```
  value = "newVal";
  alert(value);          // newVal
  alert(s1.has("val1")); // true
}

const valObj = {id: 1};

const s2 = new Set([valObj]);

// 修改值对象的属性，但对象仍然存在于集合中
for (let value of s2.values()) {
  value.id = "newVal";
  alert(value);           // {id: "newVal"}
  alert(s2.has(valObj));  // true
}
alert(valObj);            // {id: "newVal"}
```

6.6.3　定义正式集合操作

从各方面来看，Set 跟 Map 都很相似，只是 API 稍有调整。唯一需要强调的就是集合的 API 对自身的简单操作。很多开发者都喜欢使用 Set 操作，但需要手动实现：或者是子类化 Set，或者是定义一个实用函数库。要把两种方式合二为一，可以在子类上实现静态方法，然后在实例方法中使用这些静态方法。在实现这些操作时，需要考虑几个地方。

- 某些 Set 操作是有关联性的，因此最好让实现的方法能支持处理任意多个集合实例。
- Set 保留插入顺序，所有方法返回的集合必须保证顺序。
- 尽可能高效地使用内存。扩展操作符的语法很简洁，但尽可能避免集合和数组间的相互转换能够节省对象初始化成本。
- 不要修改已有的集合实例。union(a, b)或 a.union(b)应该返回包含结果的新集合实例。

```
class XSet extends Set {
  union(...sets) {
    return XSet.union(this, ...sets)
  }

  intersection(...sets) {
    return XSet.intersection(this, ...sets);
  }

  difference(set) {
    return XSet.difference(this, set);
  }

  symmetricDifference(set) {
    return XSet.symmetricDifference(this, set);
  }

  cartesianProduct(set) {
    return XSet.cartesianProduct(this, set);
  }

  powerSet() {
    return XSet.powerSet(this);
  }
```

```
// 返回两个或更多集合的并集
static union(a, ...bSets) {
  const unionSet = new XSet(a);
  for (const b of bSets) {
    for (const bValue of b) {
      unionSet.add(bValue);
    }
  }
  return unionSet;
}

// 返回两个或更多集合的交集
static intersection(a, ...bSets) {
  const intersectionSet = new XSet(a);
  for (const aValue of intersectionSet) {
    for (const b of bSets) {
      if (!b.has(aValue)) {
        intersectionSet.delete(aValue);
      }
    }
  }
  return intersectionSet;
}

// 返回两个集合的差集
static difference(a, b) {
  const differenceSet = new XSet(a);
  for (const bValue of b) {
    if (a.has(bValue)) {
      differenceSet.delete(bValue);
    }
  }
  return differenceSet;
}

// 返回两个集合的对称差集
static symmetricDifference(a, b) {
  // 按照定义，对称差集可以表达为
  return a.union(b).difference(a.intersection(b));
}

// 返回两个集合（数组对形式）的笛卡儿积
// 必须返回数组集合，因为笛卡儿积可能包含相同值的对
static cartesianProduct(a, b) {
  const cartesianProductSet = new XSet();
  for (const aValue of a) {
    for (const bValue of b) {
      cartesianProductSet.add([aValue, bValue]);
    }
  }
  return cartesianProductSet;
}

// 返回一个集合的幂集
static powerSet(a) {
  const powerSet = new XSet().add(new XSet());
  for (const aValue of a) {
```

```
        for (const set of new XSet(powerSet)) {
          powerSet.add(new XSet(set).add(aValue));
        }
      }
      return powerSet;
    }
  }
```

6.7 WeakSet

ECMAScript 6 新增的“弱集合”（WeakSet）是一种新的集合类型，为这门语言带来了集合数据结构。WeakSet 是 Set 的“兄弟”类型，其 API 也是 Set 的子集。WeakSet 中的“weak”（弱），描述的是 JavaScript 垃圾回收程序对待“弱集合”中值的方式。

