考虑：

```
var funcs = [];

for (let i = 0; i < 5; i++) {
    funcs.push( function(){
        console.log( i );
    } );
}

funcs[3]();    // 3
```

for 循环头部的 let i 不只为 for 循环本身声明了一个 i，而是为循环的每一次迭代都重新声明了一个新的 i。这意味着 loop 迭代内部创建的闭包封闭的是每次迭代中的变量，就像期望的那样。

如果试验同样的代码，只把 var i 放在 for 循环头部，得到的结果就会是 5 而不是 3，因为在外层作用域中只有一个 i，这个 i 被封闭进去，而不是每个迭代的函数会封闭一个新的 i。

还可以通过下面更详细一点的代码完成同样的事情：

```
var funcs = [];

for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3]();    // 3
```

这里我们强制在每个迭代内部创建了一个新的 j，然后闭包的工作方式是一样的。我更喜欢前面一种形式；这个特殊的额外功能是我为什么支持 for (let ..) .. 形式的原因。可能有人认为这种形式太**隐式**（隐晦），但我认为这已经足够**明晰**，也足够有用了。

let 放在 for .. in 和 for .. of 循环中也是一样的（参见 2.9 节）。

2.1.2 const 声明

还有一个块作用域声明形式需要了解：const，用于创建**常量**。

常量到底是什么？它是一个设定了初始值之后就只读的变量。考虑：

```
{
    const a = 2;
    console.log( a );      // 2
```

```
    a = 3;      // TypeError!
}
```

这个变量的值在声明时设定之后就不允许改变。const 声明必须要有显式的初始化。如果需要一个值为 undefined 的**常量**，就要声明 const a = undefined。

常量不是对这个值本身的限制，而是对赋值的那个变量的限制。换句话说，这个值并没有因为 const 被锁定或者不可变，只是赋值本身不可变。如果这个值是复杂值，比如对象或者数组，其内容仍然是可以修改的。

```
{
    const a = [1,2,3];
    a.push( 4 );
    console.log( a );     // [1,2,3,4]

    a = 42;               // TypeError!
}
```

变量 a 并不持有一个常量数组；相反地，它持有一个指向数组的常量引用。数组本身是可以随意改变的。

将一个对象或数组作为常量赋值，意味着这个值在这个常量的词法作用域结束之前不会被垃圾回收，因为指向这个值的引用没有清除。这可能是你想要的，但是如果不想出现这样的情形的话则需要小心。

本质上说，const 声明强化了多年以来我们通过代码风格表达的信号，其中我们把变量名声明为全大写字母，并将其赋值为某个字面值，小心谨慎地不去改变这个值。var 赋值没有什么强制措施，但现在有了 const 赋值，可以帮助我们发现不想出现的改变。

const 可以用在 for、for..in 以及 for..of 循环的变量声明中（参见 2.9 节）。但如果想要重新赋值就会抛出错误，比如 for 循环中常用的 i++。

是否使用 const

有一些传言认为，JavaScript 引擎在某些情况下可以对 const 进行比 let 和 var 更激进的优化。理论上说，引擎更容易了解这个变量的值 / 类型永远不会改变，那么它就可以取消某些可能的追踪。

不管这里 const 是否真的有好处，还是这只是我们自己的幻想和期望，更重要的决定因素为是否需要常量性。记住：源码的一个重要功能是通过清晰的交流表明你的意图，不只是对你自己，也是对未来的你和其他合作者。

有些开发者倾向于一开始就把所有变量都声明为 const，然后如果必须在代码中修改它就改为 let。这个思路很有趣，但是并不确定它是否真正提高代码的可读性和可理解性。

这并不像许多人相信的那样是真正的**保护**，因为任何后续的开发者想要修改一个 const 值只需要盲目地把 const 声明改成 let 即可。最好的情况下，它避免了意外的修改。但是再次说明，除了我们的直觉和感觉，这里没有客观且清晰地说明“意外”是什么或者要防止什么。类似的思路也存在于类型强制转换的情况。

我的建议是：要避免可能令人迷惑的代码，只对你有意表明不会改变的变量使用 const。换句话说，不要依赖于 const 来规范代码行为，而是在意图清晰的时候，把它作为一个表明意图的工具。

