考虑一下附着在这个数组上的迭代器（尽管任何迭代器都有如下性质）：

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

spread 运算符 ... 完全消耗了迭代器。考虑：

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

foo( ...a );            // 15
```

... 也可以把一个迭代器展开到一个数组中：

```
var b = [ 0, ...a, 6 ];
b;                      // [0,1,2,3,4,5,6]
```

数组解构（参见 2.4 节）可以部分或完全（如果和 rest / gather 运算符 ... 配对使用的话）消耗一个迭代器：

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

var [x,y] = it;
// 从it中获取前两个元素
var [z, ...w] = it;
// 获取第三个元素，然后一次取得其余所有元素

// it已经完全耗尽？是的。
it.next();              // { value: undefined, done: true }

x;                      // 1
y;                      // 2
z;                      // 3
w;                      // [4,5]
```

3.2 生成器

所有的函数都运行直到完毕，对吗？换句话说，一旦一个函数开始运行，在它结束之前不会被任何事情打断。

至少对于 JavaScript 到目前为止的整个历史来说，是这样的。而 ES6 引入了一个全新的某种程度上说是奇异的函数形式，称为生成器。生成器可以在执行当中暂停自身，可以立即恢复执行也可以过一段时间之后恢复执行。所以显然它并不像普通函数那样保证运行到完毕。

还有，在执行当中的每次暂停 / 恢复循环都提供了一个双向信息传递的机会，生成器可以返回一个值，恢复它的控制代码也可以发回一个值。

和前一节的迭代器一样，可以从多个角度理解生成器是什么，或者最适合做什么。没有单

个正确答案，我们是试着从几个角度考虑的。

参见本系列《你不知道的 JavaScript（中卷）》第二部分可以获取关于生成器的更多信息，也可以参考第 4 章同名小节。

3.2.1 语法

通过以下新语法声明生成器函数：

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

从功能上来说，* 的位置无所谓。同样的声明可以写作：

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

这里的**唯一**区别就是风格喜好。多数其他文献似乎都喜爱 `function* foo(..) { .. }` 这种形式。但我喜爱 `function *foo(..) { .. }`，所以后面章节都会采用这种形式。

我的理由纯粹就是说教性质的。本部分中，当提到生成器函数时，我都会使用 `*foo(..)`，而用 `foo(..)` 来指代普通函数。我发现 `*foo(..)` 与 `function *foo(..) { .. }` 中 * 的位置更加吻合。

还有，正如我们在第 2 章中已经看到的简洁方法，在对象字面量中有一种简洁生成器形式：

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

我要说的是，有了简洁生成器，`*foo() { .. }` 比 `* foo() { .. }` 更自然。所以更进一步支持了与 `*foo()` 的一致性。

一致性易于理解和学习。

1. 运行生成器

尽管生成器用 * 声明，但执行起来还和普通函数一样：

```
foo();
```

你也可以传递参数给它，就像：

```
function *foo(x,y) {
    // ..
}

foo( 5, 10 );
```

主要的区别是，执行生成器，比如 foo(5,10)，并不实际在生成器中运行代码。相反，它会产生一个迭代器控制这个生成器执行其代码。

我们会在 3.3.2 节回到这个主题，现在简单地说就是：

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

var it = foo();

// 要启动/继续*foo()，调用it.next(..)
```

2. yield

生成器还有一个可以在其中使用的新关键字，用来标示暂停点：yield。考虑：

```
function *foo() {
    var x = 10;
    var y = 20;

    yield;

    var z = x + y;
}
```

在这个 *foo() 生成器中，首先执行前两行操作，然后 yield 会暂停这个生成器。如果恢复的话，恢复时会运行 *foo() 的最后一行。生成器中 yield 可以出现任意多次（严格说，或者根本不出现！）。

你甚至可以把 yield 放在循环中，用来表示一个重复暂停点。实际上，一个永不结束的循环就意味着一个永不结束的生成器，这是完全有效的，有时候完全就是你所需要的。

yield 不只是一个暂停点。它是一个表达式，在暂停生成器的时候发出一个值。这里是一个生成器中的 while..true 循环，每次迭代都会 yield 出一个新的随机数：

