**第8章 函数的扩展**

**8.1 函数参数的默认值**

**基本用法**

在ES6之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

1. function log(x, y) {

2.     y = y || "world";

3.     console.log(x, y);

4. }

5.

6. log("hello"); // hello world

7. log("Hello", "China"); // Hello China

8. log("Hello",  ""); // Hello World

9.

上面的代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像以上代码的最后一行，参数y等于空字符，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数y是否被赋值，如果没有，再等于默认值。这有两种写法。

1. // 写法1

2. if (typeof y === 'undefined') {

3.     y = "world";

4. }

5.

6. // 写法2

7. if (arguments.length === 1) {

8.     y = "world";

9. }

10.

ES6允许为函数的参数设置默认值，即直接写在参数定义的后面。

1. function log(x, y = "World") {

2.     console.log(x, y);

3. }

4.

5. log("Hello", y); // Hello World

6. log("Hello", "China"); // Hello China

7. log("Hello", ""); // Hello

8.

可以看到，ES6的写法比ES5简洁许多，而且非常自然。下面是另一个例子。

1. function Point(x = 0, y = 0) {

2.     this.x = x;

3.     this.y = y;

4. }

5.

6. var p = new Point();

7. p // {x: 0, y: 0}

8.

除了简洁，ES6的写法还有两个好处：首先，阅读代码的人可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本彻底拿掉这个参数，也不会导致以前的代码无法运行。

参数变量是默认声明的，所以不能用let或const再次声明。

1. function foo(x = 5) {

2.     let x = 1; // error

3.     const x = 2; //error

4. }

5.

上面的代码中，参数变量x是默认声明的，在函数体中不能用let或const再次声明，否则会报错。

**与解构赋值默认值结合使用**

参数默认值可以与解构赋值的默认值结合起来使用。

1. function foo({x, y = 5}) {

2.     console.log(x, y);

3. }

4.

5. foo({}); // undefined, 5

6. foo({x: 1}); // 1, 5

7. foo({x: 1, y: 2}); // 1, 2

8. foo(); // TypeError: Cannot read property 'x' of undefined

9.

上面的代码使用了对象的解构赋值默认值，而没有使用函数参数的默认值。只有当函数foo的参数是一个对象时，变量x和y才会通过解构赋值而生成。如果函数foo调用时参数不是对象，变量x和y就不会生成，从而报错。只有参数对象没有y属性时，y的默认值5才会生效。

下面是另一个对象的解构赋值默认值的例子。

1. function fetch(url, { body= "", method="get", headers={} }) {

2.     console.log(method);

3. }

4.

5. fetch("http://example.com", {});

6. // "get"

7.

8. fetch("http://example.com");

9. // 报错

10.

上面的代码中，如果函数fetch的第二个参数是一个对象，就可以为它的3个属性设置默认值。上面的写法不能省略第二个参数，如果结合函数参数的默认值，就可以省略第二个参数。这时，就出现了双重默认值。

1. function fetch(url, { method = "get" } = {}) {

2.     console.log(method);

3. }

4.

5. fetch("http://example.com")

6. // "GET"

7.

上面的代码中，函数fetch没有第二个参数时，函数参数的默认值就会生效，然后才是解构赋值的默认值生效，变量method取到默认值GET。

那么下面两种写法有什么差别呢？

1. // 写法1

2. function m1({x = 0, y = 0} = {}) {

3.     return [x, y];

4. }

5.

6. // 写法2

7. function m2({x, y} = {x: 0, y: 0}) {

8.     return [x, y];

9. }

10.

上面两种写法都对函数的参数设定了默认值，区别是：写法一中函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二中函数参数的默认值是一个有具体属性的函数，但是没有设置对象解构赋值的默认值。

1. // 函数没有参数的情况

2. m1() // [0, 0]

3. m2() // [0, 0]

4.

5. // x和y都有值得情况

6. m1({x: 3, y: 8}) // [3, 8]

7. m2({x: 3, y: 8}) // [3, 8]

8.

9. // x有值，y无值得情况

10. m1({x: 3}) // [3, 0]

11. m2({x: 3}) // [3, undefined]

12.

13. // x和y都无值的情况

14. m1({}) // [0, 0]

15. m2({}) // [undefined, undefined]

16.

17. m1({z: 3}) // [0, 0]

18. m1({z: 3}) // [undefined, undefined]

19.