2.1.3 块作用域函数

从 ES6 开始，块内声明的函数，其作用域在这个块内。在 ES6 之前，规范并没有要求这一点，但是许多实现就是这么做的。所以现在是规范与现实保持一致了。

考虑：

```
{
    foo();                      // 可以这么做!

    function foo() {
        // ..
    }
}

foo();                          // ReferenceError
```

foo() 函数声明在 { .. } 块内部，ES6 支持块作用域。所以在块外不可用。但是还要注意到它是在块内“提升”了，与 let 声明相反，后者会遇到前面介绍的 TDZ 错误陷阱。

如果编写了和前面类似的代码，并依赖于旧有的非块作用域行为，那么函数声明的块作用域就可能会引发问题。

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();    // ??
```

在前 ES6 环境中，不管 something 的值是什么，foo() 都会打印出 "2"，因为两个函数声明

都被提升到了块外，第二个总是会胜出。

而在 ES6 中，最后一行会抛出一个 ReferenceError。

2.2 spread/rest

ES6 引入了一个新的运算符 ...，通常称为 spread 或 rest（展开或收集）运算符，取决于它在哪 / 如何使用。我们来看一下：

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

foo( ...[1,2,3] );              // 1 2 3
```

当 ... 用在数组之前时（实际上是任何 iterable，我们将在第 3 章中介绍），它会把这个变量“展开”为各个独立的值。

我们通常看到的是前面代码片段中的使用方式，即把一个数组展开为一组函数调用的参数。在这种用法中，... 为我们提供了可以替代 apply(..) 方法的一个简单的语法形式，在前 ES6 中我们常常这样写：

```
foo.apply( null, [1,2,3] );     // 1 2 3
```

然而，... 也可以在其他上下文中用来展开 / 扩展一个值，比如在另一个数组声明中：

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

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

在这种用法中，... 基本上代替了 concat(..)，这里的行为就像是 [1].concat(a, [5])。

... 的另外一种常见用法基本上可以被看作反向的行为；与把一个值展开不同，... 把一系列值收集到一起成为一个数组。考虑：

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

foo( 1, 2, 3, 4, 5 );           // 1 2 [3,4,5]
```

在这段代码中，...z 基本上是在说：“把**剩下的**参数（如果有的话）收集到一起组成一个名为 z 的数组。”因为 x 赋值为 1，y 赋值为 2，所以其余的参数 3、4 和 5 被收集到数组 z 中。

当然，如果没有命名参数的话，... 就会收集所有的参数：

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

foo( 1, 2, 3, 4, 5);          // [1,2,3,4,5]
```

foo(..) 函数声明中的 ...args 通常称为“rest 参数”，因为这里是在收集其余的参数。我喜欢用“收集”这个词，因为这更好地描述了它的行为而不是它的内容。

这种用法最好的一点是，它为弃用很久的 arguments 数组——实际上它并不是真正的数组，而是类似数组的对象——提供了一个非常可靠的替代形式。因为 args（或者随便你给它起什么名字——很多人喜欢用 r 或 rest）是一个真正的数组，前 ES6 中有很多技巧用来把 arguments 转变为某种我们可以当作数组来使用的东西，现在我们可以摆脱这些愚蠢的技巧了。

考虑：

```
// 按照新的ES6的行为方式实现
function foo(...args) {
    // args已经是一个真正的数组

    // 丢弃args中第一个元素
    args.shift();

    // 把整个args作为参数传给console.log(..)
    console.log( ...args );
 }

// 按照前ES6的老派行为方式实现
function bar() {
    // 把arguments转换为一个真正的数组
    var args = Array.prototype.slice.call( arguments );

    // 在尾端添加几个元素
    args.push( 4, 5 );

    // 过滤掉奇数
    args = args.filter( function(v){
        return v % 2 == 0;
    } );

    // 把整个args作为参数传给foo(..)
    foo.apply( null, args );
}

bar( 0, 1, 2, 3 );                  // 2 4
```

函数 foo(..) 声明中的 ...args 收集参数，console.log(..) 调用中的 ...args 将其展开。

这里很好地展示了运算符 ... 对称而又相反的用法。

除了在函数声明中使用 ...，还有一种情况是 ... 被用于收集值，我们将在 2.5 节中介绍。

2.3 默认参数值

可能 JavaScript 最常见的一个技巧就是关于设定函数参数默认值的。多年以来我们实现这一点的方式是这样的，看起来应该很熟悉：

```
function foo(x,y) {
    x = x || 11;
    y = y || 31;

    console.log( x + y );
}

foo();              // 42
foo( 5, 6 );        // 11
foo( 5 );           // 36
foo( null, 6 );     // 17
```

