符用于命名参数，当然同时也可以使用默认参数：

```
function getProduct(a, b, c = 1) {
  return a * b * c;
}

let getSum = (a, b, c = 0) => {
  return a + b + c;
}

console.log(getProduct(...[1,2]));      // 2
console.log(getProduct(...[1,2,3]));    // 6
console.log(getProduct(...[1,2,3,4]));  // 6

console.log(getSum(...[0,1]));          // 1
console.log(getSum(...[0,1,2]));        // 3
console.log(getSum(...[0,1,2,3]));      // 3
```

10.6.2　收集参数

在构思函数定义时，可以使用扩展操作符把不同长度的独立参数组合为一个数组。这有点类似 arguments 对象的构造机制，只不过收集参数的结果会得到一个 Array 实例。

```
function getSum(...values) {
  // 顺序累加 values 中的所有值
  // 初始值的总和为 0
  return values.reduce((x, y) => x + y, 0);
}

console.log(getSum(1,2,3)); // 6
```

收集参数的前面如果还有命名参数，则只会收集其余的参数；如果没有则会得到空数组。因为收集参数的结果可变，所以只能把它作为最后一个参数：

```
// 不可以
function getProduct(...values, lastValue) {}

// 可以
function ignoreFirst(firstValue, ...values) {
  console.log(values);
}

ignoreFirst();       // []
ignoreFirst(1);      // []
ignoreFirst(1,2);    // [2]
ignoreFirst(1,2,3);  // [2, 3]
```

箭头函数虽然不支持 arguments 对象，但支持收集参数的定义方式，因此也可以实现与使用 arguments 一样的逻辑：

```
let getSum = (...values) => {
  return values.reduce((x, y) => x + y, 0);
}

console.log(getSum(1,2,3)); // 6
```

另外，使用收集参数并不影响 arguments 对象，它仍然反映调用时传给函数的参数：

```
function getSum(...values) {
  console.log(arguments.length);  // 3
  console.log(arguments);         // [1, 2, 3]
  console.log(values);            // [1, 2, 3]
}

console.log(getSum(1,2,3));
```

10.7 函数声明与函数表达式

本章到现在一直没有把函数声明和函数表达式区分得很清楚。事实上，JavaScript 引擎在加载数据时对它们是区别对待的。JavaScript 引擎在任何代码执行之前，会先读取函数声明，并在执行上下文中生成函数定义。而函数表达式必须等到代码执行到它那一行，才会在执行上下文中生成函数定义。来看下面的例子：

```
// 没问题
console.log(sum(10, 10));
function sum(num1, num2) {
  return num1 + num2;
}
```

以上代码可以正常运行，因为函数声明会在任何代码执行之前先被读取并添加到执行上下文。这个过程叫作**函数声明提升**（function declaration hoisting）。在执行代码时，JavaScript 引擎会先执行一遍扫描，把发现的函数声明提升到源代码树的顶部。因此即使函数定义出现在调用它们的代码之后，引擎也会把函数声明**提升**到顶部。如果把前面代码中的函数声明改为等价的函数表达式，那么执行的时候就会出错：

```
// 会出错
console.log(sum(10, 10));
let sum = function(num1, num2) {
  return num1 + num2;
};
```

上面的代码之所以会出错，是因为这个函数定义包含在一个变量初始化语句中，而不是函数声明中。这意味着代码如果没有执行到加粗的那一行，那么执行上下文中就没有函数的定义，所以上面的代码会出错。这并不是因为使用 `let` 而导致的，使用 `var` 关键字也会碰到同样的问题：

```
console.log(sum(10, 10));
var sum = function(num1, num2) {
  return num1 + num2;
};
```

除了函数什么时候真正有定义这个区别之外，这两种语法是等价的。

> **注意** 在使用函数表达式初始化变量时，也可以给函数一个名称，比如 `let sum = function sum() {}`。这一点在 10.11 节讨论函数表达式时会再讨论。

10.8 函数作为值

因为函数名在 ECMAScript 中就是变量，所以函数可以用在任何可以使用变量的地方。这意味着不仅可以把函数作为参数传给另一个函数，而且还可以在一个函数中返回另一个函数。来看下面的例子：

```
function callSomeFunction(someFunction, someArgument) {
  return someFunction(someArgument);
}
```

