类似地，{ x: x, y: y, z: z } 指定了一个把 bar() 的对象值分解为独立的变量赋值的“模式”。

### 2.4.1 对象属性赋值模式

让我们继续深入探讨一下前面代码中的语法 { x: x, .. }。实际上，如果属性名和要赋值的变量名相同，这种语法还可以更简短一些：

```
var { x, y, z } = bar();

console.log( x, y, z );             // 4 5 6
```

很酷吧，是不是？

但是 { x, .. } 是省略掉了 x: 部分还是 : x 部分呢？实际上我们使用这个缩写语法的时候是略去了 x: 部分。看起来这似乎是无关紧要的细节，但不久以后你就会发现这一点的重要性。

如果可以使用这种简短形式，谁还会再用那种更冗长的形式呢？但是更长的形式支持把属性赋给非同名变量，实际上有时候这是非常有用的：

```
var { x: bam, y: baz, z: bap } = bar();

console.log( bam, baz, bap );       // 4 5 6
console.log( x, y, z );             // ReferenceError
```

关于对象解构形式的这个变体有一个很微妙、但是极其重要的细节需要理解。为了说明为什么要对这一点格外小心，我们考虑一下下面指定一般对象字面值的“模式”：

```
var X = 10, Y = 20;

var o = { a: X, b: Y };

console.log( o.a, o.b );            // 10 20
```

在 { a: X, b: Y } 中，我们知道 a 是对象属性，而 X 是要赋给它的值。换句话说，这个语法模式是 target: source，或者更明确地说是 property-alias：value。因为它和赋值符 = 的模式一样都是 target = source，所以我们很直观地理解了这一点。

但是，在使用对象解构赋值的时候——也就是说，把看起来像是对象字面值的语法 { .. } 放在 = 运算符的左侧——反转了 target: source 模式。

回忆一下：

```
var { x: bam, y: baz, z: bap } = bar();
```

这里的语法模式是 souce: target（或者说是 value:variable-alias）。x: bam 表示 x 属性是源值，而 bam 是要赋值的目标变量。换句话说，对象字面值是 target <-- source，而对

象解构赋值是 source --> target。看到这里是如何反转了吧?

还可以这么看待这种语法形式，可能会帮助理解，减少迷惑。考虑：

```
var aa = 10, bb = 20;

var o = { x: aa, y: bb };
var     { x: AA, y: BB } = o;

console.log( AA, BB );              // 10 20
```

在 { x: aa, y: bb } 这一行，x 和 y 表示对象的属性。在 { x: AA, y: BB } 这一行，x 和 y 也代表对象属性。

还记得前面指出过 { x, .. } 是省略了 x: 部分吗？在这两行里，如果去掉代码中的 x: 和 y: 这两部分，只剩下 aa, bb 和 AA, BB，其效果就是——只是概念上，而不是实际上——从 aa 赋值给 AA，从 bb 赋值给 BB。

所以，对称性可能会帮助解释为什么这个 ES6 特性的语法模式有意进行了反转。

我更喜欢解构赋值的语法是 { AA: x , BB: y }，因为这样两种用法都会保留人们更熟悉的 target: souce 模式的一致性。唉，我得训练自己的大脑习惯这个反转，部分读者肯定也是这样。

## 2.4.2 不只是声明

现在我们已经在 var 声明中应用了解构赋值（当然，也可以使用 let 和 const），但是解构是一个通用的赋值操作，不只是声明。

考虑：

```
var a, b, c, x, y, z;

[a,b,c] = foo();
( { x, y, z } = bar() );

console.log( a, b, c );                 // 1 2 3
console.log( x, y, z );                 // 4 5 6
```

这些变量可能是已经声明的，这样的话解构就只用于赋值，就像这里我们看到的。

特别对于对象解构形式来说，如果省略了 var/let/const 声明符，就必须把整个赋值表达式用 ( ) 括起来。因为如果不这样做，语句左侧的 {..} 作为语句中的第一个元素就会被当作是一个块语句而不是一个对象。

实际上，赋值表达式 (a、y 等 ) 并不必须是变量标识符。任何合法的赋值表达式都可以。举例来说：

```
var o = {};

[o.a, o.b, o.c] = foo();
( { x: o.x, y: o.y, z: o.z } = bar() );

console.log( o.a, o.b, o.c );       // 1 2 3
console.log( o.x, o.y, o.z );       // 4 5 6
```