当然，如果你之前使用过这个模式，就会知道它很有用，但同时又有点危险，比如，如果对于一个参数你需要能够传入被认为是 falsy（假）的值。考虑：

```
foo( 0, 42 );       // 53 <--哎呀，并非42
```

为什么？因为这里 0 为假，所以 x || 11 结果为 11，而不是直接传入的 0。

要修正这个问题，有些人会选择增加更多的检查，就像下面这样：

```
function foo(x,y) {
    x = (x !== undefined) ? x : 11;
    y = (y !== undefined) ? y : 31;

    console.log( x + y );
}

foo( 0, 42 );           // 42
foo( undefined, 6 );    // 17
```

当然，这意味着除了 undefined 之外的任何值都可以直接传入。然而，undefined 会表达“我没有传入信息”这样的信息。除非你确实需要能够传入 undefined，它就工作的很好。

这种情况下，可以通过它并不存在于数组 arguments 之中来确定这个参数是被省略的，可能就像下面这样：

```
function foo(x,y) {
    x = (0 in arguments) ? x : 11;
    y = (1 in arguments) ? y : 31;

    console.log( x + y );
}

foo( 5 );               // 36
foo( 5, undefined );    // NaN
```

但是如果你不能传递任何值（甚至 undefined 也不行）来表明“我省略了这个参数”，那么如何省略第一个参数 x 呢？

foo(,5) 很吸引人，但这是不合法的语法。foo.apply(null, [,5]) 看来可以实现这个技巧，但是这里 apply(..) 的诡异实现意味着这些参数被当作了 [undefined,5]，这当然不是一个省略。

如果继续深入的话，就会发现你只能通过传入比“期望”更少的参数来省略最后的若干参数（例如，右侧的），而无法省略位于参数列表中间或者起始处的参数。

这里应用了一个很重要的需要记住的 JavaScrpt 设计原则：undefined 意味着**缺失**。也就是说，undefined 和缺失是无法区别的，至少对于函数参数来说是如此。

在 JavaScript 的其他一些地方上面提到的设计原则是不成立的，这时候会引起混淆，比如对于有空槽的数组。参见本系列《你不知道的 JavaScript（中卷）》第一部分可以获取更多信息。

了解了所有这些之后，现在我们可以讨论 ES6 新增的一个有用的语法来改进为缺失参数赋默认值的流程。

```
function foo(x = 11, y = 31) {
    console.log( x + y );
}

foo();                  // 42
foo( 5, 6 );            // 11
foo( 0, 42 );           // 42

foo( 5 );               // 36
foo( 5, undefined );    // 36 <-- 丢了undefined
foo( 5, null );         // 5  <-- null被强制转换为0

foo( undefined, 6 );    // 17 <-- 丢了undefined
foo( null, 6 );         // 6  <-- null被强制转换为0
```

注意这些结果，以及它们和前面的方法之间的微妙区别和相似之处。

在函数声明中的 `x = 11` 更像是 `x !== undefined ? x : 11` 而不是常见技巧 `x || 11`，所以在把前 ES6 代码转换为 ES6 默认参数值语法的时候要格外小心。

rest/gather 参数（参见 2.2 节）不能有默认值。所以，尽管 `function foo(...vals=[1,2,3]) {` 这样的用法可能看起来很诱人，但是它并非合法的语法。如果需要的话还是得继续手动提供这种逻辑。

默认值表达式

函数默认值可以不只是像 `31` 这样的简单值；它们可以是任意合法表达式，甚至是函数调用。

```
function bar(val) {
    console.log( "bar called!" );
    return y + val;
}

function foo(x = y + 3, z = bar( x )) {
    console.log( x, z );
}

var y = 5;
foo();                      // "bar called"
                            // 8 13
foo( 10 );                  // "bar called"
                            // 10 15
y = 6;
foo( undefined, 10 );       // 9 10
```