这个函数接收两个参数。第一个参数应该是一个函数，第二个参数应该是要传给这个函数的值。任何函数都可以像下面这样作为参数传递：

```
function add10(num) {
  return num + 10;
}

let result1 = callSomeFunction(add10, 10);
console.log(result1);  // 20

function getGreeting(name) {
  return "Hello, " + name;
}

let result2 = callSomeFunction(getGreeting, "Nicholas");
console.log(result2);  // "Hello, Nicholas"
```

`callSomeFunction()`函数是通用的，第一个参数传入的是什么函数都可以，而且它始终返回调用作为第一个参数传入的函数的结果。要注意的是，如果是访问函数而不是调用函数，那就必须不带括号，所以传给 `callSomeFunction()`的必须是 `add10` 和 `getGreeting`，而不能是它们的执行结果。

从一个函数中返回另一个函数也是可以的，而且非常有用。例如，假设有一个包含对象的数组，而我们想按照任意对象属性对数组进行排序。为此，可以定义一个 `sort()`方法需要的比较函数，它接收两个参数，即要比较的值。但这个比较函数还需要想办法确定根据哪个属性来排序。这个问题可以通过定义一个根据属性名来创建比较函数的函数来解决。比如：

```
function createComparisonFunction(propertyName) {
  return function(object1, object2) {
    let value1 = object1[propertyName];
    let value2 = object2[propertyName];

    if (value1 < value2) {
      return -1;
    } else if (value1 > value2) {
      return 1;
    } else {
      return 0;
    }
  };
}
```

这个函数的语法乍一看比较复杂，但实际上就是在一个函数中返回另一个函数，注意那个 `return` 操作符。内部函数可以访问 `propertyName` 参数，并通过中括号语法取得要比较的对象的相应属性值。取得属性值以后，再按照 `sort()`方法的需要返回比较值就行了。这个函数可以像下面这样使用：

```
let data = [
  {name: "Zachary", age: 28},
  {name: "Nicholas", age: 29}
];

data.sort(createComparisonFunction("name"));
console.log(data[0].name);  // Nicholas
```

```
data.sort(createComparisonFunction("age"));
console.log(data[0].name);  // Zachary
```

在上面的代码中，数组 data 中包含两个结构相同的对象。每个对象都有一个 name 属性和一个 age 属性。默认情况下，sort()方法要对这两个对象执行 toString()，然后再决定它们的顺序，但这样得不到有意义的结果。而通过调用 createComparisonFunction("name")来创建一个比较函数，就可以根据每个对象 name 属性的值来排序，结果 name 属性值为"Nicholas"、age 属性值为 29 的对象会排在前面。而调用 createComparisonFunction("age")则会创建一个根据每个对象 age 属性的值来排序的比较函数，结果 name 属性值为"Zachary"、age 属性值为 28 的对象会排在前面。

10.9 函数内部

在 ECMAScript 5 中，函数内部存在两个特殊的对象：arguments 和 this。ECMAScript 6 又新增了 new.target 属性。

10.9.1 arguments

arguments 对象前面讨论过多次了，它是一个类数组对象，包含调用函数时传入的所有参数。这个对象只有以 function 关键字定义函数（相对于使用箭头语法创建函数）时才会有。虽然主要用于包含函数参数，但 arguments 对象其实还有一个 callee 属性，是一个指向 arguments 对象所在函数的指针。来看下面这个经典的阶乘函数：

```
function factorial(num) {
  if (num <= 1) {
    return 1;
  } else {
    return num * factorial(num - 1);
  }
}
```

阶乘函数一般定义成递归调用的，就像上面这个例子一样。只要给函数一个名称，而且这个名称不会变，这样定义就没有问题。但是，这个函数要正确执行就必须保证函数名是 factorial，从而导致了紧密耦合。使用 arguments.callee 就可以让函数逻辑与函数名解耦：

```
function factorial(num) {
  if (num <= 1) {
    return 1;
  } else {
    return num * arguments.callee(num - 1);
  }
}
```

这个重写之后的 factorial()函数已经用 arguments.callee 代替了之前硬编码的 factorial。这意味着无论函数叫什么名称，都可以引用正确的函数。考虑下面的情况：

```
let trueFactorial = factorial;

factorial = function() {
  return 0;
};

console.log(trueFactorial(5));  // 120
console.log(factorial(5));      // 0
```

