ECMAScript 6 入门

作者：阮一峰

目录

其他

Iterator 和 for...of 循环

上一章 下一章

---

1. Iterator（遍历器）的概念

JavaScript 原有的表示"集合"的数据结构，主要是数组（`Array`）和对象（`Object`），ES6 又添加了 `Map` 和 `Set`。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是 `Map`， `Map` 的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令 `for...of` 循环，Iterator 接口主要供 `for...of` 消费。

Iterator 的遍历过程是这样的。

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的 `next` 方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的 `next` 方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的 `next` 方法，直到它指向数据结构的结束位置。

每一次调用 `next` 方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含 `value` 和 `done` 两个属性的对象。其中，`value` 属性是当前成员的值， `done` 属性是一个布尔值，表示遍历是否结束。

下面是一个模拟 `next` 方法返回值的例子。

```
var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }
it.next() // { value: "b", done: false }
it.next() // { value: undefined, done: true }

function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++], done: false} :
        {value: undefined, done: true};
    }
  };
}
```

上面代码定义了一个 `makeIterator` 函数，它是一个遍历器生成函数，作用就是返回一个遍历器对象。对数组 `['a', 'b']` 执行这个函数，就会返回该数组的遍历器对象（即指针对象） `it` 。

指针对象的 `next` 方法，用来移动指针。开始时，指针指向数组的开始位置。然后，每次调用 `next` 方法，指针就会指向数组的下一个成员。第一次调用，指向 `a` ；第二次调用，指向 `b` 。

`next` 方法返回一个对象，表示当前数据成员的信息。这个对象具有 `value` 和 `done` 两个属性， `value` 属性返回当前位置的成员， `done` 属性是一个布尔值，表示遍历是否结束，即是否还有必要再一次调用 `next` 方法。

总之，调用指针对象的 `next` 方法，就可以遍历事先给定的数据结构。

对于遍历器对象来说， `done: false` 和 `value: undefined` 属性都是可以省略的，因此上面的 `makeIterator` 函数可以简写成下面的形式。

```
function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++]} :
        {done: true};
    }
  };
}
```

由于 Iterator 只是把接口规格加到数据结构之上，所以，遍历器与它所遍历的那个数据结构，实际上是分开的，完全可以写出没有对应数据结构的遍历器对象，或者说用遍历器对象模拟出数据结构。下面是一个无限运行的遍历器对象的例子。

```
var it = idMaker();

it.next().value // 0
it.next().value // 1
it.next().value // 2
// ...

function idMaker() {
  var index = 0;

  return {
    next: function() {
      return {value: index++, done: false};
    }
  };
}
```

上面的例子中，遍历器生成函数 `idMaker` ，返回一个遍历器对象（即指针对象）。但是并没有对应的数据结构，或者说，遍历器对象自己描述了一个数据结构出来。

如果使用 TypeScript 的写法，遍历器接口（Iterable）、指针对象（Iterator）和 `next` 方法返回值的规格可以描述如下。

```
interface Iterable {
  [Symbol.iterator]() : Iterator,
}

interface Iterator {
  next(value?: any) : IterationResult,
}

interface IterationResult {
  value: any,
  done: boolean,
}
```

2. 默认 Iterator 接口

Iterator 接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即 `for...of` 循环（详见下文）。当使用 `for...of` 循环遍历某种数据结构时，该循环会自动去寻找 Iterator 接口。

一种数据结构只要部署了 Iterator 接口，我们就称这种数据结构是"可遍历的"（iterable）。

ES6 规定，默认的 Iterator 接口部署在数据结构的 `Symbol.iterator` 属性，或者说，一个数据结构只要具有 `Symbol.iterator` 属性，就可以认为是"可遍历的"（iterable）。`Symbol.iterator` 属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名 `Symbol.iterator`，它是一个表达式，返回 `Symbol` 对象的 `iterator` 属性，这是一个预定义好的、类型为 Symbol 的特殊值，所以要放在方括号内（参见《Symbol》一章）。

```
const obj = {
  [Symbol.iterator] : function () {
    return {
      next: function () {
        return {
          value: 1,
          done: true
        };
      }
    };
  }
};
```

上面代码中，对象 `obj` 是可遍历的（iterable），因为具有 `Symbol.iterator` 属性。执行这个属性，会返回一个遍历器对象。该对象的根本特征就是具有 `next` 方法。每次调用 `next` 方法，都会返回一个代表当前成员的信息对象，具有 `value` 和 `done` 两个属性。