**参数默认值的位置**

通常情况下，定义了默认值的参数应该是函数的尾参数。因为这样比较容易看出，到底省略了哪些参数。如果非尾部的参数设置默认值，实际上这个参数是无法省略的。

1. // 例一

2. function f(x=1, y) {

3.     return [x, y];

4. }

5.

6. f() // [1, undefined]

7. f(2) // [2, undefined]

8. f(, 1) // 报错

9. f(undefined, 1) // [1, 1]

10.

11. // 例二

12. function f(x, y = 5, z) {

13.     return [x, y, z];

14. }

15.

16. f() // [undefined, 5, undefined]

17. f(1) // [1, 5, undefined]

18. f(1, , 2) // 报错

19. f(1, undefined, 2) // [1, 5, 2]

20.

上面的代码中，有默认值的参数都不是尾参数。这时，无法只省略该参数而不省略其后的参数，除非显示输入undefined。

如果传入undefined，将触发该参数等于默认值，null没有这个效果。

1. function foo(x = 5, y = 6) {

2.     console.log(x, y);

3. }

4.

5. foo(undefined, null)

6. // 5 null

7.

上面的代码中，x参数对应undefined，结果触发了默认值，y参数等于null，就没有触发默认值。

**函数的length属性**

指定了默认值以后，函数的length属性将返回没有指定默认值的参数个数。也就是说，指定了默认值后，length属性将失真。

1. (function (a) {}).length // 1

2. (function (a = 5) {}).length // 0

3. (function (a, b, c = 5) {}).length // 2

4.

上面的代码中，length属性的返回值等于函数的参数个数减去指定了默认值的参数个数。比如，上面的最后一个函数，定义了3个参数，其中有一个参数指定了默认值，因此length属性等于3减去1，即2。

这是因为length属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。同理，rest参数也不会计入length属性。

1. (function (...args) {}).length // 0

2.

**作用域**

一个需要注意的地方是，如果参数默认值是一个变量，则该变量所处的作用域与其他变量的作用域规则是一样的，即先是当前函数的作用域，然后才是全局作用域。

1. var x = 1;

2. function f(x, y = x) {

3.     console.log(y);

4. }

5.

6. f(2) // 2

7.

上面的代码中，参数y的默认值等于x。调用时，由于函数的作用域内部的变量x已经生成，所以y等于参数x而不是全局变量x。

如果调用时函数作用域内部的变量x没有生成，结果就会不一样。

1. let x = 1;

2. function f(y = x) {

3.     let x = 2;

4.     console.log(y);

5. }

6.

7. f() // 1

8.

上面的代码中，函数调用时y的默认值变量x尚未在函数内部生成，所以x指向全局变量。

如果此时全局变量x不存在就会报错。

1. function f(y = x) {

2.     let x = 2;

3.     console.log(y);

4. }

5.

6. f() // ReferenceError: x is not defined

7.

如果函数A的参数默认值是函数B，那么由于函数的作用域是其声明时所在的作用域，函数B的作用域就不是函数A，而是全局作用域。请看下面的例子。

1. let foo = "outer";

2. function bar(func = x => foo) {

3.     let foo = "inner";

4.     console.log(func()); // outer

5. }

6.

7. bar();

8.

上面的代码，函数bar的参数func默认是一个匿名函数，返回值为变量foo。这个匿名函数的作用域就不是bar。这个匿名函数声明时是处在外层作用域，所以内部的foo指向函数体外的声明，输出outer。它实际上等同于下面的代码。

1. let foo = "outer";

2. let f = x => foo;

3.

4. function bar(func = f) {

5.     let foo = "inner";

6.     console.log(func()); // outer

7. }

8.

9. bar();

10.

如果写成下面这样，就会报错。

1. function bar(func = () => foo) {

2.     let foo = "inner";

3.     console.log(func());

4. }

5.

6. bar(); // ReferenceError: foo is not defined

7.

**应用**

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

1. function throwIfMissing() {

2.     throw new Error("Missing parameter");

3. }

4.

5. function foo(mustBeProvided = throwIfMissing()) {

6.

7.     return mustBeProvided;

8. }

9.

10. foo();

11. // Error: Missing parameter

12.

以上代码中的foo函数，如果调用的时候没有参数，就会调用默认值throwIfMissing函数，从而抛出一个错误。

从上面的代码还可以看到，参数mustBeProvided的默认值等于throwIfMissing函数的运行结果（即函数名之后有一对圆括号），这表明参数的默认值不是在定义时执行，而是在运行时执行（即如果参数已经赋值，默认值中的函数就不会运行），这与Python语言不一样。

另外，可以将参数默认值设置为undefined，表明这个参数是可以省略的。

1. function foo(optional = undefined) { ... }

2.

