另外，还可以把集合绑定到一个事件分派程序，每次插入新实例时都会发送消息：

```
const userList = [];

function emit(newValue) {
  console.log(newValue);
}

const proxy = new Proxy(userList, {
  set(target, property, value, receiver) {
    const result = Reflect.set(...arguments);
    if (result) {
      emit(Reflect.get(target, property, receiver));
    }
    return result;
  }
});

proxy.push('John');
// John
proxy.push('Jacob');
// Jacob
```

9.4　小结

代理是 ECMAScript 6 新增的令人兴奋和动态十足的新特性。尽管不支持向后兼容，但它开辟出了一片前所未有的 JavaScript 元编程及抽象的新天地。

从宏观上看，代理是真实 JavaScript 对象的透明抽象层。代理可以定义包含**捕获器**的处理程序对象，而这些捕获器可以拦截绝大部分 JavaScript 的基本操作和方法。在这个捕获器处理程序中，可以修改任何基本操作的行为，当然前提是遵从**捕获器不变式**。

与代理如影随形的反射 API，则封装了一整套与捕获器拦截的操作相对应的方法。可以把反射 API 看作一套基本操作，这些操作是绝大部分 JavaScript 对象 API 的基础。

代理的应用场景是不可限量的。开发者使用它可以创建出各种编码模式，比如（但远远不限于）跟踪属性访问、隐藏属性、阻止修改或删除属性、函数参数验证、构造函数参数验证、数据绑定，以及可观察对象。

第10章 函　数

本章内容

- ❑ 函数表达式、函数声明及箭头函数
- ❑ 默认参数及扩展操作符
- ❑ 使用函数实现递归
- ❑ 使用闭包实现私有变量

函数是ECMAScript中最有意思的部分之一，这主要是因为函数实际上是对象。每个函数都是Function类型的实例，而Function也有属性和方法，跟其他引用类型一样。因为函数是对象，所以函数名就是指向函数对象的指针，而且不一定与函数本身紧密绑定。函数通常以函数声明的方式定义，比如：

```
function sum (num1, num2) {
  return num1 + num2;
}
```

注意函数定义最后没有加分号。

另一种定义函数的语法是函数表达式。函数表达式与函数声明几乎是等价的：

```
let sum = function(num1, num2) {
  return num1 + num2;
};
```

这里，代码定义了一个变量sum并将其初始化为一个函数。注意function关键字后面没有名称，因为不需要。这个函数可以通过变量sum来引用。

注意这里的函数末尾是有分号的，与任何变量初始化语句一样。

还有一种定义函数的方式与函数表达式很像，叫作“箭头函数”（arrow function），如下所示：

```
let sum = (num1, num2) => {
  return num1 + num2;
};
```

最后一种定义函数的方式是使用Function构造函数。这个构造函数接收任意多个字符串参数，最后一个参数始终会被当成函数体，而之前的参数都是新函数的参数。来看下面的例子：

```
let sum = new Function("num1", "num2", "return num1 + num2");  // 不推荐
```

我们不推荐使用这种语法来定义函数，因为这段代码会被解释两次：第一次是将它当作常规ECMAScript代码，第二次是解释传给构造函数的字符串。这显然会影响性能。不过，把函数想象为对象，把函数名想象为指针是很重要的。而上面这种语法很好地诠释了这些概念。

> **注意**　这几种实例化函数对象的方式之间存在微妙但重要的差别，本章后面会讨论。无论如何，通过其中任何一种方式都可以创建函数。

10.1　箭头函数

ECMAScript 6 新增了使用胖箭头（=>）语法定义函数表达式的能力。很大程度上，箭头函数实例化的函数对象与正式的函数表达式创建的函数对象行为是相同的。任何可以使用函数表达式的地方，都可以使用箭头函数：

```
let arrowSum = (a, b) => {
  return a + b;
};

let functionExpressionSum = function(a, b) {
  return a + b;
};

console.log(arrowSum(5, 8)); // 13
console.log(functionExpressionSum(5, 8)); // 13
```

箭头函数简洁的语法非常适合嵌入函数的场景：

```
let ints = [1, 2, 3];

console.log(ints.map(function(i) { return i + 1; }));  // [2, 3, 4]
console.log(ints.map((i) => { return i + 1 }));        // [2, 3, 4]
```