6.7.1 基本 API

可以使用 new 关键字实例化一个空的 WeakSet：

```
const ws = new WeakSet();
```

弱集合中的值只能是 Object 或者继承自 Object 的类型，尝试使用非对象设置值会抛出 TypeError。

如果想在初始化时填充弱集合，则构造函数可以接收一个可迭代对象，其中需要包含有效的值。可迭代对象中的每个值都会按照迭代顺序插入到新实例中：

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};
// 使用数组初始化弱集合
const ws1 = new WeakSet([val1, val2, val3]);

alert(ws1.has(val1)); // true
alert(ws1.has(val2)); // true
alert(ws1.has(val3)); // true

// 初始化是全有或全无的操作
// 只要有一个值无效就会抛出错误，导致整个初始化失败
const ws2 = new WeakSet([val1, "BADVAL", val3]);
// TypeError: Invalid value used in WeakSet
typeof ws2;
// ReferenceError: ws2 is not defined

// 原始值可以先包装成对象再用作值
const stringVal = new String("val1");
const ws3 = new WeakSet([stringVal]);
alert(ws3.has(stringVal)); // true
```

初始化之后可以使用 add()再添加新值，可以使用 has()查询，还可以使用 delete()删除：

```
const ws = new WeakSet();

const val1 = {id: 1},
      val2 = {id: 2};

alert(ws.has(val1)); // false

ws.add(val1)
```

```
  .add(val2);

alert(ws.has(val1)); // true
alert(ws.has(val2)); // true

ws.delete(val1);     // 只删除这一个值

alert(ws.has(val1)); // false
alert(ws.has(val2)); // true
```

add()方法返回弱集合实例，因此可以把多个操作连缀起来，包括初始化声明：

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};

const ws = new WeakSet().add(val1);

ws.add(val2)
  .add(val3);

alert(ws.has(val1)); // true
alert(ws.has(val2)); // true
alert(ws.has(val3)); // true
```


6.7.2 弱值

WeakSet 中“weak”表示弱集合的值是“弱弱地拿着”的。意思就是，这些值不属于正式的引用，不会阻止垃圾回收。

来看下面的例子：

```
const ws = new WeakSet();

ws.add({});
```

add()方法初始化了一个新对象，并将它用作一个值。因为没有指向这个对象的其他引用，所以当这行代码执行完成后，这个对象值就会被当作垃圾回收。然后，这个值就从弱集合中消失了，使其成为一个空集合。

再看一个稍微不同的例子：

```
const ws = new WeakSet();

const container = {
  val: {}
};

ws.add(container.val);

function removeReference() {
  container.val = null;
}
```

这一次，container 对象维护着一个对弱集合值的引用，因此这个对象值不会成为垃圾回收的目标。不过，如果调用了 removeReference()，就会摧毁值对象的最后一个引用，垃圾回收程序就可以把这个值清理掉。

6.7.3　不可迭代值

因为 WeakSet 中的值任何时候都可能被销毁，所以没必要提供迭代其值的能力。当然，也用不着像 clear()这样一次性销毁所有值的方法。WeakSet 确实没有这个方法。因为不可能迭代，所以也不可能在不知道对象引用的情况下从弱集合中取得值。即便代码可以访问 WeakSet 实例，也没办法看到其中的内容。

WeakSet 之所以限制只能用对象作为值，是为了保证只有通过值对象的引用才能取得值。如果允许原始值，那就没办法区分初始化时使用的字符串字面量和初始化之后使用的一个相等的字符串了。

6.7.4　使用弱集合

相比于 WeakMap 实例，WeakSet 实例的用处没有那么大。不过，弱集合在给对象打标签时还是有价值的。

来看下面的例子，这里使用了一个普通 Set：

```
const disabledElements = new Set();

const loginButton = document.querySelector('#login');

// 通过加入对应集合，给这个节点打上“禁用”标签
disabledElements.add(loginButton);
```

这样，通过查询元素在不在 disabledElements 中，就可以知道它是不是被禁用了。不过，假如元素从 DOM 树中被删除了，它的引用却仍然保存在 Set 中，因此垃圾回收程序也不能回收它。