ES6 的有些数据结构原生具备 Iterator 接口（比如数组），即不用任何处理，就可以被 `for...of` 循环遍历。原因在于，这些数据结构原生部署了 `Symbol.iterator` 属性（详见下文），另外一些数据结构没有（比如对象）。凡是部署了 `Symbol.iterator` 属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

原生具备 Iterator 接口的数据结构如下。

- Array
- Map
- Set
- String
- TypedArray
- 函数的 arguments 对象
- NodeList 对象

下面的例子是数组的 `Symbol.iterator` 属性。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }
iter.next() // { value: 'b', done: false }
iter.next() // { value: 'c', done: false }
iter.next() // { value: undefined, done: true }
```

上面代码中，变量 `arr` 是一个数组，原生就具有遍历器接口，部署在 `arr` 的 `Symbol.iterator` 属性上面。所以，调用这个属性，就得到遍历器对象。

对于原生部署 Iterator 接口的数据结构，不用自己写遍历器生成函数，`for...of` 循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的 Iterator 接口，都需要自己在 `Symbol.iterator` 属性上面部署，这样才会被 `for...of` 循环遍历。

对象（Object）之所以没有默认部署 Iterator 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作 Map 结构使用，ES5 没有 Map 结构，而 ES6 原生提供了。

一个对象如果要具备可被 `for...of` 循环调用的 Iterator 接口，就必须在 `Symbol.iterator` 的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }

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

  next() {
    var value = this.value;
    if (value < this.stop) {
      this.value++;
      return {done: false, value: value};
    }
    return {done: true, value: undefined};
  }
}

function range(start, stop) {
  return new RangeIterator(start, stop);
}

for (var value of range(0, 3)) {
  console.log(value); // 0, 1, 2
}
```

上面代码是一个类部署 Iterator 接口的写法。`Symbol.iterator` 属性对应一个函数，执行后返回当前对象的遍历器对象。

下面是通过遍历器实现指针结构的例子。

```
function Obj(value) {
  this.value = value;
  this.next = null;
}

Obj.prototype[Symbol.iterator] = function() {
  var iterator = { next: next };

  var current = this;

  function next() {
    if (current) {
      var value = current.value;
      current = current.next;
      return { done: false, value: value };
    } else {
      return { done: true };
    }
  }
  return iterator;
}
```

```
var one = new Obj(1);
var two = new Obj(2);
var three = new Obj(3);

one.next = two;
two.next = three;

for (var i of one){
  console.log(i); // 1, 2, 3
}
```

上面代码首先在构造函数的原型链上部署 `Symbol.iterator` 方法，调用该方法会返回遍历器对象 `iterator`，调用该对象的 `next` 方法，在返回一个值的同时，自动将内部指针移到下一个实例。

下面是另一个为对象添加 Iterator 接口的例子。

```
let obj = {
  data: [ 'hello', 'world' ],
  [Symbol.iterator]() {
    const self = this;
    let index = 0;
    return {
      next() {
        if (index < self.data.length) {
          return {
            value: self.data[index++],
            done: false
          };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};
```

对于类似数组的对象（存在数值键名和 `length` 属性），部署 Iterator 接口，有一个简便方法，就是 `Symbol.iterator` 方法直接引用数组的 Iterator 接口。

```
NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
// 或者
NodeList.prototype[Symbol.iterator] = [][Symbol.iterator];

[...document.querySelectorAll('div')] // 可以执行了
```

NodeList 对象是类似数组的对象，本来就具有遍历接口，可以直接遍历。上面代码中，我们将它的遍历接口改成数组的 `Symbol.iterator` 属性，可以看到没有任何影响。

下面是另一个类似数组的对象调用数组的 `Symbol.iterator` 方法的例子。

```
let iterable = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3,
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // 'a', 'b', 'c'
}
```

注意，普通对象部署数组的 `Symbol.iterator` 方法，并无效果。

```
let iterable = {
  a: 'a',
  b: 'b',
  c: 'c',
  length: 3,
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // undefined, undefined, undefined
}
```

如果 `Symbol.iterator` 方法对应的不是遍历器生成函数（即会返回一个遍历器对象），解释引擎将会报错。

```
var obj = {};

obj[Symbol.iterator] = () => 1;

[...obj] // TypeError: [] is not a function
```