如果只有一个参数，那也可以不用括号。只有没有参数，或者多个参数的情况下，才需要使用括号：

```
// 以下两种写法都有效
let double = (x) => { return 2 * x; };
let triple = x => { return 3 * x; };

// 没有参数需要括号
let getRandom = () => { return Math.random(); };

// 多个参数需要括号
let sum = (a, b) => { return a + b; };

// 无效的写法：
let multiply = a, b => { return a * b; };
```

箭头函数也可以不用大括号，但这样会改变函数的行为。使用大括号就说明包含“函数体”，可以在一个函数中包含多条语句，跟常规的函数一样。如果不使用大括号，那么箭头后面就只能有一行代码，比如一个赋值操作，或者一个表达式。而且，省略大括号会隐式返回这行代码的值：

```
// 以下两种写法都有效，而且返回相应的值
let double = (x) => { return 2 * x; };
let triple = (x) => 3 * x;

// 可以赋值
let value = {};
let setName = (x) => x.name = "Matt";
setName(value);
console.log(value.name); // "Matt"

// 无效的写法：
let multiply = (a, b) => return a * b;
```

箭头函数虽然语法简洁，但也有很多场合不适用。箭头函数不能使用 `arguments`、`super` 和 `new.target`，也不能用作构造函数。此外，箭头函数也没有 `prototype` 属性。

10.2 函数名

因为函数名就是指向函数的指针，所以它们跟其他包含对象指针的变量具有相同的行为。这意味着一个函数可以有多个名称，如下所示：

```
function sum(num1, num2) {
  return num1 + num2;
}

console.log(sum(10, 10));          // 20

let anotherSum = sum;
console.log(anotherSum(10, 10));  // 20

sum = null;
console.log(anotherSum(10, 10));  // 20
```

以上代码定义了一个名为 sum()的函数，用于求两个数之和。然后又声明了一个变量 anotherSum，并将它的值设置为等于 sum。注意，使用不带括号的函数名会访问函数指针，而不会执行函数。此时，anotherSum 和 sum 都指向同一个函数。调用 anotherSum()也可以返回结果。把 sum 设置为 null 之后，就切断了它与函数之间的关联。而 anotherSum()还是可以照常调用，没有问题。

ECMAScript 6 的所有函数对象都会暴露一个只读的 name 属性，其中包含关于函数的信息。多数情况下，这个属性中保存的就是一个函数标识符，或者说是一个字符串化的变量名。即使函数没有名称，也会如实显示成空字符串。如果它是使用 Function 构造函数创建的，则会标识成"anonymous"：

```
function foo() {}
let bar = function() {};
let baz = () => {};

console.log(foo.name);               // foo
console.log(bar.name);               // bar
console.log(baz.name);               // baz
console.log((() => {}).name);        // (空字符串)
console.log((new Function()).name);  // anonymous
```

如果函数是一个获取函数、设置函数，或者使用 bind()实例化，那么标识符前面会加上一个前缀：

```
function foo() {}

console.log(foo.bind(null).name);    // bound foo

let dog = {
  years: 1,
  get age() {
    return this.years;
  },
  set age(newAge) {
    this.years = newAge;
  }
}

let propertyDescriptor = Object.getOwnPropertyDescriptor(dog, 'age');
console.log(propertyDescriptor.get.name);  // get age
console.log(propertyDescriptor.set.name);  // set age
```

10.3　理解参数

ECMAScript 函数的参数跟大多数其他语言不同。ECMAScript 函数既不关心传入的参数个数，也不关心这些参数的数据类型。定义函数时要接收两个参数，并不意味着调用时就传两个参数。你可以传一个、三个，甚至一个也不传，解释器都不会报错。

之所以会这样，主要是因为 ECMAScript 函数的参数在内部表现为一个数组。函数被调用时总会接收一个数组，但函数并不关心这个数组中包含什么。如果数组中什么也没有，那没问题；如果数组的元素超出了要求，那也没问题。事实上，在使用 `function` 关键字定义（非箭头）函数时，可以在函数内部访问 `arguments` 对象，从中取得传进来的每个参数值。