**8.2 rest参数**

ES6引入了rest参数（形式为“…变量名”），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest参数搭配的变量是一个数组，该变量将多余的参数放入其中。

1. function add(...values) {

2.     let sum = 0;

3.     for (var val of values) {

4.         sum += val;

5.     }

6.     return sum;

7. }

8.

9. add(2, 5, 3);

10.

以上代码中的add函数是一个求和函数，利用rest参数可以向该函数传入任意数目的参数。下面是一个rest参数代替arguments变量的例子。

1. // arguments变量的写法

2. const sortNumbers = () => {

3.     Array.prototype.slice.call(arguments).sort();

4. }

5.

6. // rest参数的写法

7. const sortNumbers = (...numbers) => numbers.sort();

8.

比较上面的两种写法可以发现，rest参数的写法更自然也更简洁。

rest参数中的变量代表一个数组，所以数组特有的方法都可以用于这个变量。下面是一个利用rest参数改写数组push方法的例子。

1. function push(array, ...items) {

2.     items.forEach(function (item) {

3.         array.push(item);

4.         console.log(item);

5.     });

6. }

7.

8. var a = [];

9.

10. push(a, 1, 2, 3);

11.

注意，rest参数之后不能再有其他参数（即只能是最后一个参数），否则会保存。

1. // 报错

2. function f(a, ...b, c) {

3.     // ...

4. }

5.

函数的length属性不包括rest参数。

1. (function(a) {}).length; // 1

2. (function(...a) {}).length; // 0

3. (function(a, ...b) {}).length; // 1

4.

**8.3 扩展运算符**

**含义**

扩展运算符（spread）是三个点（…）。它好比rest参数的逆运算，将一个数组转为用逗号分隔的参数序列。

1. console.log(...[1,2,3]);

2. // 1 2 3

3.

4. console.log(1, ...[2,3,4], 5);

5. // 1 2 3 4 5

6.

7. [...document.querySelectorAll("div")];

8. // [<div>, <div>, <div>]

9.

该运算符主要用于函数调用。

1. function push(array, ...items) {

2.     array.push(...items);

3. }

4.

5. function add(x, y) {

6.     return x + y;

7. }

8.

9. var numbers = [4, 38];

10. add(...numbers) // 42

11.

上面的代码中，array.push(…items)和add(…numbers)这两行都是函数的调用，它们都使用了扩展运算符，该运算符将一个数组变为参数序列。

扩展运算符与正常的函数参数可以结合使用，非常灵活。

1. function f(v, w, x, y, z) {}

2. var args = [0, 1];

3.

4. f(-1, ...args, 2, ...[3]);

5.

**替代数组的apply方法**

由于扩展运算符可以展开数组，所以不在需要apply方法将数组转为函数的参数了。

1. // ES5的写法

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

3. var args = [0, 1, 2];

4. f.apply(null, args);

5.

6. // ES6的写法

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

8. var args = [0, 1, 2];

9. f(...args);

10.

下面是扩展运算符取代apply方法的一个实际的例子，应用Math.max方法简化求数组最大元素的写法。

1. // ES5的写法

2. Math.max.apply(null, [14, 3, 77]);

3.

4. // ES6的写法

5. Math.max(...[14, 3, 77]);

6.

7. // 等同于

8. Math.max(14, 3, 77);

9.

上面的代码表示，由于JavaScript不提供求数组最大元素的函数，所以只能套用Math.max函数将数组转为一个参数序列，然后求最大值。有了扩展运算符以后，就可以直接用Math.max了。

另一个例子是通过push函数将一个数组添加到另一个数组的尾部。

1. // ES5的写法

2. var arr1 = [0, 1, 2];

3. var arr2 = [3, 4, 5];

4. Array.prototype.push.apply(arr1, arr2);

5.

6. // ES6的写法

7. var arr1 = [0, 1, 2];

8. var arr2 = [3, 4, 5];

9. arr1.push(...arr2);

10.

上面的ES5写法中，push方法的参数不能是数组，所以只好通过apply方法变通使用push方法。有了扩展运算符，就可以直接将数组传入push方法。

下面是另一个例子。

1. // ES的写法

2. new (Date.bind.apply(Date, [null, 2015, 1, 1]));

3.