这里，trueFactorial 变量被赋值为 factorial，实际上把同一个函数的指针又保存到了另一个位置。然后，factorial 函数又被重写为一个返回 0 的函数。如果像 factorial()最初的版本那样不使用 arguments.callee，那么像上面这样调用 trueFactorial()就会返回 0。不过，通过将函数与名称解耦，trueFactorial()就可以正确计算阶乘，而 factorial()则只能返回 0。

10.9.2 this

另一个特殊的对象是 this，它在标准函数和箭头函数中有不同的行为。

在标准函数中，this 引用的是把函数当成方法调用的上下文对象，这时候通常称其为 this 值（在网页的全局上下文中调用函数时，this 指向 windows）。来看下面的例子：

```
window.color = 'red';
let o = {
  color: 'blue'
};

function sayColor() {
  console.log(this.color);
}

sayColor();    // 'red'

o.sayColor = sayColor;
o.sayColor();  // 'blue'
```

定义在全局上下文中的函数 sayColor()引用了 this 对象。这个 this 到底引用哪个对象必须到函数被调用时才能确定。因此这个值在代码执行的过程中可能会变。如果在全局上下文中调用 sayColor()，这结果会输出"red"，因为 this 指向 window，而 this.color 相当于 window.color。而在把 sayColor()赋值给 o 之后再调用 o.sayColor()，this 会指向 o，即 this.color 相当于 o.color，所以会显示"blue"。

在箭头函数中，this 引用的是定义箭头函数的上下文。下面的例子演示了这一点。在对 sayColor()的两次调用中，this 引用的都是 window 对象，因为这个箭头函数是在 window 上下文中定义的：

```
window.color = 'red';
let o = {
  color: 'blue'
};

let sayColor = () => console.log(this.color);

sayColor();    // 'red'

o.sayColor = sayColor;
o.sayColor();  // 'red'
```

有读者知道，在事件回调或定时回调中调用某个函数时，this 值指向的并非想要的对象。此时将回调函数写成箭头函数就可以解决问题。这是因为箭头函数中的 this 会保留定义该函数时的上下文：

```
function King() {
  this.royaltyName = 'Henry';
  // this 引用 King 的实例
  setTimeout(() => console.log(this.royaltyName), 1000);
}
```

```
function Queen() {
  this.royaltyName = 'Elizabeth';

  // this 引用 window 对象
  setTimeout(function() { console.log(this.royaltyName); }, 1000);
}

new King();  // Henry
new Queen(); // undefined
```

> **注意** 函数名只是保存指针的变量。因此全局定义的 sayColor()函数和 o.sayColor()是同一个函数，只不过执行的上下文不同。

10.9.3 caller

ECMAScript 5 也会给函数对象上添加一个属性：caller。虽然 ECMAScript 3 中并没有定义，但所有浏览器除了早期版本的 Opera 都支持这个属性。这个属性引用的是调用当前函数的函数，或者如果是在全局作用域中调用的则为 null。比如：

```
function outer() {
  inner();
}

function inner() {
  console.log(inner.caller);
}
outer();
```

以上代码会显示 outer()函数的源代码。这是因为 ourter()调用了 inner()，inner.caller 指向 outer()。如果要降低耦合度，则可以通过 arguments.callee.caller 来引用同样的值：

```
function outer() {
  inner();
}

function inner() {
  console.log(arguments.callee.caller);
}

outer();
```

在严格模式下访问 arguments.callee 会报错。ECMAScript 5 也定义了 arguments.caller，但在严格模式下访问它会报错，在非严格模式下则始终是 undefined。这是为了分清 arguments.caller 和函数的 caller 而故意为之的。而作为对这门语言的安全防护，这些改动也让第三方代码无法检测同一上下文中运行的其他代码。

严格模式下还有一个限制，就是不能给函数的 caller 属性赋值，否则会导致错误。

10.9.4 new.target

ECMAScript 中的函数始终可以作为构造函数实例化一个新对象，也可以作为普通函数被调用。ECMAScript 6 新增了检测函数是否使用 new 关键字调用的 new.target 属性。如果函数是正常调用的，