为了让垃圾回收程序回收元素的内存，可以在这里使用 WeakSet：

```
const disabledElements = new WeakSet();

const loginButton = document.querySelector('#login');

// 通过加入对应集合，给这个节点打上“禁用”标签
disabledElements.add(loginButton);
```

这样，只要 WeakSet 中任何元素从 DOM 树中被删除，垃圾回收程序就可以忽略其存在，而立即释放其内存（假设没有其他地方引用这个对象）。

6.8　迭代与扩展操作

ECMAScript 6 新增的迭代器和扩展操作符对集合引用类型特别有用。这些新特性让集合类型之间相互操作、复制和修改变得异常方便。

> **注意**　第 7 章会更详细地介绍迭代器和生成器。

如本章前面所示，有 4 种原生集合类型定义了默认迭代器：

- Array
- 所有定型数组
- Map
- Set

很简单，这意味着上述所有类型都支持顺序迭代，都可以传入 for-of 循环：

```
let iterableThings = [
  Array.of(1, 2),
  typedArr = Int16Array.of(3, 4),
  new Map([[5, 6], [7, 8]]),
  new Set([9, 10])
];

for (const iterableThing of iterableThings) {
  for (const x of iterableThing) {
    console.log(x);
  }
}

// 1
// 2
// 3
// 4
// [5, 6]
// [7, 8]
// 9
// 10
```

这也意味着所有这些类型都兼容扩展操作符。扩展操作符在对可迭代对象执行浅复制时特别有用，只需简单的语法就可以复制整个对象：

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1];

console.log(arr1);          // [1, 2, 3]
console.log(arr2);          // [1, 2, 3]
console.log(arr1 === arr2); // false
```

对于期待可迭代对象的构造函数，只要传入一个可迭代对象就可以实现复制：

```
let map1 = new Map([[1, 2], [3, 4]]);
let map2 = new Map(map1);

console.log(map1); // Map {1 => 2, 3 => 4}
console.log(map2); // Map {1 => 2, 3 => 4}
```

当然，也可以构建数组的部分元素：

```
let arr1 = [1, 2, 3];
let arr2 = [0, ...arr1, 4, 5];

console.log(arr2); // [0, 1, 2, 3, 4, 5]
```

浅复制意味着只会复制对象引用：

```
let arr1 = [{}];
let arr2 = [...arr1];

arr1[0].foo = 'bar';
console.log(arr2[0]); // { foo: 'bar' }
```

上面的这些类型都支持多种构建方法，比如 Array.of()和 Array.from()静态方法。在与扩展操作符一起使用时，可以非常方便地实现互操作：

```
let arr1 = [1, 2, 3];

// 把数组复制到定型数组
let typedArr1 = Int16Array.of(...arr1);
let typedArr2 = Int16Array.from(arr1);
console.log(typedArr1);   // Int16Array [1, 2, 3]
console.log(typedArr2);   // Int16Array [1, 2, 3]

// 把数组复制到映射
let map = new Map(arr1.map((x) => [x, 'val' + x]));
console.log(map);   // Map {1 => 'val 1', 2 => 'val 2', 3 => 'val 3'}