```
function *foo() {
    while (true) {
        yield Math.random();
    }
}
```

yield .. 表达式不只发送一个值——没有值的 yield 等价于 yield undefined——而且还会接收（也就是被替换为）最终的恢复值。考虑：

```
function *foo() {
    var x = yield 10;
    console.log( x );
}
```

这个生成器首先在暂停自身的时候 yield 出值 10。通过我们前面给出的 it.next(..) 恢复生成器的时候，恢复给定的值（如果有的话）就会替换 / 完成整个 yield 10 表达式，意味着这个值会被赋给变量 x。

yield.. 表达式可以出现在所有普通表达式可用的地方。举例来说：

```
function *foo() {
    var arr = [ yield 1, yield 2, yield 3 ];
    console.log( arr, yield 4 );
}
```

这里的 *foo() 有 4 个 yield.. 表达式。每一个 yield 都会导致这个生成器暂停等待一个恢复值，然后把这个恢复值用在各种表达式上下文中。

yield 严格上说不是一个运算符，尽管像 yield 1 这样使用它的时候确实看起来像是运算符。因为 yield 可以单独使用，比如 var x = yield;，把它当作运算符有时会令人迷惑。

严格来说，yield.. 和像 a = 3 这样的赋值表达式有同样的“表达式优先级”——类似于运算符优先级的概念。这意味着 yield.. 基本上可以出现在任何 a = 3 合法出现的位置。

让我们来考虑对称的这个例子：

```
var a, b;

a = 3;                  // 合法
b = 2 + a = 3;          // 不合法
b = 2 + (a = 3);        // 合法

yield 3;                // 合法
a = 2 + yield 3;        // 不合法
a = 2 + (yield 3);      // 合法
```

认真思考一下可以理解，yield.. 表达式和赋值表达式行为上的类似性有一定概念上的合理性。当一个暂停的 yield 表达式恢复的时候，它会被完成 / 替代为它的恢复值，采取的方式和“赋值”给这个值是一样的。

要点：如果需要 yield.. 出现在某个位置，而这个位置上像 a = 3 这样的赋值不允许出现，那么就要用 ( ) 封装。

因为 yield 关键字的优先级很低，几乎 yield.. 之后的任何表达式都会首先计算，然后再通过 yield 发送。只有 spread 运算符 ... 和逗号运算符 , 拥有更低的优先级，也就是说它

们会在 yield 已经被求值之后才会被绑定。

所以和普通语句中的多运算符一样，另外一个可能需要 ( ) 的情况是要覆盖（提升）yield 的低优先级，就像以下这些表达式的区别一样：

```
yield 2 + 3;            // 等价于yield (2 + 3)

(yield 2) + 3;          // 首先yield 2，然后+ 3
```

和 = 赋值一样，yield 也是“右结合”的，也就是说多个 yield 表达式连续出现等价于用 (..) 从右向左分组。所以，yield yield yield 3 会被当作 yield(yield(yield 3))。像 ((yield) yield) yield 3 这样的“左结合”解释是无意义的。

像对运算符一样，如果 yield 与其他运算符或者多个 yield 一起使用，通过 (..) 分组来澄清意图是好习惯，即使是在并不严格需要的时候。

要想获取其他关于运算符优先级和结合性的信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分。

3. yield*

* 使得一个 function 声明成了 function* 生成器声明，类似地，* 使得 yield 成为了 yield *，这是一个完全不同的机制，称为 **yield 委托**（yield delegation）。语法上说，yield *.. 行为方式与 yield.. 完全相同，和我们上一小节讨论的一样。

yield * .. 需要一个 iterable；然后它会调用这个 iterable 的迭代器，把自己的生成器控制委托给这个迭代器，直到其耗尽。考虑：

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

和生成器声明时的 * 位置一样（前面讨论过），* 的位置在 yield * 表达式中只是一个风格问题，可以由你自由选择。多数其他文献采用 yield* ..，而我喜欢 yield *..，原因和前面讨论过的类似。

值 [1,2,3] 产生了一个迭代器，一步输出一个值，所以 *foo() 生成器会随着消耗这些值把它们 yield 出来。展示这一特性的另一个方法是展示 yield 委托到另一个生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

