```
// foo
// bar
// baz

// 数组解构
let [a, b, c] = arr;
console.log(a, b, c); // foo, bar, baz

// 扩展操作符
let arr2 = [...arr];
console.log(arr2); // ['foo', 'bar', 'baz']

// Array.from()
let arr3 = Array.from(arr);
console.log(arr3); // ['foo', 'bar', 'baz']

// Set 构造函数
let set = new Set(arr);
console.log(set); // Set(3) {'foo', 'bar', 'baz'}

// Map 构造函数
let pairs = arr.map((x, i) => [x, i]);
console.log(pairs); // [['foo', 0], ['bar', 1], ['baz', 2]]
let map = new Map(pairs);
console.log(map); // Map(3) { 'foo'=>0, 'bar'=>1, 'baz'=>2 }
```

如果对象原型链上的父类实现了 Iterable 接口，那这个对象也就实现了这个接口：

```
class FooArray extends Array {}
let fooArr = new FooArray('foo', 'bar', 'baz');

for (let el of fooArr) {
  console.log(el);
}
// foo
// bar
// baz
```

7.2.2　迭代器协议

迭代器是一种一次性使用的对象，用于迭代与其关联的可迭代对象。迭代器 API 使用 next()方法在可迭代对象中遍历数据。每次成功调用 next()，都会返回一个 IteratorResult 对象，其中包含迭代器返回的下一个值。若不调用 next()，则无法知道迭代器的当前位置。

next()方法返回的迭代器对象 IteratorResult 包含两个属性：done 和 value。done 是一个布尔值，表示是否还可以再次调用 next()取得下一个值；value 包含可迭代对象的下一个值（done 为 false），或者 undefined（done 为 true）。done：true 状态称为“耗尽”。可以通过以下简单的数组来演示：

```
// 可迭代对象
let arr = ['foo', 'bar'];

// 迭代器工厂函数
console.log(arr[Symbol.iterator]); // f values() { [native code] }

// 迭代器
```

```
let iter = arr[Symbol.iterator]();
console.log(iter); // ArrayIterator {}

// 执行迭代
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: true, value: undefined }
```

这里通过创建迭代器并调用 next()方法按顺序迭代了数组，直至不再产生新值。迭代器并不知道怎么从可迭代对象中取得下一个值，也不知道可迭代对象有多大。只要迭代器到达 done: true 状态，后续调用 next()就一直返回同样的值了：

```
let arr = ['foo'];
let iter = arr[Symbol.iterator]();
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
```

每个迭代器都表示对可迭代对象的一次性有序遍历。不同迭代器的实例相互之间没有联系，只会独立地遍历可迭代对象：

```
let arr = ['foo', 'bar'];
let iter1 = arr[Symbol.iterator]();
let iter2 = arr[Symbol.iterator]();

console.log(iter1.next()); // { done: false, value: 'foo' }
console.log(iter2.next()); // { done: false, value: 'foo' }
console.log(iter2.next()); // { done: false, value: 'bar' }
console.log(iter1.next()); // { done: false, value: 'bar' }
```

迭代器并不与可迭代对象某个时刻的快照绑定，而仅仅是使用游标来记录遍历可迭代对象的历程。如果可迭代对象在迭代期间被修改了，那么迭代器也会反映相应的变化：

```
let arr = ['foo', 'baz'];
let iter = arr[Symbol.iterator]();

console.log(iter.next()); // { done: false, value: 'foo' }

// 在数组中间插入值
arr.splice(1, 0, 'bar');

console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: false, value: 'baz' }
console.log(iter.next()); // { done: true, value: undefined }
```

> **注意** 迭代器维护着一个指向可迭代对象的引用，因此迭代器会阻止垃圾回收程序回收可迭代对象。

“迭代器”的概念有时候容易模糊，因为它可以指通用的迭代，也可以指接口，还可以指正式的迭代器类型。下面的例子比较了一个显式的迭代器实现和一个原生的迭代器实现。

```
// 这个类实现了可迭代接口（Iterable）
// 调用默认的迭代器工厂函数会返回
// 一个实现迭代器接口（Iterator）的迭代器对象
class Foo {
```

```
  [Symbol.iterator]() {
    return {
      next() {
        return { done: false, value: 'foo' };
      }
    }
  }
}
let f = new Foo();

// 打印出实现了迭代器接口的对象
console.log(f[Symbol.iterator]()); // { next: f() {} }

// Array 类型实现了可迭代接口 (Iterable)
// 调用 Array 类型的默认迭代器工厂函数
// 会创建一个 ArrayIterator 的实例
let a = new Array();

// 打印出 ArrayIterator 的实例
console.log(a[Symbol.iterator]()); // Array Iterator {}
```