甚至可以在解构中使用计算出的属性表达式。考虑：

```
var which = "x",
    o = {};

( { [which]: o[which] } = bar() );

console.log( o.x );                 // 4
```

[which]: 这一部分是计算出的属性，结果是 x——要从涉及的对象解构出来作为赋值源的属性。o[which] 部分就是一个普通的对象键值引用，等价于 o.x 作为赋值的目标。

可以用一般的赋值来创建对象映射 / 变换，比如：

```
var o1 = { a: 1, b: 2, c: 3 },
    o2 = {};

( { a: o2.x, b: o2.y, c: o2.z } = o1 );

console.log( o2.x, o2.y, o2.z );    // 1 2 3
```

也可以把一个对象映射为一个数组，比如：

```
var o1 = { a: 1, b: 2, c: 3 },
    a2 = [];

( { a: a2[0], b: a2[1], c: a2[2] } = o1 );

console.log( a2 );                  // [1,2,3]
```

或者反过来：

```
var a1 = [ 1, 2, 3 ],
    o2 = {};

[ o2.a, o2.b, o2.c ] = a1;

console.log( o2.a, o2.b, o2.c );    // 1 2 3
```

还可以把一个数组重排序到另一个：

```
var a1 = [ 1, 2, 3 ],
    a2 = [];

[ a2[2], a2[0], a2[1] ] = a1;

console.log( a2 );                  // [2,3,1]
```

甚至可以不用临时变量解决“交换两个变量”这个经典问题：

```
var x = 10, y = 20;

[ y, x ] = [ x, y ];

console.log( x, y );                // 20 10
```

注意：除非需要把所有的赋值表达式都当作声明，否则不应该在赋值中混入声明。不然会出现语法错误。这也是前面我为什么不得不在 `[ a2[0], .. ] = ...` 解构赋值中把 `var a2 = []` 分离出来。语句 `var [ a2[0], .. ] = ..` 是不合法的，因为 `a2[0]` 不是有效的声明标识符；显然它也不会隐式地创建一个 `var a2 = []` 声明。

### 2.4.3 重复赋值

对象解构形式允许多次列出同一个源属性（持有值类型任意）。举例来说：

```
var { a: X, a: Y } = { a: 1 };

X;  // 1
Y;  // 1
```

这也意味着可以解构子对象 / 数组属性，同时捕获子对象 / 类的值本身。考虑：

```
var { a: { x: X, x: Y }, a } = { a: { x: 1 } };

X;  // 1
Y;  // 1
a;  // { x: 1 }

( { a: X, a: Y, a: [ Z ] } = { a: [ 1 ] } );

X.push( 2 );
Y[0] = 10;

X;  // [10,2]
Y;  // [10,2]
Z;  // 1
```

关于解构还有一点需要注意：可能你会忍不住要在同一行中列出所有的解构赋值，就像目前为止我们给出的所有示例一样。但是，更好的思路是把解构赋值模式分散在多行中，并

使用适当的缩进——就像使用 JSON 或者对象字面值时一样——这是为了可读性。

```
// 令人费解：
var { a: { b: [ c, d ], e: { f } }, g } = obj;

// 更好的版本：
var {
     a: {
        b: [ c, d ],
        e: { f }
    },
    g
} = obj;
```

记住：解构的目的不只是为了打字更少，而是为了可读性更强。

**解构赋值表达式**

对象或者数组解构的赋值表达式的完成值是所有右侧对象 / 数组的值。考虑：

```
var o = { a:1, b:2, c:3 },
    a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );         // 1 2 3
p === o;                        // true
```

在前面的代码中，p 赋值为对象 o 的引用，而不是 a、b 或者 c 的值之一。数组解构也是这样：

```
var o = [1,2,3],
    a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );         // 1 2 3
p === o;                        // true
```

通过持有对象 / 数组的值作为完成值，可以把解构赋值表达式组成链：

```
var o = { a:1, b:2, c:3 },
    p = [4,5,6],

    a, b, c, x, y, z;

( {a} = {b,c} = o );
[x,y] = [z] = p;

console.log( a, b, c );         // 1 2 3
console.log( x, y, z );         // 4 5 4
```

## 2.5 太多，太少，刚刚好

对于数组解构赋值和对象解构赋值来说，你不需要把存在的所有值都用来赋值。举例来说：

```
var [,b] = foo();
var { x, z } = bar();

console.log( b, x, z );          // 2 4 6
```