`arguments` 对象是一个类数组对象（但不是 `Array` 的实例），因此可以使用中括号语法访问其中的元素（第一个参数是 `arguments[0]`，第二个参数是 `arguments[1]`）。而要确定传进来多少个参数，可以访问 `arguments.length` 属性。

在下面的例子中，`sayHi()`函数的第一个参数叫 `name`：

```
function sayHi(name, message) {
  console.log("Hello " + name + ", " + message);
}
```

可以通过 `arguments[0]`取得相同的参数值。因此，把函数重写成不声明参数也可以：

```
function sayHi() {
  console.log("Hello " + arguments[0] + ", " + arguments[1]);
}
```

在重写后的代码中，没有命名参数。`name` 和 `message` 参数都不见了，但函数照样可以调用。这就表明，ECMAScript 函数的参数只是为了方便才写出来的，并不是必须写出来的。与其他语言不同，在 ECMAScript 中的命名参数不会创建让之后的调用必须匹配的函数签名。这是因为根本不存在验证命名参数的机制。

也可以通过 `arguments` 对象的 `length` 属性检查传入的参数个数。下面的例子展示了在每调用一个函数时，都会打印出传入的参数个数：

```
function howManyArgs() {
  console.log(arguments.length);
}

howManyArgs("string", 45);  // 2
howManyArgs();              // 0
howManyArgs(12);            // 1
```

这个例子分别打印出 2、0 和 1（按顺序）。既然如此，那么开发者可以想传多少参数就传多少参数。比如：

```
function doAdd() {
  if (arguments.length === 1) {
    console.log(arguments[0] + 10);
  } else if (arguments.length === 2) {
    console.log(arguments[0] + arguments[1]);
  }
}

doAdd(10);      // 20
doAdd(30, 20);  // 50
```

这个函数 doAdd()在只传一个参数时会加 10，在传两个参数时会将它们相加，然后返回。因此 doAdd(10)返回 20，而 doAdd(30,20)返回 50。虽然不像真正的函数重载那么明确，但这已经足以弥补 ECMAScript 在这方面的缺失了。

还有一个必须理解的重要方面，那就是 arguments 对象可以跟命名参数一起使用，比如：

```
function doAdd(num1, num2) {
  if (arguments.length === 1) {
    console.log(num1 + 10);
  } else if (arguments.length === 2) {
    console.log(arguments[0] + num2);
  }
}
```

在这个 doAdd()函数中，同时使用了两个命名参数和 arguments 对象。命名参数 num1 保存着与 arugments[0]一样的值，因此使用谁都无所谓。（同样，num2 也保存着跟 arguments[1]一样的值。）

arguments 对象的另一个有意思的地方就是，它的值始终会与对应的命名参数同步。来看下面的例子：

```
function doAdd(num1, num2) {
  arguments[1] = 10;
  console.log(arguments[0] + num2);
}
```

这个 doAdd()函数把第二个参数的值重写为 10。因为 arguments 对象的值会自动同步到对应的命名参数，所以修改 arguments[1]也会修改 num2 的值，因此两者的值都是 10。但这并不意味着它们都访问同一个内存地址，它们在内存中还是分开的，只不过会保持同步而已。另外还要记住一点：如果只传了一个参数，然后把 arguments[1]设置为某个值，那么这个值并不会反映到第二个命名参数。这是因为 arguments 对象的长度是根据传入的参数个数，而非定义函数时给出的命名参数个数确定的。

对于命名参数而言，如果调用函数时没有传这个参数，那么它的值就是 undefined。这就类似于定义了变量而没有初始化。比如，如果只给 doAdd()传了一个参数，那么 num2 的值就是 undefined。

严格模式下，arguments 会有一些变化。首先，像前面那样给 arguments[1]赋值不会再影响 num2 的值。就算把 arguments[1]设置为 10，num2 的值仍然还是传入的值。其次，在函数中尝试重写 arguments 对象会导致语法错误。（代码也不会执行。）

箭头函数中的参数

如果函数是使用箭头语法定义的，那么传给函数的参数将不能使用 arguments 关键字访问，而只能通过定义的命名参数访问。

```
function foo() {
  console.log(arguments[0]);
}
foo(5); // 5

let bar = () => {
  console.log(arguments[0]);
};
bar(5);  // ReferenceError: arguments is not defined
```