则 new.target 的值是 undefined；如果是使用 new 关键字调用的，则 new.target 将引用被调用的构造函数。

```
function King() {
  if (!new.target) {
    throw 'King must be instantiated using "new"'
  }
  console.log('King instantiated using "new"');
}

new King(); // King instantiated using "new"
King();     // Error: King must be instantiated using "new"
```

10.10　函数属性与方法

前面提到过，ECMAScript 中的函数是对象，因此有属性和方法。每个函数都有两个属性：length 和 prototype。其中，length 属性保存函数定义的命名参数的个数，如下例所示：

```
function sayName(name) {
  console.log(name);
}

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

function sayHi() {
  console.log("hi");
}

console.log(sayName.length);  // 1
console.log(sum.length);      // 2
console.log(sayHi.length);    // 0
```

以上代码定义了 3 个函数，每个函数的命名参数个数都不一样。sayName()函数有 1 个命名参数，所以其 length 属性为 1。类似地，sum()函数有两个命名参数，所以其 length 属性是 2。而 sayHi()没有命名参数，其 length 属性为 0。

prototype 属性也许是 ECMAScript 核心中最有趣的部分。prototype 是保存引用类型所有实例方法的地方，这意味着 toString()、valueOf()等方法实际上都保存在 prototype 上，进而由所有实例共享。这个属性在自定义类型时特别重要。（相关内容已经在第 8 章详细介绍过了。）在 ECMAScript 5 中，prototype 属性是不可枚举的，因此使用 for-in 循环不会返回这个属性。

函数还有两个方法：apply()和 call()。这两个方法都会以指定的 this 值来调用函数，即会设置调用函数时函数体内 this 对象的值。apply()方法接收两个参数：函数内 this 的值和一个参数数组。第二个参数可以是 Array 的实例，但也可以是 arguments 对象。来看下面的例子：

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

function callSum1(num1, num2) {
  return sum.apply(this, arguments); // 传入 arguments 对象
}
```

```
function callSum2(num1, num2) {
  return sum.apply(this, [num1, num2]); // 传入数组
}

console.log(callSum1(10, 10));  // 20
console.log(callSum2(10, 10));  // 20
```

在这个例子中，callSum1()会调用 sum()函数，将 this 作为函数体内的 this 值（这里等于 window，因为是在全局作用域中调用的）传入，同时还传入了 arguments 对象。callSum2()也会调用 sum()函数，但会传入参数的数组。这两个函数都会执行并返回正确的结果。

> **注意** 在严格模式下，调用函数时如果没有指定上下文对象，则 this 值不会指向 window。除非使用 apply()或 call()把函数指定给一个对象，否则 this 的值会变成 undefined。

call()方法与 apply()的作用一样，只是传参的形式不同。第一个参数跟 apply()一样，也是 this 值，而剩下的要传给被调用函数的参数则是逐个传递的。换句话说，通过 call()向函数传参时，必须将参数一个一个地列出来，比如：

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

function callSum(num1, num2) {
  return sum.call(this, num1, num2);
}

console.log(callSum(10, 10)); // 20
```

这里的 callSum()函数必须逐个地把参数传给 call()方法。结果跟 apply()的例子一样。到底是使用 apply()还是 call()，完全取决于怎么给要调用的函数传参更方便。如果想直接传 arguments 对象或者一个数组，那就用 apply()；否则，就用 call()。当然，如果不用给被调用的函数传参，则使用哪个方法都一样。

apply()和 call()真正强大的地方并不是给函数传参，而是控制函数调用上下文即函数体内 this 值的能力。考虑下面的例子：

```
window.color = 'red';
let o = {
  color: 'blue'
};

function sayColor() {
  console.log(this.color);
}

sayColor();              // red

sayColor.call(this);     // red
sayColor.call(window);   // red
sayColor.call(o);        // blue
```

这个例子是在之前那个关于 this 对象的例子基础上修改而成的。同样，sayColor()是一个全局函数，如果在全局作用域中调用它，那么会显示"red"。这是因为 this.color 会求值为 window.color。如果在全局作用域中显式调用 sayColor.call(this)或者 sayColor.call(window)，则同样都会显