4. // ES6

5. new Date(...[2015, 1, 1]);

6.

**扩展运算符的应用**

1. **合并数组**

扩展运算符提供了数组合并的新写法。

1. // ES5

2. [1, 2].concat(more);

3.

4. // ES6

5. [1, 2, ...more];

6.

7. var arr1 = ['a', 'b'];

8. var arr2 = ['c'];

9. var arr3 = ['d', 'e'];

10.

11. // ES5的合并数组

12. arr1.concat(arr2, arr3);

13. // ['a', 'b', 'c', 'd', 'e']

14.

15. // ES6的合并数组

16. [...arr1, ...arr2, ...arr3]

17. // ['a', 'b', 'c', 'd', 'e']

18.

1. **与解构赋值结合**

扩展运算符可以与解构赋值结合起来用于生成数组。

1. // ES5

2. a = list[0], rest = list.slice(1);

3.

4. // ES6

5. [a, ...rest] = list;

6.

下面是另外一些例子。

1. const [first, ...rest] = [1, 2, 3, 4, 5];

2. first // 1

3. rest // 2, 3, 4, 5

4.

5. const [first, ...rest] = [];

6. first // undefined

7. rest // []

8.

9. const [first, ...rest] = ["foo"];

10. first // "foo"

11. rest // []

12.

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

1. const [...butLast, last] = [1, 2, 3, 4, 5];

2. // 报错

3.

4. const [first, ...butLast, last] = ["foo"];

5. // 报错

6.

1. **函数的返回值**

JavaScript的函数只能返回一个值，如果需要返回多个值，只能返回数组或对象。扩展运算符提供了解决这个问题的一种变通方法。

1. var dateFields = readDateFields(database);

2. var d = new Date(...dateFields);

3.

上面的代码从数据库取出一行数据，通过扩展运算符直接将其传入构造函数Date。

1. **字符串**

扩展运算符还可以将字符串转为真正的数组。

1. [...'hello']

2. // ['h', 'e', 'l', 'l', 'o']

3.

上面的写法有一个重要的好处，那就是能够正确识别32位的Unicode字符。

1. 'x\uD83D\uDE80y'.length; // 4

2. [...'x\uD83D\uDE80y'].length; // 3

3.

上面的第一种写法，JavaScript会将32位的Unicode字符识别为2个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数可以像下面这样写。

1. function length(str) {

2.     return [...str].length;

3. }

4. length("x\uD83D\uDE80y") // 3

凡是涉及操作32位Unicode字符的函数都有这个问题。因此，最好都用扩展运算符改写。

1. let str = 'x\uD83D\uDE80y';

2. str.split('').reverse().join('');

3. // 'y\uDE80\yD83Dx'

4.

5. [...str].reverse().join('')

6. // 'y\uD83D\uDE80x

7.

上面的代码中，如果不用扩展运算符，字符串的reverse操作就不正确。

1. **类似数组的对象**

任何类似数组的对象都可以用扩展运算符转为真正的数组。

1. var nodeList = document.querySelectorAll("div");

2. var array = [...nodeList];

3.

上面的代码中，querySelectorAll方法返回的是一个nodeList对象，扩展运算符可以将其转为真正的数组。

1. **Map和Set解构，Generator函数**

扩展运算符内部调用的是数据结构的Iterator接口，因此只要具有Iterator接口的对象，都可以使用扩展运算符，比如Map结构。

1. let map = new Map([

2.     [1, "one"],

3.     [2, "two"],

4.     [3, "three"]

5. ]);

6.

7. let arr = [...map.keys()]; // 1, 2, 3

8.

Generator函数运行后返回一个遍历器对象，因此也可以使用扩展运算符。

1. var go = function\*() {

2.     yield 1;

3.     yield 2;

4.     yield 3;

5. };

6.

7. [...go()]; // 1, 2, 3

8.

上面的代码中，变量go是一个Generator函数，执行后返回的是一个遍历器对象，对这个遍历器对象执行扩展运算符，就会将内部遍历得到的值转为一个数组。

如果对没有iterator接口的对象使用扩展运算符，将会报错。

1. var obj = {

2.     a: 1,

3.     b: 2

4. };

5.

6. let arr = [...obj]; // TypeError: object is not iterable (cannot read property Symbol(Symbol.iterator))

7.

