新的简写方法的语法遵循同样的模式，但开发者要放弃给函数表达式命名（不过给作为方法的函数命名通常没什么用）。相应地，这样也可以明显缩短方法声明。

以下代码和之前的代码在行为上是等价的：

```
let person = {
  sayName(name) {
    console.log(`My name is ${name}`);
  }
};

person.sayName('Matt'); // My name is Matt
```

简写方法名对获取函数和设置函数也是适用的：

```
let person = {
  name_: '',
  get name() {
    return this.name_;
  },
  set name(name) {
    this.name_ = name;
  },
  sayName() {
    console.log(`My name is ${this.name_}`);
  }
};

person.name = 'Matt';
person.sayName(); // My name is Matt
```

简写方法名与可计算属性键相互兼容：

```
const methodKey = 'sayName';

let person = {
  [methodKey](name) {
    console.log(`My name is ${name}`);
  }
}

person.sayName('Matt'); // My name is Matt
```

> **注意**　简写方法名对于本章后面介绍的 ECMAScript 6 的类更有用。

## 8.1.7　对象解构

ECMAScript 6 新增了对象解构语法，可以在一条语句中使用嵌套数据实现一个或多个赋值操作。简单地说，对象解构就是使用与对象匹配的结构来实现对象属性赋值。

下面的例子展示了两段等价的代码，首先是不使用对象解构的：

```
// 不使用对象解构
let person = {
  name: 'Matt',
  age: 27
};
```

```
let personName = person.name,
    personAge = person.age;

console.log(personName); // Matt
console.log(personAge);  // 27
```

然后，是使用对象解构的：

```
// 使用对象解构
let person = {
  name: 'Matt',
  age: 27
};

let { name: personName, age: personAge } = person;

console.log(personName);  // Matt
console.log(personAge);   // 27
```

使用解构，可以在一个类似对象字面量的结构中，声明多个变量，同时执行多个赋值操作。如果想让变量直接使用属性的名称，那么可以使用简写语法，比如：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, age } = person;

console.log(name);  // Matt
console.log(age);   // 27
```



解构赋值不一定与对象的属性匹配。赋值的时候可以忽略某些属性，而如果引用的属性不存在，则该变量的值就是 undefined：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, job } = person;

console.log(name);  // Matt
console.log(job);   // undefined
```

也可以在解构赋值的同时定义默认值，这适用于前面刚提到的引用的属性不存在于源对象中的情况：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, job='Software engineer' } = person;

console.log(name); // Matt
console.log(job);  // Software engineer
```

解构在内部使用函数 ToObject()（不能在运行时环境中直接访问）把源数据结构转换为对象。这意味着在对象解构的上下文中，原始值会被当成对象。这也意味着（根据 ToObject()的定义），null 和 undefined 不能被解构，否则会抛出错误。

```
let { length } = 'foobar';
console.log(length);        // 6

let { constructor: c } = 4;
console.log(c === Number);  // true

let { _ } = null;           // TypeError

let { _ } = undefined;      // TypeError
```

解构并不要求变量必须在解构表达式中声明。不过，如果是给事先声明的变量赋值，则赋值表达式必须包含在一对括号中：

```
let personName, personAge;

let person = {
  name: 'Matt',
  age: 27
};

({name: personName, age: personAge} = person);

console.log(personName, personAge); // Matt, 27
```

### 1. 嵌套解构

解构对于引用嵌套的属性或赋值目标没有限制。为此，可以通过解构来复制对象属性：

```
let person = {
  name: 'Matt',
  age: 27,
  job: {
    title: 'Software engineer'
  }
};
let personCopy = {};


({
  name: personCopy.name,
  age: personCopy.age,
  job: personCopy.job
} = person);

// 因为一个对象的引用被赋值给 personCopy，所以修改
// person.job 对象的属性也会影响 personCopy
person.job.title = 'Hacker'

console.log(person);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }

console.log(personCopy);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }
```

解构赋值可以使用嵌套结构，以匹配嵌套的属性：

```
let person = {
  name: 'Matt',
  age: 27,
  job: {
    title: 'Software engineer'
  }
};

// 声明 title 变量并将 person.job.title 的值赋给它
let { job: { title } } = person;

console.log(title); // Software engineer
```

在外层属性没有定义的情况下不能使用嵌套解构。无论源对象还是目标对象都一样：

```
let person = {
  job: {
    title: 'Software engineer'
  }
};
let personCopy = {};

// foo 在源对象上是 undefined
({
  foo: {
    bar: personCopy.bar
  }
} = person);
// TypeError: Cannot destructure property 'bar' of 'undefined' or 'null'.

