假设你有一个名为 config 的对象，已经有了一部分值，但可能不是全部，现在你想要把所有空槽的位置用默认值设定，但又不想覆盖已经存在的部分：

```
var config = {
    options: {
        remove: false,
        instance: null
    }
};
```

当然你可以像过去那样手动实现：

```
config.options = config.options || {};
config.options.remove = (config.options.remove !== undefined) ?
    config.options.remove : defaults.options.remove;
config.options.enable = (config.options.enable !== undefined) ?
    config.options.enable : defaults.options.enable;
...
```

可恶。

还有人会喜欢通过覆盖赋值方法来实现这个任务。你可能会被 ES6 的 Object.assign(..) 工具诱惑（参见第 6 章）先从 defaults 克隆属性，然后用从 config 克隆的属性来覆盖。就像下面这样：

```
config = Object.assign( {}, defaults, config );
```

这看起来好多了，对吧？但是存在一个严重问题！ Object.assign(..) 是浅操作，也就是说在复制 defaults.options 的时候，只会复制对象引用，而不会深层复制这个对象的属性到 config.options 对象。需要在对象树的所有层次（某种“递归”）上应用 object.assign(..) 才能得到期望的深层克隆。

很多 JavaScript 工具库 / 框架提供了自己的对象深复制支持，但这些方法和它们的使用技巧超出了本部分的讨论范围。

所以让我们来探讨一下带默认值的 ES6 对象解构是否能够帮助实现这一点：

```
config.options = config.options || {};
config.log = config.log || {};
{
    options: {
        remove: config.options.remove = default.options.remove,
        enable: config.options.enable = default.options.enable,
        instance: config.options.instance =
                    default.options.instance
    } = {},
    log: {
```

```
            warn: config.log.warn = default.log.warn,
            error: config.log.error = default.log.error
        } = {}
    } = config;
```

我认为这虽然没有提供了虚假保证（实际上只是浅复制）的 `Object.assign(..)` 方法那么优美，但还是要比手动方法要好一点。虽然不幸的是，它还是有点繁复。

前面代码的方法之所以有效，是因为我 hack 了解构和默认值机制，实现了属性 `=== undefined` 检查和赋值决策。这里的巧妙之处在于，我们解构了 `config`（参见代码结尾处的 `= config`），但通过 `config.options.enable` 赋值引用，马上又把所有解构值赋值回到了 `config`。

还是有点繁杂。看我们能不能让实现更简洁一些。

如果确定要解构的所有各种属性都没有重名的话，下面的技巧是最优的。即使不是这样，也可以使用这一技巧，但是就没那么优美了——需要分阶段解构，或者创建唯一局部变量作为临时别名。

如果我们把所有属性彻底解构到顶层变量中，接着就可以立即重组它们来重新构造原来的嵌套对象结构了。

但是，所有这些悬置的临时变量会污染作用域。所以，我们用一个 `{ }` 把这块包起来成为一个块作用域（参见 2.1 节）：

```
// 把defaults合并进config
{
    // (带默认值赋值的)解构
    let {
        options: {
            remove = defaults.options.remove,
            enable = defaults.options.enable,
            instance = defaults.options.instance
        } = {},
        log: {
           warn = defaults.log.warn,
           error = defaults.log.error
        } = {}
    } = config;

    // 重组
    config = {
        options: { remove, enable, instance },
        log: { warn, error }
    };
}
```

这看起来好多了，是不是？

还可以用箭头 IIFE 代替一般的 { } 块和 let 声明来实现块封装。解构赋值 / 默认值会被放在参数列表中，而重组的过程会被放在函数体的 return 语句中。

重组部分中的 { warn, error } 这种语法形式对你来说可能有点陌生；这称为“简洁属性”，我们将在下一小节介绍！

2.6 对象字面量扩展

ES6 为普通 { .. } 对象字面量新增了几个重要的便利扩展。

2.6.1 简洁属性

你对下面这种形式的对象字面量声明肯定已经非常熟悉了：

```
var x = 2, y = 3,
    o = {
        x: x,
        y: y
    };
```

如果觉得总是要写 x: x 令人厌烦的话，那么这里有一个好消息。就是如果你需要定义一个与某个词法标识符同名的属性的话，可以把 x: x 简写为 x。考虑：

```
var x = 2, y = 3,
    o = {
        x,
        y
    };
```

2.6.2 简洁方法

与刚刚介绍的简洁属性思路类似，为了方便表达，关联到对象字面量属性上的函数也有简洁形式。

老方法：

```
var o = {
    x: function(){
        // ..
    },
    y: function(){
        // ..
    }
}
```

在 ES6 中则可以：

```
var o = {
    x() {
        // ..
    },
    y() {
        // ..
    }
}
```

尽管看上去 `x() { .. }` 就是 `x: function(){ .. }` 的简写形式，但简洁方法有特殊的性质，这是它们的前辈所不具备的；具体来说，就是支持 `super`（参见 2.6.5 节）。

