```
    console.log( "x:", x );
}

function bar() {
    x++;
}

foo();                  // x: 3
```

在这个例子中，我们确信 bar() 会在 x++ 和 console.log(x) 之间运行。但是，如果 bar() 并不在那里会怎样呢？显然结果就会是 2，而不是 3。

现在动脑筋想一下。如果 bar() 并不在那儿，但出于某种原因它仍然可以在 x++ 和 console.log(x) 语句之间运行，这又会怎样呢？这如何才会成为可能呢？

如果是在抢占式多线程语言中，从本质上说，这是可能发生的，bar() 可以在两个语句之间打断并运行。但 JavaScript 并不是抢占式的，（目前）也不是多线程的。然而，如果 foo() 自身可以通过某种形式在代码的这个位置指示暂停的话，那就仍然可以以一种合作式的方式实现这样的中断（并发）。

这里我之所以使用了“合作式的”一词，不只是因为这与经典并发术语之间的关联（参见第 1 章）；还因为你将会在下一段代码中看到的，ES6 代码中指示暂停点的语法是 yield，这也礼貌地表达了一种合作式的控制放弃。

下面是实现这样的合作式并发的 ES6 代码：

```
var x = 1;

function *foo() {
    x++;
    yield; // 暂停!
    console.log( "x:", x );
}

function bar() {
    x++;
}
```

很可能你看到的其他多数 JavaScript 文档和代码中的生成器声明格式都是 function* foo() { .. }，而不是我这里使用的 function *foo() { .. }：唯一区别是 * 位置的风格不同。这两种形式在功能和语法上都是等同的，还有一种是 function*foo(){ .. }（没有空格）也一样。两种风格，各有优缺，但总体上我比较喜欢 function *foo.. 的形式，因为这样在使用 *foo() 来引用生成器的时候就会比较一致。如果只用 foo() 的形式，你就不会清楚知道我指的是生成器还是常规函数。这完全是一个风格偏好问题。

现在，我们要如何运行前面的代码片段，使得 bar() 在 *foo() 内部的 yield 处执行呢？

```
// 构造一个迭代器it来控制这个生成器
var it = foo();

// 这里启动foo()!
it.next();
x;                  // 2
bar();
x;                  // 3
it.next();          // x: 3
```

好吧，这两段代码中有很多新知识，可能会让人迷惑，所以这里有很多东西需要学习。在解释 ES6 生成器的不同机制和语法之前，我们先来看看运行过程。

(1) it = foo() 运算并没有执行生成器 *foo()，而只是构造了一个迭代器（iterator），这个迭代器会控制它的执行。后面会介绍迭代器。
(2) 第一个 it.next() 启动了生成器 *foo()，并运行了 *foo() 第一行的 x++。
(3) *foo() 在 yield 语句处暂停，在这一点上第一个 it.next() 调用结束。此时 *foo() 仍在运行并且是活跃的，但处于暂停状态。
(4) 我们查看 x 的值，此时为 2。
(5) 我们调用 bar()，它通过 x++ 再次递增 x。
(6) 我们再次查看 x 的值，此时为 3。
(7) 最后的 it.next() 调用从暂停处恢复了生成器 *foo() 的执行，并运行 console.log(..) 语句，这条语句使用当前 x 的值 3。

显然，foo() 启动了，但是没有完整运行，它在 yield 处暂停了。后面恢复了 foo() 并让它运行到结束，但这不是必需的。

因此，生成器就是一类特殊的函数，可以一次或多次启动和停止，并不一定非得要完成。尽管现在还不是特别清楚它的强大之处，但随着对本章后续内容的深入学习，我们会看到它将成为用于构建以生成器作为异步流程控制的代码模式的基础构件之一。

4.1.1 输入和输出

生成器函数是一个特殊的函数，具有前面我们展示的新的执行模式。但是，它仍然是一个函数，这意味着它仍然有一些基本的特性没有改变。比如，它仍然可以接受参数（即输入），也能够返回值（即输出）。

```
function *foo(x,y) {
    return x * y;
}

var it = foo( 6, 7 );
```

```
var res = it.next();

res.value;      // 42
```

我们向 *foo(..) 传入实参 6 和 7 分别作为参数 x 和 y。*foo(..) 向调用代码返回 42。

现在我们可以看到生成器和普通函数在调用上的一个区别。显然 foo(6,7) 看起来很熟悉。但难以理解的是，生成器 *foo(..) 并没有像普通函数一样实际运行。