**8.4 name属性**

函数的name属性返回该函数的函数名。

1. function foo() {}

2. foo.name // "foo"

3.

这个属性早就被浏览器广泛支持，但是直到ES6才写入了标准。

需要注意的是，ES6对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5的name属性会返回空字符串，而ES6的name属性会返回实际的函数名。

1. var fun1 = function () {}

2.

3. // ES5

4. fun1.name; // ""

5.

6. // ES6

7. fun1.name; // "fun1"

8.

上面的代码中，变量func1等于一个匿名函数，ES5和ES6的name属性返回的值不一样。

如果将一个具名函数赋值给一个变量，则ES5和ES6的name属性都返回这个具名函数原本的名字。

1. const bar = function baz() {}

2.

3. // ES5

4. bar.name; // "baz"

5.

6. // ES6

7. bar.name; // "baz"

8.

Function构造函数返回的函数实例，name属性的值为“anonymous“。

1. (new Function).name; // "anonymous"

2.

bind返回的函数，name属性值会加上“bound“前缀。

1. function foo() {}

2. foo.bind({}).name; // "bound foo"

3.

4. (function(){}).bind({}).name; // bound

5.

**8.5 箭头函数**

**基本用法**

ES6允许使用“箭头”（=>）定义函数。

1. var f = v => v;

2.

上面的箭头函数等同于：

1. var f = function (v) {

2.     return v;

3. };

4.

如果箭头函数不需要参数或需要多个参数，就使用圆括号代表参数部分。

1. var f = () => 5;

2.

3. // 等同于

4. var f = function () {

5.     return 5;

6. };

7.

8. var sum = (num1, num2) => num1 + num2;

9.

10. // 等同于

11. var sum = function (num1, num2) {

12.     return num1 + num2;

13. };

14.

如果箭头函数的代码块部分多余一条语句，就要使用大括号将其括起来，并使用return语句返回。

1. var sum = (num1, num2) => {

2.     return num1 + num2;

3. }

4.

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号。

1. var getTempItem = id => ({id: id, name: "Temp"});

2.

箭头函数可以与变量解构结合使用。

1. const full = ({first, last}) => first + " " + last;

2.

3. // 等同于

4. function full(person) {

5.     return person.first + " " + person.last;

6. }

7.

箭头函数使得表达更加简洁。

1. const isEven = n => n % 2 == 0;

2. const square = n => n \* n;

3.

上面的代码只用了两行就定义了两个简单的工具函数。如果不用箭头函数，可能就要占用多行，而且还不如现在这样写醒目。

箭头函数得一个用处是简化回调函数。

1. // 正常函数写法

2. [1,2,3].map(function (x) {

3.     return x \* x;

4. });

5.

6. // 箭头函数写法

7. [1,2,3].map(x => x \*x);

8.

下面是另一个例子。

1. // 正常函数写法

2. var result = values.sort(function (a, b) {

3.     return a - b;

4. });

5.

6. // 箭头函数写法

7. var result = values.sort((a, b) => a - b);

8.

下面是rest参数与箭头函数结合的例子。

1. const numbers = (...nums) => nums;

2.

3. numbers(1, 2, 3, 4, 5);

4. // [1,2, 3, 4, 5]

5.

6. const headAndTail = (head, ...tail) => [head, tail]

7.

8. headAndTail(1, 2, 3, 4, 5)

9. // [1, [2, 3, 4, 5]]

10.

**使用注意点**

箭头函数有几个使用注意点。

1. 函数体内的this对象就是定义时所在的对象，而不是使用时所在的对象。

2. 不可以当作构造函数。也就是说，不可以使用new命令，否则会抛出一个错误。

3. 不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用rest参数代替。

4. 不可以使用yield命令，因此箭头函数不能用作Generator函数。

其中，第一点尤其值得注意。this对象的指向是可变的，但在箭头函数中它是固定的。

1. function foo() {

2.     setTimeout(() => {

3.         console.log("id", this.id);

4.     }, 100);

5. }

6.

7. foo.call({ id: 42 });

8. // id: 42

9.

上面的代码中，setTimeout的参数是一个箭头函数，100毫秒后执行。如果是普通函数，执行时this对象应该指向全局对象，但是箭头函数导致this对象总是指向函数所在的对象。

下面是另一个例子。

1. var handler = {

2.     id: "123456",

3.

4.     init: function () {

5.         document.addEventListener("click",

6.             event => **this**.doSomething(event.type), false);

7.     },

8.     doSomething: function (type) {

9.         console.log("Handling " + type + " for " + this.id);

10.     }

11. };

12.