虽然箭头函数中没有 arguments 对象，但可以在包装函数中把它提供给箭头函数：

```
function foo() {
  let bar = () => {
    console.log(arguments[0]); // 5
  };
  bar();
}

foo(5);
```

> **注意**　ECMAScript 中的所有参数都按值传递的。不可能按引用传递参数。如果把对象作为参数传递，那么传递的值就是这个对象的引用。

10.4　没有重载

ECMAScript 函数不能像传统编程那样重载。在其他语言比如 Java 中，一个函数可以有两个定义，只要签名（接收参数的类型和数量）不同就行。如前所述，ECMAScript 函数没有签名，因为参数是由包含零个或多个值的数组表示的。没有函数签名，自然也就没有重载。

如果在 ECMAScript 中定义了两个同名函数，则后定义的会覆盖先定义的。来看下面的例子：

```
function addSomeNumber(num) {
  return num + 100;
}

function addSomeNumber(num) {
  return num + 200;
}

let result = addSomeNumber(100); // 300
```

这里，函数 addSomeNumber()被定义了两次。第一个版本给参数加 100，第二个版本加 200。最后一行调用这个函数时，返回了 300，因为第二个定义覆盖了第一个定义。

前面也提到过，可以通过检查参数的类型和数量，然后分别执行不同的逻辑来模拟函数重载。

把函数名当成指针也有助于理解为什么 ECMAScript 没有函数重载。在前面的例子中，定义两个同名的函数显然会导致后定义的重写先定义的。而那个例子几乎跟下面这个是一样的：

```
let addSomeNumber = function(num) {
    return num + 100;
};

addSomeNumber = function(num) {
    return num + 200;
};

let result = addSomeNumber(100); // 300
```

看这段代码应该更容易理解发生了什么。在创建第二个函数时，变量 addSomeNumber 被重写成保存第二个函数对象了。

10.5 默认参数值

在 ECMAScript5.1 及以前，实现默认参数的一种常用方式就是检测某个参数是否等于 undefined，如果是则意味着没有传这个参数，那就给它赋一个值：

```
function makeKing(name) {
  name = (typeof name !== 'undefined') ? name : 'Henry';
  return `King ${name} VIII`;
}

console.log(makeKing());         // 'King Henry VIII'
console.log(makeKing('Louis'));  // 'King Louis VIII'
```

ECMAScript 6 之后就不用这么麻烦了，因为它支持显式定义默认参数了。下面就是与前面代码等价的 ES6 写法，只要在函数定义中的参数后面用=就可以为参数赋一个默认值：

```
function makeKing(name = 'Henry') {
  return `King ${name} VIII`;
}

console.log(makeKing('Louis'));  // 'King Louis VIII'
console.log(makeKing());         // 'King Henry VIII'
```

给参数传 undefined 相当于没有传值，不过这样可以利用多个独立的默认值：

```
function makeKing(name = 'Henry', numerals = 'VIII') {
  return `King ${name} ${numerals}`;
}

console.log(makeKing());                 // 'King Henry VIII'
console.log(makeKing('Louis'));          // 'King Louis VIII'
console.log(makeKing(undefined, 'VI'));  // 'King Henry VI'
```

在使用默认参数时，arguments 对象的值不反映参数的默认值，只反映传给函数的参数。当然，跟 ES5 严格模式一样，修改命名参数也不会影响 arguments 对象，它始终以调用函数时传入的值为准：

```
function makeKing(name = 'Henry') {
  name = 'Louis';
  return `King ${arguments[0]}`;
}

console.log(makeKing());         // 'King undefined'
console.log(makeKing('Louis'));  // 'King Louis'
```

默认参数值并不限于原始值或对象类型，也可以使用调用函数返回的值：

```
let romanNumerals = ['I', 'II', 'III', 'IV', 'V', 'VI'];
let ordinality = 0;

function getNumerals() {
  // 每次调用后递增
  return romanNumerals[ordinality++];
}

function makeKing(name = 'Henry', numerals = getNumerals()) {
  return `King ${name} ${numerals}`;
}

console.log(makeKing());                 // 'King Henry I'
```

```
console.log(makeKing('Louis', 'XVI'));  // 'King Louis XVI'
console.log(makeKing());                // 'King Henry II'
console.log(makeKing());                // 'King Henry III'
```