上面代码中，变量 `obj` 的 `Symbol.iterator` 方法对应的不是遍历器生成函数，因此报错。

有了遍历器接口，数据结构就可以用 `for...of` 循环遍历（详见下文），也可以使用 `while` 循环遍历。

```
var $iterator = ITERABLE[Symbol.iterator]();
var $result = $iterator.next();
while (!$result.done) {
  var x = $result.value;
  // ...
  $result = $iterator.next();
}
```

上面代码中， `ITERABLE` 代表某种可遍历的数据结构， `$iterator` 是它的遍历器对象。遍历器对象每次移动指针（ `next` 方法），都检查一下返回值的 `done` 属性，如果遍历还没结束，就移动遍历器对象的指针到下一步（ `next` 方法），不断循环。

---

3. 调用 Iterator 接口的场合

有一些场合会默认调用 Iterator 接口（即 `Symbol.iterator` 方法），除了下文会介绍的 `for...of` 循环，还有几个别的场合。

（1）解构赋值

对数组和 Set 结构进行解构赋值时，会默认调用 `Symbol.iterator` 方法。

```
let set = new Set().add('a').add('b').add('c');

let [x,y] = set;
// x='a'; y='b'

let [first, ...rest] = set;
// first='a'; rest=['b','c'];
```

（2）扩展运算符

扩展运算符（...）也会调用默认的 Iterator 接口。

```
// 例一
var str = 'hello';
[...str] //  ['h','e','l','l','o']

// 例二
let arr = ['b', 'c'];
['a', ...arr, 'd']
// ['a', 'b', 'c', 'd']
```

上面代码的扩展运算符内部就调用 Iterator 接口。

实际上，这提供了一种简便机制，可以将任何部署了 Iterator 接口的数据结构，转为数组。也就是说，只要某个数据结构部署了 Iterator 接口，就可以对它使用扩展运算符，将其转为数组。

```
let arr = [...iterable];
```

（3）yield*

`yield*` 后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。

```
let generator = function* () {
  yield 1;
  yield* [2,3,4];
  yield 5;
};

var iterator = generator();

iterator.next() // { value: 1, done: false }
iterator.next() // { value: 2, done: false }
iterator.next() // { value: 3, done: false }
iterator.next() // { value: 4, done: false }
iterator.next() // { value: 5, done: false }
iterator.next() // { value: undefined, done: true }
```

（4）其他场合

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

- for...of
- Array.from()
- Map(), Set(), WeakMap(), WeakSet()（比如 `new Map([['a',1],['b',2]])` ）
- Promise.all()
- Promise.race()

---

4. 字符串的 Iterator 接口

字符串是一个类似数组的对象，也原生具有 Iterator 接口。

```
var someString = "hi";
typeof someString[Symbol.iterator]
// "function"

var iterator = someString[Symbol.iterator]();
```

```
iterator.next()  // { value: "h", done: false }
iterator.next()  // { value: "i", done: false }
iterator.next()  // { value: undefined, done: true }
```

上面代码中，调用 `Symbol.iterator` 方法返回一个遍历器对象，在这个遍历器上可以调用 next 方法，实现对于字符串的遍历。

可以覆盖原生的 `Symbol.iterator` 方法，达到修改遍历器行为的目的。

```
var str = new String("hi");

[...str] // ["h", "i"]

str[Symbol.iterator] = function() {
  return {
    next: function() {
      if (this._first) {
        this._first = false;
        return { value: "bye", done: false };
      } else {
        return { done: true };
      }
    },
    _first: true
  };
};

[...str] // ["bye"]
str // "hi"
```

上面代码中，字符串 str 的 `Symbol.iterator` 方法被修改了，所以扩展运算符（`...`）返回的值变成了 `bye`，而字符串本身还是 `hi`。

---

5. Iterator 接口与 Generator 函数

`Symbol.iterator` 方法的最简单实现，还是使用下一章要介绍的 Generator 函数。

```
let myIterable = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3;
  }
}
[...myIterable] // [1, 2, 3]

// 或者采用下面的简洁写法

let obj = {
  * [Symbol.iterator]() {
    yield 'hello';
    yield 'world';
  }
};

for (let x of obj) {
  console.log(x);
}
// "hello"
// "world"
```

上面代码中， `Symbol.iterator` 方法几乎不用部署任何代码，只要用 yield 命令给出每一步的返回值即可。

---

6. 遍历器对象的 return()，throw()

遍历器对象除了具有 `next` 方法，还可以具有 `return` 方法和 `throw` 方法。如果你自己写遍历器对象生成函数，那么 `next` 方法是必须部署的， `return` 方法和 `throw` 方法是否部署是可选的。