事实上，我们只是创建了一个迭代器对象，把它赋给了一个变量 it，用于控制生成器 *foo(..)。然后调用 it.next()，指示生成器 *foo(..) 从当前位置开始继续运行，停在下一个 yield 处或者直到生成器结束。

这个 next(..) 调用的结果是一个对象，它有一个 value 属性，持有从 *foo(..) 返回的值（如果有的话）。换句话说，yield 会导致生成器在执行过程中发送出一个值，这有点类似于中间的 return。

目前还不清楚为什么需要这一整个间接迭代器对象来控制生成器。会清楚的，我保证。

1. 迭代消息传递

除了能够接受参数并提供返回值之外，生成器甚至提供了更强大更引人注目的内建消息输入输出能力，通过 yield 和 next(..) 实现。

考虑：

```
function *foo(x) {
    var y = x * (yield);
    return y;
}

var it = foo( 6 );

// 启动foo(..)
it.next();

var res = it.next( 7 );

res.value;      // 42
```

首先，传入 6 作为参数 x。然后调用 it.next()，这会启动 *foo(..)。

在 *foo(..) 内部，开始执行语句 var y = x ..，但随后就遇到了一个 yield 表达式。它就会在这一点上暂停 *foo(..)（在赋值语句中间！），并在本质上要求调用代码为 yield 表达式提供一个结果值。接下来，调用 it.next(7)，这一句把值 7 传回作为被暂停的 yield 表达式的结果。

所以，这时赋值语句实际上就是 `var y = 6 * 7`。现在，`return y` 返回值 42 作为调用 `it.next( 7 )` 的结果。

注意，这里有一点非常重要，但即使对于有经验的 JavaScript 开发者也很有迷惑性：根据你的视角不同，`yield` 和 `next(..)` 调用有一个不匹配。一般来说，需要的 `next(..)` 调用要比 `yield` 语句多一个，前面的代码片段有一个 `yield` 和两个 `next(..)` 调用。

为什么会有这个不匹配？

因为第一个 `next(..)` 总是启动一个生成器，并运行到第一个 `yield` 处。不过，是第二个 `next(..)` 调用完成第一个被暂停的 `yield` 表达式，第三个 `next(..)` 调用完成第二个 `yield`，以此类推。

2. 两个问题的故事

实际上，你首先考虑的是哪一部分代码将会影响这个不匹配是否被察觉到。

只考虑生成器代码：

```
var y = x * (yield);
return y;
```

第一个 `yield` 基本上是提出了一个问题：“这里我应该插入什么值？”

谁来回答这个问题呢？第一个 `next()` 已经运行，使得生成器启动并运行到此处，所以显然它无法回答这个问题。因此必须由第二个 `next(..)` 调用回答第一个 `yield` 提出的这个问题。

看到不匹配了吗——第二个对第一个？

把视角转化一下：不从生成器的视角看这个问题，而是从迭代器的角度。

为了恰当阐述这个视角，我们还需要解释一下：消息是双向传递的——`yield..` 作为一个表达式可以发出消息响应 `next(..)` 调用，`next(..)` 也可以向暂停的 `yield` 表达式发送值。考虑下面这段稍稍调整过的代码：

```
function *foo(x) {
    var y = x * (yield "Hello");     // <-- yield一个值!
    return y;
}

var it = foo( 6 );

var res = it.next();    // 第一个next(),并不传入任何东西
res.value;              // "Hello"

res = it.next( 7 );     // 向等待的yield传入7
res.value;              // 42
```

yield .. 和 next(..) 这一对组合起来，在生成器的执行过程中构成了一个双向消息传递系统。

那么只看下面这一段迭代器代码：

```
var res = it.next();      // 第一个next(),并不传入任何东西
res.value;                // "Hello"

res = it.next( 7 );       // 向等待的yield传入7
res.value;                // 42
```

我们并没有向第一个 next() 调用发送值，这是有意为之。只有暂停的 yield 才能接受这样一个通过 next(..) 传递的值，而在生成器的起始处我们调用第一个 next() 时，还没有暂停的 yield 来接受这样一个值。规范和所有兼容浏览器都会默默丢弃传递给第一个 next() 的任何东西。传值过去仍然不是一个好思路，因为你创建了沉默的无效代码，这会让人迷惑。因此，启动生成器时一定要用不带参数的 next()。