函数的默认参数只有在函数被调用时才会求值，不会在函数定义时求值。而且，计算默认值的函数只有在调用函数但未传相应参数时才会被调用。

箭头函数同样也可以这样使用默认参数，只不过在只有一个参数时，就必须使用括号而不能省略了：

```
let makeKing = (name = 'Henry') => `King ${name}`;

console.log(makeKing()); // King Henry
```

默认参数作用域与暂时性死区

因为在求值默认参数时可以定义对象，也可以动态调用函数，所以函数参数肯定是在某个作用域中求值的。

给多个参数定义默认值实际上跟使用 `let` 关键字顺序声明变量一样。来看下面的例子：

```
function makeKing(name = 'Henry', numerals = 'VIII') {
  return `King ${name} ${numerals}`;
}

console.log(makeKing()); // King Henry VIII
```

这里的默认参数会按照定义它们的顺序依次被初始化。可以依照如下示例想象一下这个过程：

```
function makeKing() {
  let name = 'Henry';
  let numerals = 'VIII';

  return `King ${name} ${numerals}`;
}
```

因为参数是按顺序初始化的，所以后定义默认值的参数可以引用先定义的参数。看下面这个例子：

```
function makeKing(name = 'Henry', numerals = name) {
  return `King ${name} ${numerals}`;
}

console.log(makeKing()); // King Henry Henry
```

参数初始化顺序遵循“暂时性死区”规则，即前面定义的参数不能引用后面定义的。像这样就会抛出错误：

```
// 调用时不传第一个参数会报错
function makeKing(name = numerals, numerals = 'VIII') {
  return `King ${name} ${numerals}`;
}
```

参数也存在于自己的作用域中，它们不能引用函数体的作用域：

```
// 调用时不传第二个参数会报错
function makeKing(name = 'Henry', numerals = defaultNumeral) {
  let defaultNumeral = 'VIII';
  return `King ${name} ${numerals}`;
}
```

10.6 参数扩展与收集

ECMAScript 6 新增了扩展操作符，使用它可以非常简洁地操作和组合集合数据。扩展操作符最有用的场景就是函数定义中的参数列表，在这里它可以充分利用这门语言的弱类型及参数长度可变的特点。扩展操作符既可以用于调用函数时传参，也可以用于定义函数参数。

10.6.1 扩展参数

在给函数传参时，有时候可能不需要传一个数组，而是要分别传入数组的元素。

假设有如下函数定义，它会将所有传入的参数累加起来：

```
let values = [1, 2, 3, 4];

function getSum() {
  let sum = 0;
  for (let i = 0; i < arguments.length; ++i) {
    sum += arguments[i];
  }
  return sum;
}
```

这个函数希望将所有加数逐个传进来，然后通过迭代 `arguments` 对象来实现累加。如果不使用扩展操作符，想把定义在这个函数这面的数组拆分，那么就得求助于 `apply()`方法：

```
console.log(getSum.apply(null, values)); // 10
```

但在 ECMAScript 6 中，可以通过扩展操作符极为简洁地实现这种操作。对可迭代对象应用扩展操作符，并将其作为一个参数传入，可以将可迭代对象拆分，并将迭代返回的每个值单独传入。

比如，使用扩展操作符可以将前面例子中的数组像这样直接传给函数：

```
console.log(getSum(...values)); // 10
```

因为数组的长度已知，所以在使用扩展操作符传参的时候，并不妨碍在其前面或后面再传其他的值，包括使用扩展操作符传其他参数：

```
console.log(getSum(-1, ...values));          // 9
console.log(getSum(...values, 5));           // 15
console.log(getSum(-1, ...values, 5));       // 14
console.log(getSum(...values, ...[5,6,7]));  // 28
```

对函数中的 `arguments` 对象而言，它并不知道扩展操作符的存在，而是按照调用函数时传入的参数接收每一个值：

```
let values = [1,2,3,4]

function countArguments() {
  console.log(arguments.length);
}

countArguments(-1, ...values);          // 5
countArguments(...values, 5);           // 5
countArguments(-1, ...values, 5);       // 6
countArguments(...values, ...[5,6,7]);  // 7
```

`arguments` 对象只是消费扩展操作符的一种方式。在普通函数和箭头函数中，也可以将扩展操作