```
for (let _ of nTimes(3)) {
  console.log('foo');
}
// foo
// foo
// foo
```

传给生成器的函数可以控制迭代循环的次数。在 n 为 0 时，while 条件为假，循环退出，生成器函数返回。

2. 使用 yield 实现输入和输出

除了可以作为函数的中间返回语句使用，yield 关键字还可以作为函数的中间参数使用。上一次让生成器函数暂停的 yield 关键字会接收到传给 next()方法的第一个值。这里有个地方不太好理解——第一次调用 next()传入的值不会被使用，因为这一次调用是为了开始执行生成器函数：

```
function* generatorFn(initial) {
  console.log(initial);
  console.log(yield);
  console.log(yield);
}

let generatorObject = generatorFn('foo');

generatorObject.next('bar');  // foo
generatorObject.next('baz');  // baz
generatorObject.next('qux');  // qux
```

yield 关键字可以同时用于输入和输出，如下例所示：

```
function* generatorFn() {
  return yield 'foo';
}

let generatorObject = generatorFn();

console.log(generatorObject.next());       // { done: false, value: 'foo' }
console.log(generatorObject.next('bar'));  // { done: true, value: 'bar' }
```

因为函数必须对整个表达式求值才能确定要返回的值，所以它在遇到 yield 关键字时暂停执行并计算出要产生的值："foo"。下一次调用 next()传入了"bar"，作为交给同一个 yield 的值。然后这个值被确定为本次生成器函数要返回的值。

yield 关键字并非只能使用一次。比如，以下代码就定义了一个无穷计数生成器函数：

```
function* generatorFn() {
  for (let i = 0;;++i) {
    yield i;
  }
}

let generatorObject = generatorFn();

console.log(generatorObject.next().value);  // 0
console.log(generatorObject.next().value);  // 1
console.log(generatorObject.next().value);  // 2
console.log(generatorObject.next().value);  // 3
console.log(generatorObject.next().value);  // 4
console.log(generatorObject.next().value);  // 5
...
```

假设我们想定义一个生成器函数，它会根据配置的值迭代相应次数并产生迭代的索引。初始化一个新数组可以实现这个需求，但不用数组也可以实现同样的行为：

```
function* nTimes(n) {
  for (let i = 0; i < n; ++i) {
    yield i;
  }
}

for (let x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

另外，使用 `while` 循环也可以，而且代码稍微简洁一点：

```
function* nTimes(n) {
  let i = 0;
  while(n--) {
    yield i++;
  }
}

for (let x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

这样使用生成器也可以实现范围和填充数组：

```
function* range(start, end) {
  while(end > start) {
    yield start++;
  }
}

for (const x of range(4, 7)) {
  console.log(x);
}
// 4
// 5
// 6

function* zeroes(n) {
  while(n--) {
    yield 0;
  }
}

console.log(Array.from(zeroes(8))); // [0, 0, 0, 0, 0, 0, 0, 0]
```

3. 产生可迭代对象

可以使用星号增强 `yield` 的行为，让它能够迭代一个可迭代对象，从而一次产出一个值：

```
// 等价的 generatorFn:
// function* generatorFn() {
//   for (const x of [1, 2, 3]) {
//     yield x;
//   }
// }
function* generatorFn() {
  yield* [1, 2, 3];
}

let generatorObject = generatorFn();

for (const x of generatorFn()) {
  console.log(x);
}
// 1
// 2
// 3
```

与生成器函数的星号类似，`yield` 星号两侧的空格不影响其行为：

```
function* generatorFn() {
  yield* [1, 2];
  yield *[3, 4];
  yield * [5, 6];
}

for (const x of generatorFn()) {
  console.log(x);
}
// 1
// 2
// 3
// 4
// 5
// 6
```

因为 `yield*`实际上只是将一个可迭代对象序列化为一连串可以单独产出的值，所以这跟把 `yield` 放到一个循环里没什么不同。下面两个生成器函数的行为是等价的：

```
function* generatorFnA() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

for (const x of generatorFnA()) {
  console.log(x);
}
// 1
// 2
// 3

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

for (const x of generatorFnB()) {
  console.log(x);
}
```

```
// 1
// 2
// 3
```

yield*的值是关联迭代器返回 done: true 时的 value 属性。对于普通迭代器来说，这个值是undefined：

```
function* generatorFn() {
  console.log('iter value:', yield* [1, 2, 3]);
}

for (const x of generatorFn()) {
  console.log('value:', x);
}
// value: 1
// value: 2
// value: 3
// iter value: undefined
```

对于生成器函数产生的迭代器来说，这个值就是生成器函数返回的值：

```
function* innerGeneratorFn() {
  yield 'foo';
  return 'bar';
}
function* outerGeneratorFn(genObj) {
  console.log('iter value:', yield* innerGeneratorFn());
}

for (const x of outerGeneratorFn()) {
  console.log('value:', x);
}
// value: foo
// iter value: bar
```