```
function *bar() {
    yield *foo();
}
```

*bar() 调用 *foo() 的时候产生的迭代器通过 yield * 委托，这意味着不管 *foo() 产生什么值，这些值都会被 *bar() 产出。

使用 yield..，表达式的完成值来自于用 it.next(..) 恢复生成器的值，而对于 yield *.. 表达式来说，完成值来自于被委托的迭代器的返回值（如果有的话）。

正如我们在 3.1.4 节讨论过的，内置迭代器通常没有返回值。而如果自定义迭代器（或者生成器）的话，可以设计为 return 一个值，yield *.. 可以捕获这个值：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    return 4;
}

function *bar() {
    var x = yield *foo();
    console.log( "x:", x );
}

for (var v of bar()) {
    console.log( v );
}
// 1 2 3
// x: 4
```

值 1、2 和 3 从 *foo() 中 yield 出来后再从 *bar() 中 yield 出来，然后从 *foo() 返回的值 4 是 yield *foo() 表达式的完成值，被赋给了 x。

因为 yield * 可以调用另外一个生成器（通过委托到其迭代器），所以它也可以通过调用自身执行某种生成器递归：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

foo( 1 );
```

foo(1) 以及之后的调用迭代器的 next() 来运行递归步骤的结果是 24。第一个 *foo(..) 运行 x 值为 1，满足 x < 3。x + 1 被递归地传给 *foo(..)，所以这一次 x 为 2。再次的递归调用使得 x 值为 3。

现在，因为不满足 x < 3，递归停止，返回 3 * 2 也就是 6 给前一个调用的 yield *.. 表达式，这个值被赋给 x。再次返回 6 * 2 的结果 12 给前一次调用的 x。最后是 12 * 2，也就是 24，返回给 *foo() 生成器的完成结果。

3.2.2 迭代器控制

前面我们简单介绍过生成器由迭代器控制这个概念。这里再深入探讨一下。

回忆一下前一小节中的递归 *foo(..)。下面是运行它的方式：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

var it = foo( 1 );
it.next();              // { value: 24, done: true }
```

在上面的例子中，生成器没有真正暂停，因为并没有 yield .. 表达式。相反，yield * 只是通过递归调用保存当前的迭代步骤。所以，只要一次调用迭代器的 next() 函数就运行了整个生成器。

现在，让我们来考虑一个有多个步骤，因此有多个产生值的生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

我们已经知道，可以通过 for..of 循环消耗迭代器，即使是一个附着在 *foo() 这样的生成器上的迭代器：

```
for (var v of foo()) {
    console.log( v );
}
// 1 2 3
```

for..of 循环需要一个 iterable。生成器函数引用（比如 foo）自己并不是一个 iterable；需要通过 foo() 执行它才能得到一个迭代器（也是一个 iterable，本章前面我们已经解释过）。理论上说可以为 GeneratorPrototype（所有生成器函数的原型）扩展一个主要就是 return this() 的 Symbol.iterator 函数。这会使得 foo 引用本身成为一个 iterable，也就是说 for (var v of foo) { .. }（注意 foo 上没有 ()）可以工作。

下面让我们来手动迭代这个生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();              // { value: 1, done: false }
it.next();              // { value: 2, done: false }
it.next();              // { value: 3, done: false }

it.next();              // { value: undefined, done: true }
```

如果仔细观察可以看到，其中有 3 个 yield 语句和 4 个 next() 调用。这个不匹配看起来很奇怪。实际上，假定所有都被计算，生成器完整运行到结束，next() 调用总是会比 yield 语句多 1 个。

但是如果从相反的角度观察（由内向外而不是由外向内），yield 和 next() 的匹配更合理一些。

别忘了 yield.. 表达式用恢复生成器所用的值完成。这意味着传给 next(..) 的参数完成了当前 yield.. 表达式暂停等待完成的。

我们用以下方式说明这种思路：