上面的init方法中使用了箭头函数，这导致this总是指向handler对象。否则，回调函数运行时，this.doSomething这一行会报错，因为此时this指向全局对象。

1. function Timer() {

2.     this.seconds = 0;

3.     setInterval(() => this.seconds++, 1000);

4. }

5.

6. var timer = new Timer();

7. setTimeout(() => console.log(timer.seconds), 3100);

8.

9. // 3

10.

上面的代码中，Timer函数内部的setInterval调用了this.seconds属性，通过箭头函数让this总是指向Timer的实例对象。否则，输出结果是0，而不是3。

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this对象，导致内部的this就是外层代码块的this。正因为它没有this，所以也就不能用作构造函数。

请问，下面的代码中有几个this？

1. function foo() {

2.     return () => {

3.         return () => {

4.             return () => {

5.                 console.log("id:", this.id);

6.             };

7.         };

8.     };

9. }

10.

11. foo.call({ id: 42 })()()();

12. // 42

13.

上面的代码中只有一个this，就是函数foo的this。因为所有的内层函数都是箭头函数，都没有自己的this，所以它们的this其实都是最外层foo函数的this。

除了this，以下3个变量在箭头函数中也是不存在的，分别指向外层函数的对应变量：**arguments**、**super**和**new.target**。

1. function foo() {

2.     setTimeout(() => {

3.         console.log("args:", arguments);

4.     }, 100);

5. }

6.

7. foo(2, 3, 4, 6, 8);

8. // args: [2, 3, 4, 6, 8]

9.

上面的代码中，箭头函数内部的变量arguments，其实是函数foo的arguments变量。

另外，由于箭头函数没有自己的this，当然也就不能用call()、apply()、bind()这些方法去改变this的指向。

1. (function () {

2.     return [

3.         (() => { console.log(this.x); }).bind({ x: "inner" })

4.     ];

5. }).call({ x: "outer"})[0]();

6. // outer

上面的代码中，箭头函数没有自己的this，所以bind方法无效，内部的this指向外部的this。

长期以来，Javascript语言的this对象一直是一个令人头疼的问题，在对象方法中使用this必须非常小心。箭头函数“绑定“this，很大程度上解决了这个困扰。

**嵌套的箭头函数**

箭头函数内部还可以再使用箭头函数。下面是一个ES5语法的多重嵌套函数。

1. function insert(value) {

2.     return {

3.         into: function (array) {

4.             return {

5.                 after: function (afterValue) {

6.                     array.splice(array.indexOf(afterValue) + 1, 0, value);

7.                     return array;

8.                 }

9.             };

10.         }

11.     };

12. }

13.

1. console.log(insert(2).into([1, 3]).after(1));;

2. // [ 1, 2, 3 ]

上面这个函数可以使用箭头函数改写如下。

1. function insert(value) {

2.     return {

3.         into: (array) => {

4.             return {

5.                 after: (afterValue) => {

6.                     array.splice(array.indexOf(afterValue) + 1, 0, value);

7.                     return array;

8.                 }

9.             };

10.         }

11.     };

12. }

13.

下面是一个部署管道机制（pipeline）的例子，即前一个函数的输出是后一个函数的输入。

1. const pipeline = (...funcs) => {

2.     return val => funcs.reduce((a, b) => b(a), val);

3. };

4.

5. const plus1 = a => a + 1;

6. const mult2 = a => a \* 2;

7. const addThenMult = pipeline(plus1, mult2);

8.

9. console.log(addThenMult(5));

10.

如果觉得上面的写法可读性比较差，也可以采用下面的写法。

1. const plus1 = a => a + 1;

2. const mult2 = a => a \* 2;

3.

4. mult2(plus1(5));

5. // 12

6.

箭头函数还有一个功能，就是可以很方便地改写演算。

1. // λ演算的写法

2. fix = λf.(λx.f(λv.x(x)(v)))(λx.f(λv.x(x)(v)));

3.

4. // ES6的写法

5. var fix = f => x => f(v => x(x)(v));

6.     x => f(v => x(x)(v));

7.