4. 使用 yield*实现递归算法

yield*最有用的地方是实现递归操作，此时生成器可以产生自身。看下面的例子：

```
function* nTimes(n) {
  if (n > 0) {
    yield* nTimes(n - 1);
    yield n - 1;
  }
}

for (const x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

在这个例子中，每个生成器首先都会从新创建的生成器对象产出每个值，然后再产出一个整数。结果就是生成器函数会递归地减少计数器值，并实例化另一个生成器对象。从最顶层来看，这就相当于创建一个可迭代对象并返回递增的整数。

使用递归生成器结构和 yield*可以优雅地表达递归算法。下面是一个图的实现，用于生成一个随机的双向图：

```
class Node {
  constructor(id) {
    this.id = id;
    this.neighbors = new Set();
  }

  connect(node) {
    if (node !== this) {
      this.neighbors.add(node);
      node.neighbors.add(this);
    }
  }
}

class RandomGraph {
  constructor(size) {
    this.nodes = new Set();

    // 创建节点
    for (let i = 0; i < size; ++i) {
      this.nodes.add(new Node(i));
    }

    // 随机连接节点
    const threshold = 1 / size;
    for (const x of this.nodes) {
      for (const y of this.nodes) {
        if (Math.random() < threshold) {
          x.connect(y);
        }
      }
    }
  }

  // 这个方法仅用于调试
  print() {
    for (const node of this.nodes) {
      const ids = [...node.neighbors]
                      .map((n) => n.id)
                      .join(',');

      console.log(`${node.id}: ${ids}`);
    }
  }
}

const g = new RandomGraph(6);

g.print();
// 示例输出:
// 0: 2,3,5
// 1: 2,3,4,5
// 2: 1,3
// 3: 0,1,2,4
// 4: 2,3
// 5: 0,4
```

图数据结构非常适合递归遍历，而递归生成器恰好非常合用。为此，生成器函数必须接收一个可迭代对象，产出该对象中的每一个值，并且对每个值进行递归。这个实现可以用来测试某个图是否连通，

即是否没有不可到达的节点。只要从一个节点开始，然后尽力访问每个节点就可以了。结果就得到了一个非常简洁的深度优先遍历：

```
class Node {
  constructor(id) {
    ...
  }

  connect(node) {
    ...
  }
}

class RandomGraph {
  constructor(size) {
    ...
  }

  print() {
    ...
  }

  isConnected() {
    const visitedNodes = new Set();

    function* traverse(nodes) {
      for (const node of nodes) {
        if (!visitedNodes.has(node)) {
          yield node;
          yield* traverse(node.neighbors);
        }
      }
    }

    // 取得集合中的第一个节点
    const firstNode = this.nodes[Symbol.iterator]().next().value;

    // 使用递归生成器迭代每个节点
    for (const node of traverse([firstNode])) {
      visitedNodes.add(node);
    }

    return visitedNodes.size === this.nodes.size;
  }
}
```

7.3.3 生成器作为默认迭代器

因为生成器对象实现了 `Iterable` 接口，而且生成器函数和默认迭代器被调用之后都产生迭代器，所以生成器格外适合作为默认迭代器。下面是一个简单的例子，这个类的默认迭代器可以用一行代码产出类的内容：

```
class Foo {
  constructor() {
    this.values = [1, 2, 3];
  }
```

```
  * [Symbol.iterator]() {
    yield* this.values;
  }
}

const f = new Foo();
for (const x of f) {
  console.log(x);
}
// 1
// 2
// 3
```

这里，for-of 循环调用了默认迭代器（它恰好又是一个生成器函数）并产生了一个生成器对象。这个生成器对象是可迭代的，所以完全可以在迭代中使用。

7.3.4　提前终止生成器

与迭代器类似，生成器也支持“可关闭”的概念。一个实现 Iterator 接口的对象一定有 next()方法，还有一个可选的 return()方法用于提前终止迭代器。生成器对象除了有这两个方法，还有第三个方法：throw()。

```
function* generatorFn() {}

const g = generatorFn();

console.log(g);          // generatorFn {<suspended>}
console.log(g.next);     // f next() { [native code] }
console.log(g.return);   // f return() { [native code] }
console.log(g.throw);    // f throw() { [native code] }
```

return()和 throw()方法都可以用于强制生成器进入关闭状态。

1. return()

return()方法会强制生成器进入关闭状态。提供给 return()方法的值，就是终止迭代器对象的值：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

console.log(g);              // generatorFn {<suspended>}
console.log(g.return(4));    // { done: true, value: 4 }
console.log(g);              // generatorFn {<closed>}
```