示"red"。而在使用 sayColor.call(o)把函数的执行上下文即 this 切换为对象 o 之后，结果就变成了显示"blue"了。

使用 call()或 apply()的好处是可以将任意对象设置为任意函数的作用域，这样对象可以不用关心方法。在前面例子最初的版本中，为切换上下文需要先把 sayColor()直接赋值为 o 的属性，然后再调用。而在这个修改后的版本中，就不需要这一步操作了。

ECMAScript 5 出于同样的目的定义了一个新方法：bind()。bind()方法会创建一个新的函数实例，其 this 值会被**绑定**到传给 bind()的对象。比如：

```
window.color = 'red';
var o = {
  color: 'blue'
};

function sayColor() {
  console.log(this.color);
}
let objectSayColor = sayColor.bind(o);
objectSayColor();  // blue
```

这里，在 sayColor()上调用 bind()并传入对象 o 创建了一个新函数 objectSayColor()。objectSayColor()中的 this 值被设置为 o，因此直接调用这个函数，即使是在全局作用域中调用，也会返回字符串"blue"。

对函数而言，继承的方法 toLocaleString()和 toString()始终返回函数的代码。返回代码的具体格式因浏览器而异。有的返回源代码，包含注释，而有的只返回代码的内部形式，会删除注释，甚至代码可能被解释器修改过。由于这些差异，因此不能在重要功能中依赖这些方法返回的值，而只应在调试中使用它们。继承的方法 valueOf()返回函数本身。

10.11　函数表达式

函数表达式虽然更强大，但也更容易让人迷惑。我们知道，定义函数有两种方式：函数声明和函数表达式。函数声明是这样的：

```
function functionName(arg0, arg1, arg2) {
  // 函数体
}
```

函数声明的关键特点是**函数声明提升**，即函数声明会在代码执行之前获得定义。这意味着函数声明可以出现在调用它的代码之后：

```
sayHi();
function sayHi() {
  console.log("Hi!");
}
```

这个例子不会抛出错误，因为 JavaScript 引擎会先读取函数声明，然后再执行代码。

第二种创建函数的方式就是函数表达式。函数表达式有几种不同的形式，最常见的是这样的：

```
let functionName = function(arg0, arg1, arg2) {
  // 函数体
};
```

函数表达式看起来就像一个普通的变量定义和赋值，即创建一个函数再把它赋值给一个变量 functionName。这样创建的函数叫作**匿名函数**（anonymous funtion），因为 function 关键字后面没有标识符。（匿名函数有也时候也被称为**兰姆达函数**）。未赋值给其他变量的匿名函数的 name 属性是空字符串。

函数表达式跟 JavaScript 中的其他表达式一样，需要先赋值再使用。下面的例子会导致错误：

```
sayHi();  // Error! function doesn't exist yet
let sayHi = function() {
  console.log("Hi!");
};
```

理解函数声明与函数表达式之间的区别，关键是理解提升。比如，以下代码的执行结果可能会出乎意料：

```
// 千万别这样做!
if (condition) {
  function sayHi() {
    console.log('Hi!');
  }
} else {
  function sayHi() {
    console.log('Yo!');
  }
}
```

这段代码看起来很正常，就是如果 condition 为 true，则使用第一个 sayHi()定义；否则，就使用第二个。事实上，这种写法在 ECAMScript 中不是有效的语法。JavaScript 引擎会尝试将其纠正为适当的声明。问题在于浏览器纠正这个问题的方式并不一致。多数浏览器会忽略 condition 直接返回第二个声明。Firefox 会在 condition 为 true 时返回第一个声明。这种写法很危险，不要使用。不过，如果把上面的函数声明换成函数表达式就没问题了：

```
// 没问题
let sayHi;
if (condition) {
  sayHi = function() {
    console.log("Hi!");
  };
} else {
  sayHi = function() {
    console.log("Yo!");
  };
}
```

这个例子可以如预期一样，根据 condition 的值为变量 sayHi 赋予相应的函数。

创建函数并赋值给变量的能力也可以用于在一个函数中把另一个函数当作值返回：

```
function createComparisonFunction(propertyName) {
  return function(object1, object2) {
    let value1 = object1[propertyName];
    let value2 = object2[propertyName];

    if (value1 < value2) {
      return -1;
    } else if (value1 > value2) {
      return 1;
    } else {
```