生成器（参见第 4 章）也有一个简洁方法形式：

```
var o = {
    *foo() { .. }
};
```

1. 简洁未命名

这种方便的简写形式是很诱人的，但有一个微妙的细节需要注意。我们分析下面这段前 ES6 代码来展示这一点，这段代码可能令人想要通过简洁方法重构：

```
function runSomething(o) {
    var x = Math.random(),
        y = Math.random();

    return o.something( x, y );
}

runSomething( {
    something: function something(x,y) {
        if (x > y) {
            // 交换x和y的递归调用
            return something( y, x );
        }

        return y - x;
     }
} );
```

这段简单直接的代码就是产生两个随机数字，然后用大的减去小的。但这里重点不是这段代码做了些什么，而是它是如何做的。我们重点关注对象字面量和函数定义，可以看到如下所示：

```
runSomething( {
    something: function something(x,y) {
```

```
        // ..
    }
} );
```

为什么这里既有 something: 又有 function something？这不是重复吗？实际上并不是，二者各有不同的作用，都是必要的。属性 something 使得我们能够通过 o.something(..) 来调用，像是它的公开名称。而第二个 something 是一个词法名称，用于在其自身内部引用这个函数，目的是用于递归。

你能看出为什么 return something(y,x) 这一行需要名称 something 来引用这个函数吗？这个对象没有词法名称，因此它无法使用 return o.something(y,x) 或者某种类似的形式。

当对象字面量有一个标识名称时，这实际上是一个很常见的方法。比如：

```
var controller = {
    makeRequest: function(..){
        // ..
        controller.makeRequest(..);
    }
};
```

这是一个好方法吗？可能是，也可能不是。这里是在假定名称 controller 将会一直指向所需的对象。但是可能实际并非如此——makeRequest(..) 函数并不控制外部代码，因此无法强制这一点。这可能反过来会伤到你自己。

还有一些人可能喜欢采用 this 这种方法来定义：

```
var controller = {
    makeRequest: function(..){
        // ..
        this.makeRequest(..);
    }
};
```

这看起来不错，如果总是通过 controller.makeRequest(..) 调用方法也可以工作。但是，如果要像下面这么做的话，现在就有了一个 this 绑定陷阱：

```
btn.addEventListener( "click", controller.makeRequest, false );
```

当然，可以通过传递 controller.makeRequest.bind(controller) 作为处理函数引用来绑定这个事件。但是这种方法就不怎么吸引人了。

或者，如果内层的 this.makeRequest(..) 调用需要从嵌套函数内部调用会怎样呢？那就得有另外一个 this 绑定，这种情况通常用 var self = this 这种 hack 的方法来解决，就像：

```
var controller = {
    makeRequest: function(..){
```

```
        var self = this;

            btn.addEventListener( "click", function(){
                // ..
                self.makeRequest(..);
            }, false );
        }
};
```

这就更恶心了。

关于 this 绑定规则和陷阱的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分的 1~2 章。

好，那么所有这些又和简洁方法有什么关系呢？回想一下我们的 something(..) 方法定义：

```
runSomething( {
    something: function something(x,y) {
        // ..
    }
} );
```

这里的第二个 something 提供了一个超级方便的词法标识符，总是指向这个函数本身，为我们提供了一个完美的用于递归、事件绑定 / 解绑定等的引用——不会和 this 纠缠也不需要并不可靠的对象引用。

非常棒！

所以，现在我们可以把这个函数引用重构为下面的 ES6 简洁方法形式：

```
runSomething( {
    something(x,y) {
        if (x > y) {
            return something( y, x );
        }

        return y - x;
    }
} );
```

第一眼看上去很好，但是这段代码会崩溃。return something(..) 调用将找不到 something 标识符，因此会得到一个 ReferenceError。这是为什么呢？

前面的 ES6 代码片段会被解释为：

```
runSomething( {
    something: function(x,y){
        if (x > y) {
```

```
            return something( y, x );
        }

        return y - x;
    }
} );
```

仔细观察。看到问题所在了吗？这个简洁方法定义意味着 `something: function(x,y)`。看出我们依赖的第二个 `something` 是如何被省略了吗？换句话说，简洁方法意味着匿名函数表达式。

啊，有点恶心。

可能你会把 `=>` 箭头函数当作是对这种情况的一个好的解决方案，但是它同样也是不够的，因为它们也是匿名函数表达式。我们将在 2.8 节介绍这一部分。

部分挽回局面的消息是我们的 `something(x,y)` 简洁方法不会是完全匿名的。参见 7.1 节来获取关于 ES6 函数名推导规则的信息。对于我们的递归来说这没有什么帮助，但是至少有助于调试。

所以对于简洁方法能得出什么结论呢？它们简洁方便。但是应该只在不需要它们执行递归或者事件绑定 / 解绑定的时候使用。否则的话，就按照老式的 `something: function something(..)` 方法来定义吧。

大量方法可能会从简洁方法定义中获益，所以这是好消息！只是需要注意这几种有命名问题的情况。

2.ES5 Getter/Setter

严格说来，ES5 定义了 getter/setter 字面量形式，但是没怎么被使用，主要是因为缺少 transpiler 来处理这个新语法（实际上也是 ES5 新增的唯一主要新语法）。所以尽管这并不是一个新的 ES6 特性，我们还是简单介绍一下这种形式，因为很可能在 ES6 及以后它们会得到更广泛地使用。