// job 在目标对象上是 undefined
({
  job: {
    title: personCopy.job.title
  }
} = person);
// TypeError: Cannot set property 'title' of undefined
```



#### 2. 部分解构

需要注意的是，涉及多个属性的解构赋值是一个输出无关的顺序化操作。如果一个解构表达式涉及多个赋值，开始的赋值成功而后面的赋值出错，则整个解构赋值只会完成一部分：

```
let person = {
  name: 'Matt',
  age: 27
};

let personName, personBar, personAge;

try {
  // person.foo 是 undefined，因此会抛出错误
  ({name: personName, foo: { bar: personBar }, age: personAge} = person);
} catch(e) {}

console.log(personName, personBar, personAge);
// Matt, undefined, undefined
```

#### 3. 参数上下文匹配

在函数参数列表中也可以进行解构赋值。对参数的解构赋值不会影响 `arguments` 对象，但可以在

函数签名中声明在函数体内使用局部变量：

```
let person = {
  name: 'Matt',
  age: 27
};

function printPerson(foo, {name, age}, bar) {
  console.log(arguments);
  console.log(name, age);
}

function printPerson2(foo, {name: personName, age: personAge}, bar) {
  console.log(arguments);
  console.log(personName, personAge);
}

printPerson('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27

printPerson2('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27
```

## 8.2　创建对象

虽然使用 Object 构造函数或对象字面量可以方便地创建对象，但这些方式也有明显不足：创建具有同样接口的多个对象需要重复编写很多代码。

### 8.2.1　概述

综观 ECMAScript 规范的历次发布，每个版本的特性似乎都出人意料。ECMAScript 5.1 并没有正式支持面向对象的结构，比如类或继承。但是，正如接下来几节会介绍的，巧妙地运用原型式继承可以成功地模拟同样的行为。

ECMAScript 6 开始正式支持类和继承。ES6 的类旨在完全涵盖之前规范设计的基于原型的继承模式。不过，无论从哪方面看，ES6 的类都仅仅是封装了 ES5.1 构造函数加原型继承的语法糖而已。

> **注意**　不要误会：采用面向对象编程模式的 JavaScript 代码还是应该使用 ECMAScript 6 的类。但不管怎么说，理解 ES6 类出现之前的惯例总是有益无害的。特别是 ES6 的类定义本身就相当于对原有结构的封装。因此，在介绍 ES6 的类之前，本书会循序渐进地介绍被类取代的那些底层概念。

### 8.2.2　工厂模式

工厂模式是一种众所周知的设计模式，广泛应用于软件工程领域，用于抽象创建特定对象的过程。（本书后面还会讨论其他设计模式及其在 JavaScript 中的实现。）下面的例子展示了一种按照特定接口创建对象的方式：

```
function createPerson(name, age, job) {
  let o = new Object();
  o.name = name;
  o.age = age;
  o.job = job;
  o.sayName = function() {
    console.log(this.name);
  };
  return o;
}

let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");
```

这里，函数 createPerson()接收 3 个参数，根据这几个参数构建了一个包含 Person 信息的对象。可以用不同的参数多次调用这个函数，每次都会返回包含 3 个属性和 1 个方法的对象。这种工厂模式虽然可以解决创建多个类似对象的问题，但没有解决对象标识问题（即新创建的对象是什么类型）。

### 8.2.3 构造函数模式

前面几章提到过，ECMAScript 中的构造函数是用于创建特定类型对象的。像 Object 和 Array 这样的原生构造函数，运行时可以直接在执行环境中使用。当然也可以自定义构造函数，以函数的形式为自己的对象类型定义属性和方法。

比如，前面的例子使用构造函数模式可以这样写：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg
```