第一个 next() 调用（没有参数的）基本上就是在提出一个问题："生成器 *foo(..) 要给我的下一个值是什么"。谁来回答这个问题呢？第一个 yield "hello" 表达式。

看见了吗？这里没有不匹配。

根据你认为提出问题的是谁，yield 和 next(..) 调用之间要么有不匹配，要么没有。

但是，稍等！与 yield 语句的数量相比，还是多出了一个额外的 next()。所以，最后一个 it.next(7) 调用再次提出了这样的问题：生成器将要产生的下一个值是什么。但是，再没有 yield 语句来回答这个问题了，是不是？那么谁来回答呢？

return 语句回答这个问题！

如果你的生成器中没有 return 的话——在生成器中和在普通函数中一样，return 当然不是必需的——总有一个假定的 / 隐式的 return;（也就是 return undefined;），它会在默认情况下回答最后的 it.next(7) 调用提出的问题。

这样的提问和回答是非常强大的：通过 yield 和 next(..) 建立的双向消息传递。但目前还不清楚这些机制是如何与异步流程控制联系到一起的。会清楚的！

4.1.2 多个迭代器

从语法使用的方面来看，通过一个迭代器控制生成器的时候，似乎是在控制声明的生成器函数本身。但有一个细微之处很容易忽略：每次构建一个迭代器，实际上就隐式构建了生成器的一个实例，通过这个迭代器来控制的是这个生成器实例。

同一个生成器的多个实例可以同时运行，它们甚至可以彼此交互：

```
function *foo() {
    var x = yield 2;
    z++;
    var y = yield (x * z);
    console.log( x, y, z );
}

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value;           // 2 <-- yield 2
var val2 = it2.next().value;           // 2 <-- yield 2

val1 = it1.next( val2 * 10 ).value;    // 40   <-- x:20,  z:2
val2 = it2.next( val1 * 5 ).value;     // 600  <-- x:200, z:3

it1.next( val2 / 2 );                  // y:300
                                       // 20 300 3
it2.next( val1 / 4 );                  // y:10
                                       // 200 10 3
```

同一个生成器的多个实例并发运行的最常用处并不是这样的交互，而是生成器在没有输入的情况下，可能从某个独立连接的资源产生自己的值。下一节中我们会详细介绍值产生。

我们简单梳理一下执行流程。

(1) `*foo()` 的两个实例同时启动，两个 `next()` 分别从 `yield 2` 语句得到值 2。

(2) `val2 * 10` 也就是 `2 * 10`，发送到第一个生成器实例 `it1`，因此 `x` 得到值 `20`。`z` 从 1 增加到 2，然后 `20 * 2` 通过 `yield` 发出，将 `val1` 设置为 `40`。

(3) `val1 * 5` 也就是 `40 * 5`，发送到第二个生成器实例 `it2`，因此 `x` 得到值 `200`。`z` 再次从 2 递增到 3，然后 `200 * 3` 通过 `yield` 发出，将 `val2` 设置为 `600`。

(4) `val2 / 2` 也就是 `600 / 2`，发送到第一个生成器实例 `it1`，因此 `y` 得到值 `300`，然后打印出 `x y z` 的值分别是 `20 300 3`。

(5) `val1 / 4` 也就是 `40 / 4`，发送到第二个生成器实例 `it2`，因此 `y` 得到值 `10`，然后打印出 `x y z` 的值分别为 `200 10 3`。

在脑海中运行一遍这个例子很有趣。理清楚了吗？

交替执行

回想一下 1.3 节中关于完整运行的这个场景：

```
var a = 1;
var b = 2;
```

```
function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}
```

如果是普通的 JavaScript 函数的话，显然，要么是 `foo()` 首先运行完毕，要么是 `bar()` 首先运行完毕，但 `foo()` 和 `bar()` 的语句不能交替执行。所以，前面的程序只有两种可能的输出。

但是，使用生成器的话，交替执行（甚至在语句当中！）显然是可能的：

```
var a = 1;
var b = 2;

function *foo() {
    a++;
    yield;
    b = b * a;
    a = (yield b) + 3;
}

function *bar() {
    b--;
    yield;
    a = (yield 8) + b;
    b = a * (yield 2);
}
```

根据迭代器控制的 `*foo()` 和 `*bar()` 调用的相对顺序不同，前面的程序可能会产生多种不同的结果。换句话说，通过两个生成器在共享的相同变量上的迭代交替执行，我们实际上可以（以某种模拟的方式）印证第 1 章讨论的理论上的多线程竞态条件环境。