`return` 方法的使用场合是，如果 `for...of` 循环提前退出（通常是因为出错，或者有 `break` 语句），就会调用 `return` 方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署 `return` 方法。

```
function readLinesSync(file) {
  return {
    [Symbol.iterator]() {
      return {
        next() {
          return { done: false };
        },
        return() {
          file.close();
          return { done: true };
        }
      };
    },
  };
}
```

上面代码中，函数 `readLinesSync` 接受一个文件对象作为参数，返回一个遍历器对象，其中除了 `next` 方法，还部署了 `return` 方法。下面的两种情况，都会触发执行 `return` 方法。

```
// 情况一
for (let line of readLinesSync(fileName)) {
  console.log(line);
  break;
}

// 情况二
for (let line of readLinesSync(fileName)) {
  console.log(line);
  throw new Error();
}
```

上面代码中，情况一输出文件的第一行以后，就会执行 `return` 方法，关闭这个文件；情况二会在执行 `return` 方法关闭文件之后，再抛出错误。

注意， `return` 方法必须返回一个对象，这是 Generator 规格决定的。

`throw` 方法主要是配合 Generator 函数使用，一般的遍历器对象用不到这个方法。请参阅《Generator 函数》一章。

---

7. for...of 循环

ES6 借鉴 C++、Java、C# 和 Python 语言，引入了 `for...of` 循环，作为遍历所有数据结构的统一的方法。

一个数据结构只要部署了 `Symbol.iterator` 属性，就被视为具有 iterator 接口，就可以用 `for...of` 循环遍历它的成员。也就是说，`for...of` 循环内部调用的是数据结构的 `Symbol.iterator` 方法。

`for...of` 循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如 `arguments` 对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串。

数组

数组原生具备 `iterator` 接口（即默认部署了 `Symbol.iterator` 属性）， `for...of` 循环本质上就是调用这个接口产生的遍历器，可以用下面的代码证明。

```
const arr = ['red', 'green', 'blue'];

for(let v of arr) {
  console.log(v); // red green blue
}

const obj = {};
obj[Symbol.iterator] = arr[Symbol.iterator].bind(arr);

for(let v of obj) {
  console.log(v); // red green blue
}
```

上面代码中，空对象 `obj` 部署了数组 `arr` 的 `Symbol.iterator` 属性，结果 `obj` 的 `for...of` 循环，产生了与 `arr` 完全一样的结果。

`for...of` 循环可以代替数组实例的 `forEach` 方法。

```
const arr = ['red', 'green', 'blue'];

arr.forEach(function (element, index) {
  console.log(element); // red green blue
  console.log(index);   // 0 1 2
});
```

JavaScript 原有的 `for...in` 循环，只能获得对象的键名，不能直接获取键值。ES6 提供 `for...of` 循环，允许遍历获得键值。

```
var arr = ['a', 'b', 'c', 'd'];

for (let a in arr) {
  console.log(a); // 0 1 2 3
}

for (let a of arr) {
  console.log(a); // a b c d
}
```

上面代码表明， `for...in` 循环读取键名， `for...of` 循环读取键值。如果要通过 `for...of` 循环，获取数组的索引，可以借助数组实例的 `entries` 方法和 `keys` 方法（参见《数组的扩展》一章）。

`for...of` 循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性。这一点跟 `for...in` 循环也不一样。

```
let arr = [3, 5, 7];
arr.foo = 'hello';
```

```
for (let i in arr) {
  console.log(i); // "0", "1", "2", "foo"
}

for (let i of arr) {
  console.log(i); //  "3", "5", "7"
}
```

上面代码中， `for...of` 循环不会返回数组 `arr` 的 `foo` 属性。

Set 和 Map 结构

Set 和 Map 结构也原生具有 Iterator 接口，可以直接使用 `for...of` 循环。

```
var engines = new Set(["Gecko", "Trident", "Webkit", "Webkit"]);
for (var e of engines) {
  console.log(e);
}
// Gecko
// Trident
// Webkit

var es6 = new Map();
es6.set("edition", 6);
es6.set("committee", "TC39");
es6.set("standard", "ECMA-262");
for (var [name, value] of es6) {
  console.log(name + ": " + value);
}
// edition: 6
// committee: TC39
// standard: ECMA-262
```