在这个例子中，Person()构造函数代替了 createPerson()工厂函数。实际上，Person()内部的代码跟 createPerson()基本是一样的，只是有如下区别。

- 没有显式地创建对象。
- 属性和方法直接赋值给了 this。
- 没有 return。

另外，要注意函数名 Person 的首字母大写了。按照惯例，构造函数名称的首字母都是要大写的，非构造函数则以小写字母开头。这是从面向对象编程语言那里借鉴的，有助于在 ECMAScript 中区分构造函数和普通函数。毕竟 ECMAScript 的构造函数就是能创建对象的函数。

要创建 Person 的实例，应使用 new 操作符。以这种方式调用构造函数会执行如下操作。

(1) 在内存中创建一个新对象。

(2) 这个新对象内部的[[Prototype]]特性被赋值为构造函数的 prototype 属性。

(3) 构造函数内部的 this 被赋值为这个新对象（即 this 指向新对象）。

(4) 执行构造函数内部的代码（给新对象添加属性）。

(5) 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

上一个例子的最后，person1 和 person2 分别保存着 Person 的不同实例。这两个对象都有一个 constructor 属性指向 Person，如下所示：

```
console.log(person1.constructor == Person);  // true
console.log(person2.constructor == Person);  // true
```

constructor 本来是用于标识对象类型的。不过，一般认为 instanceof 操作符是确定对象类型更可靠的方式。前面例子中的每个对象都是 Object 的实例，同时也是 Person 的实例，如下面调用 instanceof 操作符的结果所示：

```
console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

定义自定义构造函数可以确保实例被标识为特定类型，相比于工厂模式，这是一个很大的好处。在这个例子中，person1 和 person2 之所以也被认为是 Object 的实例，是因为所有自定义对象都继承自 Object（后面再详细讨论这一点）。

构造函数不一定要写成函数声明的形式。赋值给变量的函数表达式也可以表示构造函数：

```
let Person = function(name, age, job) {
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg

console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

在实例化时，如果不想传参数，那么构造函数后面的括号可加可不加。只要有 new 操作符，就可以调用相应的构造函数：

```
function Person() {
  this.name = "Jake";
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person();
let person2 = new Person;
```

```
person1.sayName();  // Jake
person2.sayName();  // Jake

console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

#### 1. 构造函数也是函数

构造函数与普通函数唯一的区别就是调用方式不同。除此之外，构造函数也是函数。并没有把某个函数定义为构造函数的特殊语法。任何函数只要使用 new 操作符调用就是构造函数，而不使用 new 操作符调用的函数就是普通函数。比如，前面的例子中定义的 Person()可以像下面这样调用：

```
// 作为构造函数
let person = new Person("Nicholas", 29, "Software Engineer");
person.sayName();    // "Nicholas"

// 作为函数调用
Person("Greg", 27, "Doctor");   // 添加到 window 对象
window.sayName();    // "Greg"

// 在另一个对象的作用域中调用
let o = new Object();
Person.call(o, "Kristen", 25, "Nurse");
o.sayName();   // "Kristen"
```

这个例子一开始展示了典型的构造函数调用方式，即使用 new 操作符创建一个新对象。然后是普通函数的调用方式，这时候没有使用 new 操作符调用 Person()，结果会将属性和方法添加到 window 对象。这里要记住，在调用一个函数而没有明确设置 this 值的情况下（即没有作为对象的方法调用，或者没有使用 call()/apply()调用），this 始终指向 Global 对象（在浏览器中就是 window 对象）。因此在上面的调用之后，window 对象上就有了一个 sayName()方法，调用它会返回"Greg"。最后展示的调用方式是通过 call()（或 apply()）调用函数，同时将特定对象指定为作用域。这里的调用将对象 o 指定为 Person()内部的 this 值，因此执行完函数代码后，所有属性和 sayName()方法都会添加到对象 o 上面。

#### 2. 构造函数的问题

构造函数虽然有用，但也不是没有问题。构造函数的主要问题在于，其定义的方法会在每个实例上都创建一遍。因此对前面的例子而言，person1 和 person2 都有名为 sayName()的方法，但这两个方法不是同一个 Function 实例。我们知道，ECMAScript 中的函数是对象，因此每次定义函数时，都会初始化一个对象。逻辑上讲，这个构造函数实际上是这样的：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = new Function("console.log(this.name)"); // 逻辑等价
}
```