首先，来构建一个名为 `step(..)` 的辅助函数，用于控制迭代器：

```
function step(gen) {
    var it = gen();
    var last;

    return function() {
        // 不管yield出来的是什么,下一次都把它原样传回去!
        last = it.next( last ).value;
    };
}
```

step(..) 初始化了一个生成器来创建迭代器 it，然后返回一个函数，这个函数被调用的时候会将迭代器向前迭代一步。另外，前面的 yield 发出的值会在下一步发送回去。于是，yield 8 就是 8，而 yield b 就是 b（yield 发出时的值）。

现在，只是为了好玩，我们来试验一下交替运行 *foo() 和 *bar() 代码块的效果。我们从乏味的基本情况开始，确保 *foo() 在 *bar() 之前完全结束（和第 1 章中做的一样）：

```
// 确保重新设置a和b
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// 首次运行*foo()
s1();
s1();
s1();

// 现在运行*bar()
s2();
s2();
s2();
s2();

console.log( a, b );    // 11 22
```

最后的结果是 11 和 22，和第 1 章中的版本一样。现在交替执行顺序，看看 a 和 b 的值是如何改变的：

```
// 确保重新设置a和b
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();       // b--;
s2();       // yield 8
s1();       // a++;
s2();       // a = 8 + b;
            // yield 2
s1();       // b = b * a;
            // yield b
s1();       // a = b + 3;
s2();       // b = a * 2;
```

在告诉你结果之前，你能推断出前面的程序运行后 a 和 b 的值吗？不要作弊！

```
console.log( a, b );      // 12 18
```

作为留给大家的练习，请试着重新安排 s1() 和 s2() 的调用顺序，看看还能够得到多少种结果组合。不要忘了，你总是需要 3 次 s1() 调用和 4 次 s2() 调用。回忆一下前面关于 next() 和 yield 匹配的讨论，想想为什么。

当然，你基本不可能故意创建让人迷惑到这种程度的交替运行实际代码，因为这给理解代码带来了极大的难度。但这个练习很有趣，对于理解多个生成器如何在共享的作用域上并发运行也有指导意义，因为这个功能有很多用武之地。

我们将在 4.6 节中更深入讨论生成器并发。

4.2 生成器产生值

在前面一节中，我们提到生成器的一种有趣用法是作为一种产生值的方式。这并不是本章的重点，但是如果不介绍一些基础的话，就会缺乏完整性了，特别是因为这正是“生成器”这个名称最初的使用场景。

下面要偏一下题，先介绍一点*迭代器*，不过我们还会回来介绍它们与生成器的关系以及如何使用生成器来*生成*值。

4.2.1 生产者与迭代器

假定你要产生一系列值，其中每个值都与前面一个有特定的关系。要实现这一点，需要一个有状态的生产者能够记住其生成的最后一个值。

可以实现一个直接使用函数闭包的版本（参见本系列的《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分），类似如下：

```
var gimmeSomething = (function(){
    var nextVal;

    return function(){
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) +6;
        }

        return nextVal;
    };
})();

gimmeSomething();       // 1
gimmeSomething();       // 9
gimmeSomething();       // 33
gimmeSomething();       // 105
```

这里 nextVal 的计算逻辑已经简化了，但是从概念上说，我们希望直到下一次 gimmeSomething() 调用发生时才计算下一个值（即 nextVal）。否则，一般来说，对更持久化或比起简单数字资源更受限的生产者来说，这可能就是资源泄漏的设计。

生成任意数字序列并不是一个很实际的例子。但如果是想要从数据源生成记录呢？可以采用基本相同的代码。

实际上，这个任务是一个非常通用的设计模式，通常通过迭代器来解决。*迭代器*是一个定义良好的接口，用于从一个生产者一步步得到一系列值。JavaScript 迭代器的接口，与多数语言类似，就是每次想要从生产者得到下一个值的时候调用 next()。

可以为我们的数字序列生成器实现标准的*迭代器*接口：

```
var something = (function(){
    var nextVal;

    return {
        // for..of循环需要
        [Symbol.iterator]: function(){ return this; },

        // 标准迭代器接口方法
        next: function(){
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            return { done:false, value:nextVal };
        }
    };
})();

something.next().value;     // 1
something.next().value;     // 9
something.next().value;     // 33
something.next().value;     // 105
```