上面的两种写法几乎是一一对应的。由于λ演算对于计算机科学非常重要，这使得我们可以用ES6作为替代工具，探索计算机科学。

**8.6 函数绑定**

箭头函数可以绑定this对象，大大减少了显示绑定this对象的写法（call、apply、bind）。但是，箭头函数并非适用于所有场合，所以ES7提出了“函数绑定”（function bind）运算符，用来取代call、apply、bind调用。虽然该语法还是ES7的一个提案（http://github.com/zenparsing/es-function-bind）,但是Babel转码器已经支持。

函数绑定运算符是并排的双冒号（::），双冒号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象作为上下文环境（即this对象）绑定到右边的函数上。

1. foo::bar;

2. // 等同于

3. bar.bind(foo);

4.

5. foo::bar(...arguments);

6. // 等同于

7. bar.apply(foo, arguments);

8.

9. const hasOwnProperty = Object.prototype.hasOwnProperty;

10. function hasOwn(obj, key) {

11.     return obj::hasOwnProperty(key);

12. }

13.

如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上。

1. var method = obj::obj.foo;

2. // 等同于

3. var method = ::obj.foo;

4.

5. let log = ::console.log;

6. // 等同于

7. var log = console.log.bind(console);

8.

由于双冒号运算符返回的还是元对象，因此可以采用链式写法。

1. // 例1

2. import {map, takeWhile, forEach} from "iterlib";

3.

4. getPlayers()

5. ::map(x=>x.character())

6. ::takeWhile(x => x.strength > 100)

7. ::forEach(x => console.log(x));

8.

9. var jake = {

10.     find: "xxx",

11.     html: "jake-html"

12. }

13. // 例2

14. let {find, html} = jake;

15.

16. document.querySelectorAll("div.myClass")

17. ::find("p")

18. ::html("hahaha");

19.

**8.7 尾调用优化**

**什么是尾调用**

尾调用（Tail Call）是函数式编程的一个重要概念，本身非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。

1. function f(x) {

2.     return g(x);

3. }

4.

上面的代码中，函数f的最后一步是调用函数g，这就叫尾调用。

以下情况都不属于尾调用。

1. // 情况1

2. function f(x) {

3.     let y = g(x);

4.     return y;

5. }

6.

7. // 情况2

8. function f(x) {

9.     return g(x) + 1;

10. }

11.

12. // 情况3

13. function f(x) {

14.     g(x);

15. }

16.

上面的代码中，情况1是调用g之后还有赋值操作，所以不属于尾调用，即使语义完全一样；情况2也属于调用后还有操作，即使写在一行内；情况3等同于下面的代码。

1. function f(x) {

2.     g(x);

3.     return undefined;

4. }

5.

尾调用不一定出现在函数尾部，只要是最后一步操作即可。

1. function f(x) {

2.     if (x > 0) {

3.         return m(x);

4.     }

5.     return n(x);

6. }

7.

上面的代码中，函数m和n都属于尾调用，因为它们都是函数f的最后一步操作。

**尾调用优化**

尾调用之所以与其他调用不同，就在于其特殊的调用位置。

我们知道，函数调用会在内存形成一个“调用记录“，又称为”调用帧“（call frame），保存调用位置和内部变量等信息。如果在函数A内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用了函数C，那就还有一个C的调用帧，以此类推。所有的调用帧就形成了一个”调用栈“（call stack）。

尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置，内部变量的信息都不会用到了，直接用内层函数调用帧取代外层函数调用帧即可。

1. function f() {

2.     let m = 1;

3.     let n = 2;

4.     return g(n + m);

5. }

6. f();

7.

8. // 等同于

9. function f() {

10.     return g(3);

11. }

12. f();

13.

14. // 等同于

15. g(3);

16.

上面的代码中，如果函数g不是尾调用，函数f就需要保存内部变量n和m、g的调用位置等信息。但由于调用g之后，函数f就结束了，所以执行到最后一步，完全可以删除f(x)的调用帧，只保留g(3)的调用帧。

这就叫做“尾调用优化”（tail Call Optimization），即只保留内部函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

注意，只有不在用到外层函数的内部变量，内层函数的调用帧才会取代外层函数调用帧，否则就无法进行“尾调用优化”。

1. function addOne(a) {

2.     var one = 1;

3.     function inner(b) {

4.         return b + **one**;

5.     }

6.     return inner(a);

7. }

8.