可以看到，默认值表达式是惰性求值的，这意味着它们只在需要的时候运行——也就是说，是在参数的值省略或者为 `undefined` 的时候。

这里有一个微妙的细节，注意函数声明中形式参数是在它们自己的作用域中（可以把它看作是就在函数声明包裹的 `(  ..  )` 的作用域中），而不是在函数体作用域中。这意味着在默认值表达式中的标识符引用首先匹配到形式参数作用域，然后才会搜索外层作用域。参见本系列《你不知道的 JavaScript（上卷）》第一部分可以获取更多信息。

考虑：

```
var w = 1, z = 2;

function foo( x = w + 1, y = x + 1, z = z + 1 ) {
    console.log( x, y, z );
}

foo();                  // ReferenceError
```

`w + 1` 默认值表达式中的 `w` 在形式参数列表作用域中寻找 `w`，但是没有找到，所以就使用外层作用域的 `w`。接下来，`x + 1` 默认值表达式中的 `x` 找到了形式参数作用域中的 `x`，很幸运这里 `x` 已经初始化了，所以对 `y` 的赋值可以正常工作。

但是，`z + 1` 中的 `z` 发现 `z` 是一个此刻还没初始化的参数变量，所以它永远不会试图从外层作用域寻找 `z`。

正如我们在 2.1.1 节中提到的，ES6 引入了 TDZ，它防止变量在未初始化的状态下被访问。因此，`z + 1` 默认值表达式会抛出一个 TDZ`ReferenceError` 错误。

默认值表达式甚至可以是 inline 函数表达式调用——一般称为立即调用函数表达式（IIFE），但这对于代码明晰性没什么好处。

```
function foo( x =
    (function(v){ return v + 11; })( 31 )
) {
    console.log( x );
}

foo();          // 42
```

IIFE（或者其他任何执行 inline 函数表达式）适用于默认值表达式的情况是很少见的。如果你发现自己需要这么做，那么一定要后退一步重新思考一下！

如果这个 IIFE 试图访问标识符 `x`，并且没有声明自己的 `x` 的话，也会引发 TDZ 错误，就像前面讨论的一样。

前面代码中的默认值表达式是一个 IIFE，因为这是一个通过 `(31)` 立即在线执行的函数。如果没有这一部分，那么赋给 `x` 的默认值就会是一个函数引用本身，可能就像默认回调那样。这个模式在某些情况下可能是很有用的，比如：

```
function ajax(url, cb = function(){}) {
    // ..
}

ajax( "http://some.url.1" );
```

在这个例子中，我们需要在没有指定其他函数情况下的默认 `cb` 是一个没有操作的空函数调用。这个函数表达式就是一个函数引用，而不是函数调用本身（后面没有调用形式 `()`），这实现了我们的目标。

从 JavaScript 的早期开始，就有一个不为人知但是很有用的技巧可以使用：`Function.prototype` 本身就是一个没有操作的空函数。所以，这个声明可以是 `cb = Function.prototype`，这样就省去了在线函数表达式的创建过程。

2.4 解构

ES6 引入了一个新的语法特性，名为**解构**（destructuring），把这个功能看作是一个**结构化赋值**（structured assignment）方法，可能会容易理解一些。考虑：

```
function foo() {
    return [1,2,3];
}

var tmp = foo(),
    a = tmp[0], b = tmp[1], c = tmp[2];

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

可以看到，我们构造了一个手动赋值，把 `foo()` 返回数组中的值赋给独立变量 `a`、`b` 和 `c`，为了实现这一点，我们（不幸地）需要一个临时变量 `tmp`。

类似地，对于对象可以像下面这么做：

```
function bar() {
    return {
        x: 4,
        y: 5,
        z: 6
    };
}

var tmp = bar(),
    x = tmp.x, y = tmp.y, z = tmp.z;

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

`tmp.x` 属性值赋给了变量 `x`，同样地，`tmp.y` 赋给了 `y`，`tmp.z` 赋给了 `z`。

可以把将数组或者对象属性中带索引的值手动赋值看作结构化赋值。ES6 为解构新增了一个专门语法，专用于**数组解构**和**对象解构**。这个语法消除了前面代码中对临时变量 `tmp` 的需求，使代码简洁很多。考虑：

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

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

你可能更习惯 `[a,b,c]` 这样的语法作为要赋的值出现在赋值操作符 `=` 的右侧。

解构语法对称地翻转了这个模式，于是赋值符 `=` 左侧的 `[a,b,c]` 被当作某种“模式”，用来把右侧数组值解构赋值到独立的变量中。