`foo()` 返回的值 `1` 和 `3` 被丢弃了，`bar()` 返回的值 `5` 也是一样。

类似地，如果为比解构 / 分解出来的值更多的值赋值，那么就像期望的一样，多余的值会被赋为 `undefined`：

```
var [,,c,d] = foo();
var { w, z } = bar();

console.log( c, z );               // 3 6
console.log( d, w );               // undefined undefined
```

这个特性是符合前面介绍的“`undefined` 就是缺失”原则的。

本章前面我们介绍了 `...` 运算符，了解到有时候它可以用于把数组中的值散开为独立的值，有时候也可以用于做相反的动作：把一组值组合到一起成为一个数组。

除了在函数声明中的 gather/rest 用法，`...` 也可以执行解构赋值同样的动作。要展示这一点，我们先来回忆一下本章前面的这一段代码：

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

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

这里我们看到 `...a` 把 `a` 展开，因为它出现在 `[ .. ]` 数组值的位置。如果 `...a` 出现在数组解构的位置，就执行集合操作：

```
var a = [2,3,4];
var [ b, ...c ] = a;

console.log( b, c );               // 2 [3,4]
```

`var [ .. ] = a` 解构赋值展开 `a` 用来给 `[ .. ]` 中指定的模式赋值。第一部分名为 `b` 得到 `a` 中的第一个值（`2`）。然后则是 `...c` 收集了其余的值（`3` 和 `4`）赋给一个名为 `c` 的数组。

我们已经看到了 `...` 是如何和数组一起工作的，但是如果是和对象呢？这并非 ES6 的特性，但可以参考第 8 章“ES6 之后”，其中讨论了一个可能的新特性，即如何通过 `...` 展开和集合对象。

### 2.5.1 默认值赋值

使用与前面默认函数参数值类似的 = 语法，解构的两种形式都可以提供一个用来赋值的默认值。

考虑：

```
var [ a = 3, b = 6, c = 9, d = 12 ] = foo();
var { x = 5, y = 10, z = 15, w = 20 } = bar();

console.log( a, b, c, d );          // 1 2 3 12
console.log( x, y, z, w );          // 4 5 6 20
```

可以组合使用默认值赋值和前面介绍的赋值表达式语法。举例来说：

```
var { x, y, z, w: WW = 20 } = bar();

console.log( x, y, z, WW );         // 4 5 6 20
```

如果在解构中使用一个对象或者数组作为默认值的话，注意不要绕晕了自己（或者其他阅读你代码的开发者）。你有可能会写出非常晦涩难懂的代码：

```
var x = 200, y = 300, z = 100;
var o1 = { x: { y: 42 }, z: { y: z } };

( { y: x = { y: y } } = o1 );
( { z: y = { y: z } } = o1 );
( { x: z = { y: x } } = o1 );
```

能从上面的代码中看出 x、y 和 z 最后的值是什么吗？我觉得你可能需要花点时间认真思考一下。这里给出了最终答案。

```
console.log( x.y, y.y, z.y );       // 300 100 42
```

记住这一点：解构很不错也可以很有用，但它也是一把利剑，如果不明智使用的话可能会伤了自己（的大脑）。

### 2.5.2 嵌套解构

如果解构的值中有嵌套的对象或者数组，也可以解构这些嵌套的值：

```
var a1 = [ 1, [2, 3, 4], 5 ];
var o1 = { x: { y: { z: 6 } } };

var [ a, [ b, c, d ], e ] = a1;
var { x: { y: { z: w } } } = o1;

console.log( a, b, c, d, e );       // 1 2 3 4 5
console.log( w );                   // 6
```

可以把嵌套解构当作一种展平对象名字空间的简单方法。举例来说：

```
var App = {
    model: {
        User: function(){ .. }
    }
};

// 不用：
// var User = App.model.User;

var { model: { User } } = App;
```

### 2.5.3 解构参数

你能在下面的代码中找到其中的赋值吗？

```
function foo(x) {
    console.log( x );
}

foo( 42 );
```

这里的赋值某种程度上说是隐藏的：在执行 `foo(42)` 的时候把 `42`（实参）赋给了 `x`（形参）。如果实参 / 形参配对是一个赋值，那么也就是说它是可以解构的，对吗？当然！

考虑下面的参数数组解构：

```
function foo( [ x, y ] ) {
    console.log( x, y );
}

foo( [ 1, 2 ] );                // 1 2
foo( [ 1 ] );                   // 1 undefined
foo( [] );                      // undefined undefined
```