// 把数组复制到集合
let set = new Set(typedArr2);
console.log(set);   // Set {1, 2, 3}

// 把集合复制回数组
let arr2 = [...set];
console.log(arr2);  // [1, 2, 3]
```

6.9　小结

JavaScript 中的对象是引用值，可以通过几种内置引用类型创建特定类型的对象。

- 引用类型与传统面向对象编程语言中的类相似，但实现不同。
- `Object` 类型是一个基础类型，所有引用类型都从它继承了基本的行为。
- `Array` 类型表示一组有序的值，并提供了操作和转换值的能力。
- 定型数组包含一套不同的引用类型，用于管理数值在内存中的类型。
- `Date` 类型提供了关于日期和时间的信息，包括当前日期和时间以及计算。
- `RegExp` 类型是 ECMAScript 支持的正则表达式的接口，提供了大多数基本正则表达式以及一些高级正则表达式的能力。

JavaScript 比较独特的一点是，函数其实是 `Function` 类型的实例，这意味着函数也是对象。由于函数是对象，因此也就具有能够增强自身行为的方法。

因为原始值包装类型的存在，所以 JavaScript 中的原始值可以拥有类似对象的行为。有 3 种原始值包装类型：`Boolean`、`Number` 和 `String`。它们都具有如下特点。

- 每种包装类型都映射到同名的原始类型。
- 在以读模式访问原始值时，后台会实例化一个原始值包装对象，通过这个对象可以操作数据。
- 涉及原始值的语句只要一执行完毕，包装对象就会立即销毁。

JavaScript 还有两个在一开始执行代码时就存在的内置对象：`Global` 和 `Math`。其中，`Global` 对象在大多数 ECMAScript 实现中无法直接访问。不过浏览器将 `Global` 实现为 `window` 对象。所有全局变量和函数都是 `Global` 对象的属性。`Math` 对象包含辅助完成复杂数学计算的属性和方法。

ECMAScript 6 新增了一批引用类型：`Map`、`WeakMap`、`Set` 和 `WeakSet`。这些类型为组织应用程序数据和简化内存管理提供了新能力。

第 7 章

迭代器与生成器

本章内容

- ❑ 理解迭代
- ❑ 迭代器模式
- ❑ 生成器

迭代的英文“iteration”源自拉丁文 itero，意思是“重复”或“再来”。在软件开发领域，“迭代”的意思是按照顺序反复多次执行一段程序，通常会有明确的终止条件。ECMAScript 6 规范新增了两个高级特性：迭代器和生成器。使用这两个特性，能够更清晰、高效、方便地实现迭代。

7.1 理解迭代

在 JavaScript 中，计数循环就是一种最简单的迭代：

```
for (let i = 1; i <= 10; ++i) {
  console.log(i);
}
```

循环是迭代机制的基础，这是因为它可以指定迭代的次数，以及每次迭代要执行什么操作。每次循环都会在下一次迭代开始之前完成，而每次迭代的顺序都是事先定义好的。

迭代会在一个有序集合上进行。（“有序”可以理解为集合中所有项都可以按照既定的顺序被遍历到，特别是开始和结束项有明确的定义。）数组是 JavaScript 中有序集合的最典型例子。

```
let collection = ['foo', 'bar', 'baz'];

for (let index = 0; index < collection.length; ++index) {
  console.log(collection[index]);
}
```

因为数组有已知的长度，且数组每一项都可以通过索引获取，所以整个数组可以通过递增索引来遍历。由于如下原因，通过这种循环来执行例程并不理想。

- ❑ **迭代之前需要事先知道如何使用数据结构**。数组中的每一项都只能先通过引用取得数组对象，然后再通过`[]`操作符取得特定索引位置上的项。这种情况并不适用于所有数据结构。
- ❑ **遍历顺序并不是数据结构固有的**。通过递增索引来访问数据是特定于数组类型的方式，并不适用于其他具有隐式顺序的数据结构。

ES5 新增了`Array.prototype.forEach()`方法，向通用迭代需求迈进了一步（但仍然不够理想）：

```
let collection = ['foo', 'bar', 'baz'];

collection.forEach((item) => console.log(item));
```

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

这个方法解决了单独记录索引和通过数组对象取得值的问题。不过，没有办法标识迭代何时终止。因此这个方法只适用于数组，而且回调结构也比较笨拙。

在 ECMAScript 较早的版本中，执行迭代必须使用循环或其他辅助结构。随着代码量增加，代码会变得越发混乱。很多语言都通过原生语言结构解决了这个问题，开发者无须事先知道如何迭代就能实现迭代操作。这个解决方案就是**迭代器模式**。Python、Java、C++，还有其他很多语言都对这个模式提供了完备的支持。JavaScript 在 ECMAScript 6 以后也支持了迭代器模式。

7.2　迭代器模式

迭代器模式（特别是在 ECMAScript 这个语境下）描述了一个方案，即可以把有些结构称为“可迭代对象”（iterable），因为它们实现了正式的 `Iterable` 接口，而且可以通过迭代器 `Iterator` 消费。

可迭代对象是一种抽象的说法。基本上，可以把可迭代对象理解成数组或集合这样的集合类型的对象。它们包含的元素都是有限的，而且都具有无歧义的遍历顺序：