与迭代器不同，所有生成器对象都有 return()方法，只要通过它进入关闭状态，就无法恢复了。后续调用 next()会显示 done：true 状态，而提供的任何返回值都不会被存储或传播：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}
```

```
const g = generatorFn();

console.log(g.next());     // { done: false, value: 1 }
console.log(g.return(4));  // { done: true, value: 4 }
console.log(g.next());     // { done: true, value: undefined }
console.log(g.next());     // { done: true, value: undefined }
console.log(g.next());     // { done: true, value: undefined }
```

for-of 循环等内置语言结构会忽略状态为 done: true 的 IteratorObject 内部返回的值。

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

for (const x of g) {
  if (x > 1) {
    g.return(4);
  }
  console.log(x);
}
// 1
// 2
```

2. throw()

throw()方法会在暂停的时候将一个提供的错误注入到生成器对象中。如果错误未被处理，生成器就会关闭：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

console.log(g);   // generatorFn {<suspended>}
try {
  g.throw('foo');
} catch (e) {
  console.log(e); // foo
}
console.log(g);   // generatorFn {<closed>}
```

不过，假如生成器函数**内部**处理了这个错误，那么生成器就不会关闭，而且还可以恢复执行。错误处理会跳过对应的 yield，因此在这个例子中会跳过一个值。比如：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    try {
      yield x;
    } catch(e) {}
  }
}
```

```
const g = generatorFn();

console.log(g.next()); // { done: false, value: 1}
g.throw('foo');
console.log(g.next()); // { done: false, value: 3}
```

在这个例子中，生成器在 try/catch 块中的 yield 关键字处暂停执行。在暂停期间，throw()方法向生成器对象内部注入了一个错误：字符串"foo"。这个错误会被 yield 关键字抛出。因为错误是在生成器的 try/catch 块中抛出的，所以仍然在生成器内部被捕获。可是，由于 yield 抛出了那个错误，生成器就不会再产出值 2。此时，生成器函数继续执行，在下一次迭代再次遇到 yield 关键字时产出了值 3。

> **注意**　如果生成器对象还没有开始执行，那么调用 throw()抛出的错误不会在函数内部被捕获，因为这相当于在函数块外部抛出了错误。

7.4　小结

迭代是一种所有编程语言中都可以看到的模式。ECMAScript 6 正式支持迭代模式并引入了两个新的语言特性：迭代器和生成器。

迭代器是一个可以由任意对象实现的接口，支持连续获取对象产出的每一个值。任何实现 Iterable 接口的对象都有一个 Symbol.iterator 属性，这个属性引用默认迭代器。默认迭代器就像一个迭代器工厂，也就是一个函数，调用之后会产生一个实现 Iterator 接口的对象。

迭代器必须通过连续调用 next()方法才能连续取得值，这个方法返回一个 IteratorObject。这个对象包含一个 done 属性和一个 value 属性。前者是一个布尔值，表示是否还有更多值可以访问；后者包含迭代器返回的当前值。这个接口可以通过手动反复调用 next()方法来消费，也可以通过原生消费者，比如 for-of 循环来自动消费。

生成器是一种特殊的函数，调用之后会返回一个生成器对象。生成器对象实现了 Iterable 接口，因此可用在任何消费可迭代对象的地方。生成器的独特之处在于支持 yield 关键字，这个关键字能够暂停执行生成器函数。使用 yield 关键字还可以通过 next()方法接收输入和产生输出。在加上星号之后，yield 关键字可以将跟在它后面的可迭代对象序列化为一连串值。

第8章 对象、类与面向对象编程

本章内容

- 理解对象
- 理解对象创建过程
- 理解继承
- 理解类

ECMA-262 将对象定义为一组属性的无序集合。严格来说，这意味着对象就是一组没有特定顺序的值。对象的每个属性或方法都由一个名称来标识，这个名称映射到一个值。正因为如此（以及其他还未讨论的原因），可以把 ECMAScript 的对象想象成一张散列表，其中的内容就是一组名/值对，值可以是数据或者函数。

8.1 理解对象

创建自定义对象的通常方式是创建 `Object` 的一个新实例，然后再给它添加属性和方法，如下例所示：

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";
person.sayName = function() {
  console.log(this.name);
};
```

这个例子创建了一个名为 `person` 的对象，而且有三个属性（`name`、`age` 和 `job`）和一个方法（`sayName()`）。`sayName()`方法会显示 `this.name` 的值，这个属性会解析为 `person.name`。早期 JavaScript 开发者频繁使用这种方式创建新对象。几年后，对象字面量变成了更流行的方式。前面的例子如果使用对象字面量则可以这样写：

```
let person = {
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};
```

这个例子中的 `person` 对象跟前面例子中的 `person` 对象是等价的，它们的属性和方法都一样。这些属性都有自己的特征，而这些特征决定了它们在 JavaScript 中的行为。