参数的对象解构也是可以的：

```
function foo( { x, y } ) {
    console.log( x, y );
}

foo( { y: 1, x: 2 } );          // 2 1
foo( { y: 42 } );               // undefined 42
foo( {} );                      // undefined undefined
```

这个技术已经接近于命名参数了（一个 JavaScript 渴望已久的特性！），因为这里对象的属性映射到了同名的解构参数。这也意味着我们免费得到了（任意位置上的）可选参数功能；可以看到，省略“参数”`x` 就像我们期望的那样工作。

当然，前面介绍的解构的所有变体都可用于参数解构，包括嵌套解构、默认值，等等。解

构还可以和其他的 ES6 函数参数功能同时使用，比如默认参数值和 rest/gather 参数。

下面是一些细节展示（当然不足以囊括所有可能的变体）：

```
function f1([ x=2, y=3, z ]) { .. }
function f2([ x, y, ...z], w) { .. }
function f3([ x, y, ...z], ...w) { .. }

function f4({ x: X, y }) { .. }
function f5({ x: X = 10, y = 20 }) { .. }
function f6({ x = 10 } = {}, { y } = { y: 10 }) { .. }
```

让我们从前面的代码中找一个例子来解释一下：

```
function f3([ x, y, ...z], ...w) {
    console.log( x, y, z, w );
}

f3( [] );                           // undefined undefined [] []
f3( [1,2,3,4], 5, 6 );              // 1 2 [3,4] [5,6]
```

这里使用了两个 `...` 运算符，它们都用于收集数组（`z` 和 `w`）中的值，当然 `...z` 是从第一个数组参数中剩下的值中收集，而 `...w` 是从主参数去除第一个值后剩下的值中收集。

**1. 解构默认值＋参数默认值**

有一点比较微妙需要指出，也是你应该特别注意的，那就是解构默认值和函数参数默认值之间的差别。举例来说：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                               // 10 10
```

第一眼看上去，我们似乎为参数 `x` 和 `y` 都声明了一个默认值 `10`，虽然是以两种不同的形式。但是，这两种方法在某些情况下的行为是有所不同的，其区别非常微妙。

考虑：

```
f6( {}, {} );                       // 10 undefined
```

稍等，为什么会这样？显然，参数 `x` 不是作为第一个参数对象的同名属性传入得到默认值 `10`。

但是为什么 `y` 值为 `undefined`？作为函数参数默认值的 `{ y: 10 }` 值是一个对象，而不是解构默认值。因此，它只在第二参数没有传入，或者传入 `undefined` 的时候才会生效。

在前面的代码中，我们传入了第二个参数（`{}`），所以没有使用默认值 `{ y: 10 }`，而是在

传入的空对象值 {} 上进行 { y } 解构。

现在，比较一下 { y } = { y: 10 } 和 { x = 10 } = {}。

对于 x 这种形式的用法来说，如果第一个函数参数省略或者是 undefined，就会应用 {} 空对象默认值。然后，在第一个参数位置传入的任何值——或者是默认 {} 或者是你传入的任何值——都使用 { x = 10 } 来解构，这会检查是否有 x 属性，如果没有（或者 undefined），就会为名为 x 的参数应用默认值 10。

缓口气。回头把上面几段重读一遍。我们通过代码来复习一下：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                                   // 10 10
f6( undefined, undefined );             // 10 10
f6( {}, undefined );                    // 10 10

f6( {}, {} );                           // 10 undefined
f6( undefined, {} );                    // 10 undefined

f6( { x: 2 }, { y: 3 } );               // 2 3
```

看起来 x 参数的默认值特性可能比 y 的情况更符合期望，也更合理一些。因此，理解为什么 { x = 10 } = {} 形式与 { y } = { y: 10} 形式有所区别以及如何进行区别是很重要的。

如果还是有点模糊的话，那么回头再读一遍，然后自己试验一下。花些时间把这个微妙的知识点搞清楚，将来你会感谢自己现在所做的一切的。

#### 2. 嵌套默认：解构并重组

尽管一开始看上去可能很难掌握，这里出现了一个很有趣的为嵌套对象属性设置默认值的技巧：使用对象解构以及我称之为**重组**（restructuring）的技术。

考虑在一个嵌套对象结构内的一组默认值，就像下面这样：

```
// 来自于:
// http://es-discourse.com/t/partial-default-arguments/120/7

var defaults = {
    options: {
        remove: true,
        enable: false,
        instance: {}
    },
    log: {
        warn: true,
        error: true
    }
};
```