我们将在 4.2.2 节解释为什么在这段代码中需要 [Symbol.iterator]: .. 这一部分。从语法上说，这涉及了两个 ES6 特性。首先，[..] 语法被称为*计算属性名*（参见本系列的《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分）。这在对象术语定义中是指，指定一个表达式并用这个表达式的结果作为属性的名称。另外，Symbol.iterator 是 ES6 预定义的特殊 Symbol 值之一（参见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分）。

next() 调用返回一个对象。这个对象有两个属性：done 是一个 boolean 值，标识迭代器的完成状态；value 中放置迭代值。

ES6 还新增了一个 for..of 循环，这意味着可以通过原生循环语法自动迭代标准迭代器：

```
for (var v of something) {
    console.log( v );

    // 不要死循环!
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

因为我们的迭代器 something 总是返回 done:false，因此这个 for..of 循环将永远运行下去，这也就是为什么我们要在里面放一个 break 条件。迭代器永不结束是完全没问题的，但是也有一些情况下，迭代器会在有限的值集合上运行，并最终返回 done:true。

for..of 循环在每次迭代中自动调用 next()，它不会向 next() 传入任何值，并且会在接收到 done:true 之后自动停止。这对于在一组数据上循环很方便。

当然，也可以手工在迭代器上循环，调用 next() 并检查 done:true 条件来确定何时停止循环：

```
for (
    var ret;
    (ret = something.next()) && !ret.done;
) {
    console.log( ret.value );

    // 不要死循环!
    if (ret.value > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

这种手工 for 方法当然要比 ES6 的 for..of 循环语法丑陋，但其优点是，这样就可以在需要时向 next() 传递值。

除了构造自己的迭代器，许多 JavaScript 的内建数据结构（从 ES6 开始），比如 array，也有默认的迭代器：

```
var a = [1,3,5,7,9];

for (var v of a) {
    console.log( v );
}
// 1 3 5 7 9
```

for..of 循环向 a 请求它的迭代器，并自动使用这个迭代器迭代遍历 a 的值。

这里可能看起来像是 ES6 一个奇怪的缺失，不过一般的 object 是故意不像 array 一样有默认的迭代器。这里我们并不会深入探讨其中的缘由。如果你只是想要迭代一个对象的所有属性的话（不需要保证特定的顺序），可以通过 Object.keys(..) 返回一个 array，类似于 for (var k of Object.keys(obj)) { .. 这样使用。这样在一个对象的键值上使用 for..of 循环与 for..in 循环类似，除了 Object.keys(..) 并不包含来自于 [[Prototype]] 链上的属性，而 for..in 则包含（参见本系列的《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分）。

4.2.2 iterable

前面例子中的 something 对象叫作迭代器，因为它的接口中有一个 next() 方法。而与其紧密相关的一个术语是 iterable（可迭代），即指一个包含可以在其值上迭代的迭代器的对象。

从 ES6 开始，从一个 iterable 中提取迭代器的方法是：iterable 必须支持一个函数，其名称是专门的 ES6 符号值 Symbol.iterator。调用这个函数时，它会返回一个迭代器。通常每次调用会返回一个全新的迭代器，虽然这一点并不是必须的。

前面代码片段中的 a 就是一个 iterable。for..of 循环自动调用它的 Symbol.iterator 函数来构建一个迭代器。我们当然也可以手工调用这个函数，然后使用它返回的迭代器：

```
var a = [1,3,5,7,9];

var it = a[Symbol.iterator]();

it.next().value;    // 1
it.next().value;    // 3
it.next().value;    // 5
..
```

前面的代码中列出了定义的 something，你可能已经注意到了这一行：

```
[Symbol.iterator]: function(){ return this; }
```

这段有点令人疑惑的代码是在将 something 的值（迭代器 something 的接口）也构建成为一个 iterable。现在它既是 iterable，也是迭代器。然后我们把 something 传给 for..of 循环：

```
for (var v of something) {
    ..
}
```

`for..of` 循环期望 `something` 是 iterable，于是它寻找并调用它的 `Symbol.iterator` 函数。我们将这个函数定义为就是简单的 `return this`，也就是把自身返回，而 `for..of` 循环并不知情。