```
// 数组的元素是有限的
// 递增索引可以按序访问每个元素
let arr = [3, 1, 4];

// 集合的元素是有限的
// 可以按插入顺序访问每个元素
let set = new Set().add(3).add(1).add(4);
```

不过，可迭代对象不一定是集合对象，也可以是仅仅具有类似数组行为的其他数据结构，比如本章开头提到的计数循环。该循环中生成的值是暂时性的，但循环本身是在执行迭代。计数循环和数组都具有可迭代对象的行为。

> **注意**　临时性可迭代对象可以实现为生成器，本章后面会讨论。

任何实现 `Iterable` 接口的数据结构都可以被实现 `Iterator` 接口的结构“消费”（consume）。**迭代器**（iterator）是按需创建的一次性对象。每个迭代器都会关联一个**可迭代对象**，而迭代器会暴露迭代其关联可迭代对象的 API。迭代器无须了解与其关联的可迭代对象的结构，只需要知道如何取得连续的值。这种概念上的分离正是 `Iterable` 和 `Iterator` 的强大之处。

7.2.1　可迭代协议

实现 `Iterable` 接口（可迭代协议）要求同时具备两种能力：支持迭代的自我识别能力和创建实现 `Iterator` 接口的对象的能力。在 ECMAScript 中，这意味着必须暴露一个属性作为“默认迭代器”，而且这个属性必须使用特殊的 `Symbol.iterator` 作为键。这个默认迭代器属性必须引用一个迭代器工厂函数，调用这个工厂函数必须返回一个新迭代器。

很多内置类型都实现了 `Iterable` 接口：

- 字符串
- 数组

- 映射
- 集合
- arguments 对象
- NodeList 等 DOM 集合类型

检查是否存在默认迭代器属性可以暴露这个工厂函数：

```
let num = 1;
let obj = {};

// 这两种类型没有实现迭代器工厂函数
console.log(num[Symbol.iterator]); // undefined
console.log(obj[Symbol.iterator]); // undefined

let str = 'abc';
let arr = ['a', 'b', 'c'];
let map = new Map().set('a', 1).set('b', 2).set('c', 3);
let set = new Set().add('a').add('b').add('c');
let els = document.querySelectorAll('div');

// 这些类型都实现了迭代器工厂函数
console.log(str[Symbol.iterator]); // f values() { [native code] }
console.log(arr[Symbol.iterator]); // f values() { [native code] }
console.log(map[Symbol.iterator]); // f values() { [native code] }
console.log(set[Symbol.iterator]); // f values() { [native code] }
console.log(els[Symbol.iterator]); // f values() { [native code] }

// 调用这个工厂函数会生成一个迭代器
console.log(str[Symbol.iterator]()); // StringIterator {}
console.log(arr[Symbol.iterator]()); // ArrayIterator {}
console.log(map[Symbol.iterator]()); // MapIterator {}
console.log(set[Symbol.iterator]()); // SetIterator {}
console.log(els[Symbol.iterator]()); // ArrayIterator {}
```

实际写代码过程中，不需要显式调用这个工厂函数来生成迭代器。实现可迭代协议的所有类型都会自动兼容接收可迭代对象的任何语言特性。接收可迭代对象的原生语言特性包括：

- for-of 循环
- 数组解构
- 扩展操作符
- Array.from()
- 创建集合
- 创建映射
- Promise.all()接收由期约组成的可迭代对象
- Promise.race()接收由期约组成的可迭代对象
- yield*操作符，在生成器中使用

这些原生语言结构会在后台调用提供的可迭代对象的这个工厂函数，从而创建一个迭代器：

```
let arr = ['foo', 'bar', 'baz'];

// for-of 循环
for (let el of arr) {
  console.log(el);
}
```