考虑：

```
var o = {
    __id: 10,
    get id() { return this.__id++; },
    set id(v) { this.__id = v; }
}

o.id;           // 10
o.id;           // 11
o.id = 20;
```

```
o.id;           // 20

// and:
o.__id;         // 21
o.__id;         // 21--保持不变!
```

这些 getter 和 setter 字面量形式也可以出现在类中，参见第 3 章。

可能不是显而易见，实际上 setter 字面量必须有且只有一个声明参数；省略这个参数或者列出多余的都是语法错误。所需的单个参数可以使用解构和默认值（例如，`set id({ id: v = 0 }) { .. }`），但是 gather/rest`...` 是不允许的（`set id(...v) { .. }`）。

2.6.3 计算属性名

你可能也经历过下面代码片段中的这种情况，其中的一个或多个属性名来自于某个表达式，因此无法用对象字面量表达。

```
var prefix = "user_";

var o = {
    baz: function(..){ .. }
};

o[ prefix + "foo" ] = function(..){ .. };
o[ prefix + "bar" ] = function(..){ .. };
..
```

ES6 对对象字面定义新增了一个语法，用来支持指定一个要计算的表达式，其结果作为属性名。考虑：

```
var prefix = "user_";

var o = {
    baz: function(..){ .. },
    [ prefix + "foo" ]: function(..){ .. },
    [ prefix + "bar" ]: function(..){ .. }
    ..
};
```

对象字面定义属性名位置的 `[ .. ]` 中可以放置任意合法表达式。

计算属性名最常见的用法可能就是和 `Symbols` 共同使用（我们将在 2.13 节中介绍）。比如：

```
var o = {
    [Symbol.toStringTag]: "really cool thing",
    ..
};
```

Symbol.toStringTag 是一个特殊的内置值，我们用 [..] 语法为其求值，所以我们可以把值 "really cool thing" 赋给这个特殊的属性名。

计算属性名也可以作为简洁方法或者简洁生成器的名称出现：

```
var o = {
    ["f" + "oo"]() { .. }   // 计算出的简洁方法
    *["b" + "ar"]() { .. }  // 计算出的简洁生成器
};
```

2.6.4 设定 [[Prototype]]

这里我们不会详细介绍原型，要想了解更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分。

有时候在声明对象字面量的时候设定这个对象的 [[Prototype]] 是有用的。下面的用法在很多 JavaScript 引擎中已经作为非标准扩展有一段时间了，而在 ES6 中这已经标准化了：

```
var o1 = {
    // ..
};

var o2 = {
    __proto__: o1,
    // ..
};
```

o2 通过普通的对象字面量声明，但是它也 [[Prototype]] 连接到了 o1。这里的 __proto__ 属性名也可以是字符串 "__proto__"，但是注意它不能是计算属性名结果（参见前一节）。

退一步讲，对 __proto__ 的使用是有争议的。它是 JavaScript 多年前的属性扩展，最后被 ES6 标准化，但似乎标准化得不情不愿。许多开发者认为不应该使用它。实际上，它是在 ES6 的“附录 B”中出现的，这一部分列出的都是 JavaScript 只因兼容性问题不得不标准化的特性。

我勉强支持 __proto__ 作为对象字面量定义的一个键值，但我绝对不支持作为对象属性形式来使用它，比如 o.__proto__。这种形式既是 getter 又是 setter（也是由于兼容性的原因），但是肯定还有更好的选择。参见本系列《你不知道的 JavaScript（上卷）》第二部分。

要为已经存在的对象设定 [[Prototype]]，可以使用 ES6 工具 Object.setPrototypeOf(..)。考虑：

```
var o1 = {
    // ..
```

```
};

var o2 = {
    // ..
};

Object.setPrototypeOf( o2, o1 );
```

我们会在第 6 章再次讨论 Object。“Object.setPrototypeOf(..) 静态函数”一节详细介绍了 Object.setProto typeOf(..)。还可以参考 6.2.4 节了解把 o2 原型关联到 o1 的另外一种形式。

2.6.5 super 对象

通常把 super 看作只与类相关。但是，鉴于 JavaScript 的原型类而非类对象的本质，super 对于普通（plain）对象的简洁方法也一样有效，特性也基本相同。

考虑：

```
var o1 = {
    foo() {
        console.log( "o1:foo" );
    }
};

var o2 = {
    foo() {
        super.foo();
        console.log( "o2:foo" );
    }
};

Object.setPrototypeOf( o2, o1 );

o2.foo();       // o1:foo
                // o2:foo
```

super 只允许在简洁方法中出现，而不允许在普通函数表达式属性中出现。也只允许以 super.XXX 的形式（用于属性 / 方法访问）出现，而不能以 super() 的形式出现。

o2.foo() 方法中的 super 引用静态锁定到 o2，具体说是锁定到 o2 的 [[Prototype]]。基本上这里的 super 就是 Object.getPrototypeOf(o2)——当然会决议到 o1——这是它如何找到并调用 o1.foo() 的过程。

关于 super 的完整细节，参见 3.4 节。