7.2.3　自定义迭代器

与 Iterable 接口类似，任何实现 Iterator 接口的对象都可以作为迭代器使用。下面这个例子中的 Counter 类只能被迭代一定的次数：

```
class Counter {
  // Counter 的实例应该迭代 limit 次
  constructor(limit) {
    this.count = 1;
    this.limit = limit;
  }

  next() {
    if (this.count <= this.limit) {
      return { done: false, value: this.count++ };
    } else {
      return { done: true, value: undefined };
    }
  }
  [Symbol.iterator]() {
    return this;
  }
}

let counter = new Counter(3);

for (let i of counter) {
  console.log(i);
}
// 1
// 2
// 3
```

这个类实现了 Iterator 接口，但不理想。这是因为它的每个实例只能被迭代一次：

```
for (let i of counter) { console.log(i); }
// 1
```

```
// 2
// 3

for (let i of counter) { console.log(i); }
// (nothing logged)
```

为了让一个可迭代对象能够创建多个迭代器，必须每创建一个迭代器就对应一个新计数器。为此，可以把计数器变量放到闭包里，然后通过闭包返回迭代器：

```
class Counter {
  constructor(limit) {
    this.limit = limit;
  }

  [Symbol.iterator]() {
    let count = 1,
        limit = this.limit;
    return {
      next() {
        if (count <= limit) {
          return { done: false, value: count++ };
        } else {
          return { done: true, value: undefined };
        }
      }
    };
  }
}

let counter = new Counter(3);

for (let i of counter) { console.log(i); }
// 1
// 2
// 3

for (let i of counter) { console.log(i); }
// 1
// 2
// 3
```

每个以这种方式创建的迭代器也实现了 Iterable 接口。Symbol.iterator 属性引用的工厂函数会返回相同的迭代器：

```
let arr = ['foo', 'bar', 'baz'];
let iter1 = arr[Symbol.iterator]();

console.log(iter1[Symbol.iterator]);  // f values() { [native code] }

let iter2 = iter1[Symbol.iterator]();

console.log(iter1 === iter2);         // true
```

因为每个迭代器也实现了 Iterable 接口，所以它们可以用在任何期待可迭代对象的地方，比如 for-of 循环：

```
let arr = [3, 1, 4];
let iter = arr[Symbol.iterator]();
```

```
for (let item of arr ) { console.log(item); }
// 3
// 1
// 4

for (let item of iter ) { console.log(item); }
// 3
// 1
// 4
```

7.2.4　提前终止迭代器

可选的 return()方法用于指定在迭代器提前关闭时执行的逻辑。执行迭代的结构在想让迭代器知道它不想遍历到可迭代对象耗尽时，就可以“关闭”迭代器。可能的情况包括：

- for-of 循环通过 break、continue、return 或 throw 提前退出；
- 解构操作并未消费所有值。

return()方法必须返回一个有效的 IteratorResult 对象。简单情况下，可以只返回{ done: true }。因为这个返回值只会用在生成器的上下文中，所以本章后面再讨论这种情况。

如下面的代码所示，内置语言结构在发现还有更多值可以迭代，但不会消费这些值时，会自动调用 return()方法。

```
class Counter {
  constructor(limit) {
    this.limit = limit;
  }

  [Symbol.iterator]() {
    let count = 1,
      limit = this.limit;
    return {
      next() {
        if (count <= limit) {
          return { done: false, value: count++ };
        } else {
          return { done: true };
        }
      },
      return() {
        console.log('Exiting early');
        return { done: true };
      }
    };
  }
}


let counter1 = new Counter(5);

for (let i of counter1) {
  if (i > 2) {
    break;
  }
  console.log(i);
}
```

```
// 1
// 2
// Exiting early


let counter2 = new Counter(5);

try {
  for (let i of counter2) {
    if (i > 2) {
      throw 'err';
    }
    console.log(i);
  }
} catch(e) {}
// 1
// 2
// Exiting early


let counter3 = new Counter(5);

let [a, b] = counter3;
// Exiting early
```

如果迭代器没有关闭，则还可以继续从上次离开的地方继续迭代。比如，数组的迭代器就是不能关闭的：

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

for (let i of iter) {
  console.log(i);
  if (i > 2) {
    break
  }
}
// 1
// 2
// 3

for (let i of iter) {
  console.log(i);
}
// 4
// 5
```

因为 `return()`方法是可选的，所以并非所有迭代器都是可关闭的。要知道某个迭代器是否可关闭，可以测试这个迭代器实例的 `return` 属性是不是函数对象。不过，仅仅给一个不可关闭的迭代器增加这个方法**并不能**让它变成可关闭的。这是因为调用 `return()`不会强制迭代器进入关闭状态。即便如此，`return()`方法还是会被调用。

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

iter.return = function() {
  console.log('Exiting early');
  return { done: true };
```

```
};

for (let i of iter) {
  console.log(i);
  if (i > 2) {
    break
  }
}
// 1
// 2
// 3
// 提前退出

for (let i of iter) {
  console.log(i);
}
// 4
// 5
```

7.3　生成器

生成器是 ECMAScript 6 新增的一个极为灵活的结构，拥有在一个函数块内暂停和恢复代码执行的能力。这种新能力具有深远的影响，比如，使用生成器可以自定义迭代器和实现协程。