这样理解这个构造函数可以更清楚地知道，每个 Person 实例都会有自己的 Function 实例用于显示 name 属性。当然了，以这种方式创建函数会带来不同的作用域链和标识符解析。但创建新 Function 实例的机制是一样的。因此不同实例上的函数虽然同名却不相等，如下所示：

```
console.log(person1.sayName == person2.sayName); // false
```

因为都是做一样的事，所以没必要定义两个不同的 Function 实例。况且，this 对象可以把函数与对象的绑定推迟到运行时。

要解决这个问题，可以把函数定义转移到构造函数外部：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = sayName;
}

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

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg
```

在这里，sayName()被定义在了构造函数外部。在构造函数内部，sayName 属性等于全局 sayName()函数。因为这一次 sayName 属性中包含的只是一个指向外部函数的指针，所以 person1 和 person2 共享了定义在全局作用域上的 sayName()函数。这样虽然解决了相同逻辑的函数重复定义的问题，但全局作用域也因此被搞乱了，因为那个函数实际上只能在一个对象上调用。如果这个对象需要多个方法，那么就要在全局作用域中定义多个函数。这会导致自定义类型引用的代码不能很好地聚集一起。这个新问题可以通过原型模式来解决。

### 8.2.4　原型模式

每个函数都会创建一个 prototype 属性，这个属性是一个对象，包含应该由特定引用类型的实例共享的属性和方法。实际上，这个对象就是通过调用构造函数创建的对象的原型。使用原型对象的好处是，在它上面定义的属性和方法可以被对象实例共享。原来在构造函数中直接赋给对象实例的值，可以直接赋值给它们的原型，如下所示：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
person1.sayName(); // "Nicholas"

let person2 = new Person();
person2.sayName(); // "Nicholas"

console.log(person1.sayName == person2.sayName); // true
```

使用函数表达式也可以：

```
let Person = function() {};

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
person1.sayName();   // "Nicholas"

let person2 = new Person();
person2.sayName();   // "Nicholas"

console.log(person1.sayName == person2.sayName); // true
```

这里，所有属性和 sayName()方法都直接添加到了 Person 的 prototype 属性上，构造函数体中什么也没有。但这样定义之后，调用构造函数创建的新对象仍然拥有相应的属性和方法。与构造函数模式不同，使用这种原型模式定义的属性和方法是由所有实例共享的。因此 person1 和 person2 访问的都是相同的属性和相同的 sayName()函数。要理解这个过程，就必须理解 ECMAScript 中原型的本质。

#### 1. 理解原型

无论何时，只要创建一个函数，就会按照特定的规则为这个函数创建一个 prototype 属性（指向原型对象）。默认情况下，所有原型对象自动获得一个名为 constructor 的属性，指回与之关联的构造函数。对前面的例子而言，Person.prototype.constructor 指向 Person。然后，因构造函数而异，可能会给原型对象添加其他属性和方法。

在自定义构造函数时，原型对象默认只会获得 constructor 属性，其他的所有方法都继承自 Object。每次调用构造函数创建一个新实例，这个实例的内部[[Prototype]]指针就会被赋值为构造函数的原型对象。脚本中没有访问这个[[Prototype]]特性的标准方式，但 Firefox、Safari 和 Chrome 会在每个对象上暴露__proto__属性，通过这个属性可以访问对象的原型。在其他实现中，这个特性完全被隐藏了。关键在于理解这一点：实例与构造函数原型之间有直接的联系，但实例与构造函数之间没有。

这种关系不好可视化，但可以通过下面的代码来理解原型的行为：

```
/**
 * 构造函数可以是函数表达式
 * 也可以是函数声明，因此以下两种形式都可以：
 *   function Person() {}
 *   let Person = function() {}
 */
function Person() {}

/**
 * 声明之后，构造函数就有了一个
 * 与之关联的原型对象：
 */
console.log(typeof Person.prototype);
console.log(Person.prototype);
// {
//   constructor: f Person(),
//   __proto__: Object
```