4.2.3 生成器迭代器

了解了迭代器的背景，让我们把注意力转回生成器上。可以把生成器看作一个值的生产者，我们通过迭代器接口的 `next()` 调用一次提取出一个值。

所以，严格说来，生成器本身并不是 iterable，尽管非常类似——当你执行一个生成器，就得到了一个迭代器：

```
function *foo(){ .. }

var it = foo();
```

可以通过生成器实现前面的这个 `something` 无限数字序列生产者，类似这样：

```
function *something() {
    var nextVal;

    while (true) {
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        yield nextVal;
    }
}
```

通常在实际的 JavaScript 程序中使用 `while..true` 循环是非常糟糕的主意，至少如果其中没有 `break` 或 `return` 的话是这样，因为它有可能会同步地无限循环，并阻塞和锁住浏览器 UI。但是，如果在生成器中有 `yield` 的话，使用这样的循环就完全没有问题。因为生成器会在每次迭代中暂停，通过 `yield` 返回到主程序或事件循环队列中。简单地说就是：“生成器把 `while..true` 带回了 JavaScript 编程的世界！”

这样就简单明确多了，是不是？因为生成器会在每个 `yield` 处暂停，函数 `*something()` 的状态（作用域）会被保持，即意味着不需要闭包在调用之间保持变量状态。

这段代码不仅更简洁，我们不需要构造自己的迭代器接口，实际上也更合理，因为它更清晰地表达了意图。比如，while..true 循环告诉我们这个生成器就是要永远运行：只要我们一直索要，它就会一直生成值。

现在，可以通过 for..of 循环使用我们雕琢过的新的 *something() 生成器。你可以看到，其工作方式基本是相同的：

```
for (var v of something()) {
    console.log( v );

    // 不要死循环!
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

但是，不要忽略了这段 for (var v of something()) .. ！我们并不是像前面的例子那样把 something 当作一个值来引用，而是调用了 *something() 生成器以得到它的迭代器供 for..of 循环使用。

如果认真思考的话，你也许会从这段生成器与循环的交互中提出两个问题。

- 为什么不能用 for (var v of something) .. ？因为这里的 something 是生成器，并不是 iterable。我们需要调用 something() 来构造一个生产者供 for..of 循环迭代。
- something() 调用产生一个迭代器，但 for..of 循环需要的是一个 iterable，对吧？是的。生成器的迭代器也有一个 Symbol.iterator 函数，基本上这个函数做的就是 return this，和我们前面定义的 iterable something 一样。换句话说，生成器的迭代器也是一个 iterable！

停止生成器

在前面的例子中，看起来似乎 *something() 生成器的迭代器实例在循环中的 break 调用之后就永远留在了挂起状态。

其实有一个隐藏的特性会帮助你管理此事。for..of 循环的“异常结束”（也就是“提前终止”），通常由 break、return 或者未捕获异常引起，会向生成器的迭代器发送一个信号使其终止。

严格地说，在循环正常结束之后，for..of 循环也会向迭代器发送这个信号。对于生成器来说，这本质上是没有意义的操作，因为生成器的迭代器需要先完成 for..of 循环才能结束。但是，自定义的迭代器可能会需要从 for..of 循环的消费者那里接收这个额外的信号。

尽管 for..of 循环会自动发送这个信号，但你可能会希望向一个迭代器手工发送这个信号。可以通过调用 return(..) 实现这一点。

如果在生成器内有 try..finally 语句，它将总是运行，即使生成器已经外部结束。如果需要清理资源的话（数据库连接等），这一点非常有用：

```
function *something() {
    try {
        var nextVal;

        while (true) {
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            yield nextVal;
        }
    }
    // 清理子句
    finally {
        console.log( "cleaning up!" );
    }
}
```

之前的例子中，for..of 循环内的 break 会触发 finally 语句。但是，也可以在外部通过 return(..) 手工终止生成器的迭代器实例：

```
var it = something();
for (var v of it) {
    console.log( v );

    // 不要死循环!
    if (v > 500) {
        console.log(
            // 完成生成器的迭代器
            it.return( "Hello World" ).value
        );
        // 这里不需要break
    }
}
// 1 9 33 105 321 969
// 清理!
// Hello World
```

调用 it.return(..) 之后，它会立即终止生成器，这当然会运行 finally 语句。另外，它还会把返回的 value 设置为传入 return(..) 的内容，这也就是 "Hello World" 被传出去的过程。现在我们也不需要包含 break 语句了，因为生成器的迭代器已经被设置为 done:true，所以 for..of 循环会在下一个迭代终止。