] of map) { // … }

5.

**输入模块的指定方法**

加载模块时，往往需要指定哪些方法。解构赋值使得输入语句非常清晰。

1. const { SourceMapConsumer, SourceNode } = require(“source-map”);

2.