```
function *foo() {
    var x = yield 1;
    var y = yield 2;
    var z = yield 3;
    console.log( x, y, z );
}
```

在这段代码中，每个 yield.. 从（1, 2, 3）中发出一个值，更直接地说，它是暂停生成器来等待一个值。换句话说几乎等价于在问“这里我应该用什么值？请回复。”这个问题。

下面是我们如何控制 *foo() 来启动它：

```
var it = foo();

it.next();              // { value: 1, done: false }
```

第一个 next() 调用初始的暂停状态启动生成器，运行直到第一个 yield。在调用第一个 next() 的时候，并没有 yield.. 表达式等待完成。如果向第一个 next() 调用传入一个值，这个值会马上被丢弃，因为并没有 yield 等待接收这个值。

“ES6 之后”的一个早期提案会通过生成器内部一个独立的元属性（参考第 7 章），支持访问传入最初 next(..) 调用的值。

现在，让我们来回答当前这个遗留问题，即“赋给 x 的值应该是什么？”我们通过发送一个值给**下一个** next(..) 调用来回答这个问题：

```
it.next( "foo" );           // { value: 2, done: false }
```

现在，x 的值就是 "foo"，但我们又提出了一个新问题，即“我们要给 y 赋什么值？”答案是：

```
it.next( "bar" );           // { value: 3, done: false }
```

给出答案，并提出一个新问题。最后答案是：

```
it.next( "baz" );           // "foo" "bar" "baz"
                            // { value: undefined, done: true }
```

现在应该更清楚每个 yield.. 的“问题”是如何由下一个 next(..) 调用来回答了，所以我们看到的“额外的”next() 调用就是启动所有这一切的第一个。

让我们把所有步骤集合到一起：

```
var it = foo();

// 启动生成器
it.next();              // { value: 1, done: false }

// 回答第一个问题
it.next( "foo" );       // { value: 2, done: false }

// 回答第二个问题
it.next( "bar" );       // { value: 3, done: false }

// 回答第三个问题
it.next( "baz" );       // "foo" "bar" "baz"
                        // { value: undefined, done: true }
```

你可以把生成器看作是值的产生器，其中每次迭代就是产生一个值来消费。

但是，从更通用的意义上来说，可能更合理的角度是把生成器看作一个受控的、可传递的代码执行，更像是 3.1.5 节中的 tasks 队列示例。

这个角度就是我们将在第 4 章中再次讨论生成器的动机。具体来说，并不需要 next(..) 在前一个 next(..) 结束后再被调用。在生成器的内部执行上下文被暂停时，程序的其余部分仍是未被阻塞的，包括控制生成器恢复的异步动作能力。

3.2.3 提前完成

本章前面讨论过，生成器上附着的迭代器支持可选的 `return(..)` 和 `throw(..)` 方法。这两种方法都有立即终止一个暂停的生成器的效果。

考虑：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();            // { value: 1, done: false }

it.return( 42 );      // { value: 42, done: true }

it.next();            // { value: undefined, done: true }
```

`return(x)` 有点像强制立即执行一个 `return x`，这样就能够立即得到指定值。一旦生成器完成，或者正常完毕或者像前面展示的那样提前结束，都不会再执行任何代码也不会返回任何值。

`return(..)` 除了可以手动调用，还可以在每次迭代的末尾被任何消耗迭代器的 ES6 构件自动调用，比如 `for..of` 循环和 spread 运算符 `...`。

这个功能的目的是通知生成器如果控制代码不再在它上面迭代，那么它可能就会执行清理任务（释放资源、重置状态等）。和普通的函数清理模式相同，完成这一点的主要方式是通过 `finally` 子句：

```
function *foo() {
    try {
        yield 1;
        yield 2;
        yield 3;
    }
    finally {
        console.log( "cleanup!" );
    }
}

for (var v of foo()) {
    console.log( v );
}
// 1 2 3
// cleanup!

var it = foo();
```