上面的函数不会进行“尾调用的优化”，因为内层函数inner用到了外层函数addOne的内部变量one。

**尾递归**

函数调用自身称为递归。如果尾调用自身就称为尾递归。

递归非常消耗内存，因为需要同时保存成百上千个调用帧，所以很容易发生“栈溢出”（stack overflow）。但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。

1. function factorial(n) {

2.     if (n === 1) {

3.         return n;

4.     }

5.     return n \* factorial(n - 1);

6. }

7. factorial(5); // 120

上面的代码是一个阶乘函数，计算n的阶乘，最多需要保存n个调用记录，复杂度为O(n)。如果改写为尾递归，只保留一个调用记录，则复杂度为O(1)。

1. function factorial(n, total) {

2.     "use strict";

3.     if (n === 1) return total;

4.     return factorial(n - 1, n \* total);

5. }

6. factorial(5, 1); // 120

由此可见，“尾调用优化“对递归操作的意义重大，所以一些函数式编程语言将其写入了语言规格。ES6也是如此，第一次明确规定，所有ECMAScript的实现，都必须部署”尾调用优化”。这就是说，在ES6中，只要使用了尾递归，就不会发生栈溢出，相对节省内存。

注意，只有开启严格模式，尾调用优化才会生效。一旦启用了尾调用优化，func.arguments和func.caller这两个函数内部对象就失去了意义，因为外层的调用帧会被整个替换掉，这两个对象包含的信息会被移除。严格模式下，这两个对象也是不可用的。

1. function restricted() {

2.     "use strict";

3.     restricted.caller;

4.     restricted.arguments;

5. }

6. restricted();

**递归函数的改写**

尾递归的实现往往需要改写递归函数，确保最后一步只调用自身。做到这一点的方法，就是把所有用到的内部变量改写成函数的参数。比如上面这个例子，阶乘函数factorial需要用到一个中间变量total，那就把这个中间变量改写成函数的参数，这样做的缺点是不太直观，第一眼很难看出来，为什么计算5的阶乘需要传入两个参数5和1？

有两个方法可以解决这个问题。方法一是在尾递归函数之外再提供一个正常形式的函数。

1. function factorial(n, total) {

2.     "use strict";

3.     if (n === 1) return total;

4.     return factorial(n - 1, n \* total);

5. }

6.

7. function factorial(n) {

8.     return factorial(n, 1);

9. }

10. factorial(5); // 120

上面的代码通过一个正常形式的阶乘函数factorial调用尾递归函数tailFactorial，看起来就正常多了。

函数编程有一个概念，叫做柯里化（currying），意思是将多参数函数转换成单参数的形式。

这里也可以使用柯里化。

1. function currying(fn, n) {

2.     return function (m) {

3.         return fn.call(this, m, n);

4.     }

5. }

6.

7. function tailFunction(n, total) {

8.     if (n === 1) {

9.         return 1;

10.     }

11.     return tailFunction(n - 1, n \* total);

12. }

13.

14. const factorial = currying(tailFunction, n);

15.

16. factorial(5); //

17.

18.

上面的代码通过柯里化将尾递归函数tailFactorial变为只接受一个参数的factorial。

第二种方法就简单多了，那就是采用ES6的函数默认值。

1. function factorial(n, total = 1) {

2.     if (n === 1) {

3.         return total;

4.     }

5.     return factorial(n - 1, n \* total);

6. }

7.

8. factorial(5); // 120

9.

上面的代码中，参数total有默认值1， 所以调用时不用提供这个值。

总结一下，递归本质上是一种循环操作。纯粹的函数式编程语言没有循环操作命令，所有的循环都用递归实现，这就是为什么尾递归对这些语言极其重要。对于其他支持“尾调用优化”的语言（比如Lua，ES6），只需要知道循环可以用递归代替，而一旦使用递归，就最好使用尾递归。

**8.8 函数参数的尾逗号**

ES7有一个[提案](https://github.com/tc39/proposal-trailing-function-commas)，允许函数最后一个参数有尾逗号（trailing comma）。

目前，函数定义和调用时都不允许参数有尾逗号。

1. function clownEverywhere(param1, param2) {

2.     /\*\*

3.      \*

4.      \*/

5. }

6.

7. clownEverywhere("foo", "bar");

8.

如果以后要在函数的定义中添加参数，就势必要添加一个逗号。对版本管理系统来说，就会显示添加逗号的那一行也发生了变动。这看上去有点冗余，因此新提案允许定义和调用时尾部有一个逗号。

1. function clownEverywhere(param1, param2,) {

2.     /\*\*

3.      \*

4.      \*/

5. }

6.

7. clownEverywhere("foo", "bar",);

8.