7.3.1　生成器基础

生成器的形式是一个函数，函数名称前面加一个星号（*）表示它是一个生成器。只要是可以定义函数的地方，就可以定义生成器。

```
// 生成器函数声明
function* generatorFn() {}

// 生成器函数表达式
let generatorFn = function* () {}

// 作为对象字面量方法的生成器函数
let foo = {
  * generatorFn() {}
}

// 作为类实例方法的生成器函数
class Foo {
  * generatorFn() {}
}

// 作为类静态方法的生成器函数
class Bar {
  static * generatorFn() {}
}
```

> **注意**　箭头函数不能用来定义生成器函数。

标识生成器函数的星号不受两侧空格的影响：

```
// 等价的生成器函数：
function* generatorFnA() {}
function *generatorFnB() {}
function * generatorFnC() {}

// 等价的生成器方法：
class Foo {
  *generatorFnD() {}
  * generatorFnE() {}
}
```

调用生成器函数会产生一个**生成器对象**。生成器对象一开始处于暂停执行（suspended）的状态。与迭代器相似，生成器对象也实现了 `Iterator` 接口，因此具有 `next()`方法。调用这个方法会让生成器开始或恢复执行。

```
function* generatorFn() {}

const g = generatorFn();

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

`next()`方法的返回值类似于迭代器，有一个 `done` 属性和一个 `value` 属性。函数体为空的生成器函数中间不会停留，调用一次 `next()`就会让生成器到达 `done: true` 状态。

```
function* generatorFn() {}

let generatorObject = generatorFn();

console.log(generatorObject);         // generatorFn {<suspended>}
console.log(generatorObject.next());  // { done: true, value: undefined }
```

`value` 属性是生成器函数的返回值，默认值为 `undefined`，可以通过生成器函数的返回值指定：

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

let generatorObject = generatorFn();

console.log(generatorObject);         // generatorFn {<suspended>}
console.log(generatorObject.next());  // { done: true, value: 'foo' }
```

生成器函数只会在初次调用 `next()`方法后开始执行，如下所示：

```
function* generatorFn() {
  console.log('foobar');
}

// 初次调用生成器函数并不会打印日志
let generatorObject = generatorFn();

generatorObject.next();  // foobar
```

生成器对象实现了 `Iterable` 接口，它们默认的迭代器是自引用的：

```
function* generatorFn() {}

console.log(generatorFn);
// f* generatorFn() {}
console.log(generatorFn()[Symbol.iterator]);
```

```
// f [Symbol.iterator]() {native code}
console.log(generatorFn());
// generatorFn {<suspended>}
console.log(generatorFn()[Symbol.iterator]());
// generatorFn {<suspended>}

const g = generatorFn();

console.log(g === g[Symbol.iterator]());
// true
```

7.3.2　通过 yield 中断执行

yield 关键字可以让生成器停止和开始执行，也是生成器最有用的地方。生成器函数在遇到 yield 关键字之前会正常执行。遇到这个关键字后，执行会停止，函数作用域的状态会被保留。停止执行的生成器函数只能通过在生成器对象上调用 next()方法来恢复执行：

```
function* generatorFn() {
  yield;
}

let generatorObject = generatorFn();

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

此时的 yield 关键字有点像函数的中间返回语句，它生成的值会出现在 next()方法返回的对象里。通过 yield 关键字退出的生成器函数会处在 done: false 状态；通过 return 关键字退出的生成器函数会处于 done: true 状态。

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

let generatorObject = generatorFn();

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

生成器函数内部的执行流程会针对每个生成器对象区分作用域。在一个生成器对象上调用 next()不会影响其他生成器：

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

let generatorObject1 = generatorFn();
let generatorObject2 = generatorFn();


console.log(generatorObject1.next()); // { done: false, value: 'foo' }
console.log(generatorObject2.next()); // { done: false, value: 'foo' }
```

```
console.log(generatorObject2.next()); // { done: false, value: 'bar' }
console.log(generatorObject1.next()); // { done: false, value: 'bar' }
```

yield 关键字只能在生成器函数内部使用，用在其他地方会抛出错误。类似函数的 return 关键字，yield 关键字必须直接位于生成器函数定义中，出现在嵌套的非生成器函数中会抛出语法错误：

```
// 有效
function* validGeneratorFn() {
  yield;
}

// 无效
function* invalidGeneratorFnA() {
  function a() {
    yield;
  }
}

// 无效
function* invalidGeneratorFnB() {
  const b = () => {
    yield;
  }
}

// 无效
function* invalidGeneratorFnC() {
  (() => {
    yield;
  })();
}
```


1. 生成器对象作为可迭代对象

在生成器对象上显式调用 next()方法的用处并不大。其实，如果把生成器对象当成可迭代对象，那么使用起来会更方便：

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

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

在需要自定义迭代对象时，这样使用生成器对象会特别有用。比如，我们需要定义一个可迭代对象，而它会产生一个迭代器，这个迭代器会执行指定的次数。使用生成器，可以通过一个简单的循环来实现：

```
function* nTimes(n) {
  while(n--) {
    yield;
  }
}
```