上面代码演示了如何遍历 Set 结构和 Map 结构。值得注意的地方有两个，首先，遍历的顺序是按照各个成员被添加进数据结构的顺序。其次，Set 结构遍历时，返回的是一个值，而 Map 结构遍历时，返回的是一个数组，该数组的两个成员分别为当前 Map 成员的键名和键值。

```
let map = new Map().set('a', 1).set('b', 2);
for (let pair of map) {
  console.log(pair);
}
// ['a', 1]
// ['b', 2]

for (let [key, value] of map) {
  console.log(key + ' : ' + value);
}
// a : 1
// b : 2
```

计算生成的数据结构

有些数据结构是在现有数据结构的基础上，计算生成的。比如，ES6 的数组、Set、Map 都部署了以下三个方法，调用后都返回遍历器对象。

- `entries()` 返回一个遍历器对象，用来遍历 `[键名, 键值]` 组成的数组。对于数组，键名就是索引值；对于 Set，键名与键值相同。Map 结构的 Iterator 接口，默认就是调用 `entries` 方法。
- `keys()` 返回一个遍历器对象，用来遍历所有的键名。
- `values()` 返回一个遍历器对象，用来遍历所有的键值。

这三个方法调用后生成的遍历器对象，所遍历的都是计算生成的数据结构。

```
let arr = ['a', 'b', 'c'];
for (let pair of arr.entries()) {
  console.log(pair);
}
// [0, 'a']
// [1, 'b']
// [2, 'c']
```

类似数组的对象

类似数组的对象包括好几类。下面是 `for...of` 循环用于字符串、DOM NodeList 对象、 `arguments` 对象的例子。

```
// 字符串
let str = "hello";

for (let s of str) {
  console.log(s); // h e l l o
}

// DOM NodeList对象
let paras = document.querySelectorAll("p");

for (let p of paras) {
  p.classList.add("test");
}

// arguments对象
function printArgs() {
  for (let x of arguments) {
    console.log(x);
  }
}
printArgs('a', 'b');
// 'a'
// 'b'
```

对于字符串来说， `for...of` 循环还有一个特点，就是会正确识别 32 位 UTF-16 字符。

```
for (let x of 'a\uD83D\uDC0A') {
  console.log(x);
}
// 'a'
// '\uD83D\uDC0A'
```

并不是所有类似数组的对象都具有 Iterator 接口，一个简便的解决方法，就是使用 `Array.from` 方法将其转为数组。

```
let arrayLike = { length: 2, 0: 'a', 1: 'b' };
```

```
// 报错
for (let x of arrayLike) {
  console.log(x);
}

// 正确
for (let x of Array.from(arrayLike)) {
  console.log(x);
}
```

对象

对于普通的对象，`for...of` 结构不能直接使用，会报错，必须部署了 Iterator 接口后才能使用。但是，这样情况下，`for...in` 循环依然可以用来遍历键名。

```
let es6 = {
  edition: 6,
  committee: "TC39",
  standard: "ECMA-262"
};

for (let e in es6) {
  console.log(e);
}
// edition
// committee
// standard

for (let e of es6) {
  console.log(e);
}
// TypeError: es6[Symbol.iterator] is not a function
```

上面代码表示，对于普通的对象，`for...in` 循环可以遍历键名，`for...of` 循环会报错。

一种解决方法是，使用 `Object.keys` 方法将对象的键名生成一个数组，然后遍历这个数组。

```
for (var key of Object.keys(someObject)) {
  console.log(key + ': ' + someObject[key]);
}
```

另一个方法是使用 Generator 函数将对象重新包装一下。

```
function* entries(obj) {
  for (let key of Object.keys(obj)) {
    yield [key, obj[key]];
  }
}

for (let [key, value] of entries(obj)) {
  console.log(key, '->', value);
}
// a -> 1
// b -> 2
// c -> 3
```

与其他遍历语法的比较

以数组为例，JavaScript 提供多种遍历语法。最原始的写法就是 `for` 循环。

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

这种写法比较麻烦，因此数组提供内置的 `forEach` 方法。

```
myArray.forEach(function (value) {
  console.log(value);
});
```

这种写法的问题在于，无法中途跳出 `forEach` 循环， `break` 命令或 `return` 命令都不能奏效。

`for...in` 循环可以遍历数组的键名。

```
for (var index in myArray) {
  console.log(myArray[index]);
}
```

`for...in` 循环有几个缺点。

- 数组的键名是数字，但是 `for...in` 循环是以字符串作为键名“0”、“1”、“2”等等。
- `for...in` 循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。
- 某些情况下， `for...in` 循环会以任意顺序遍历键名。

总之， `for...in` 循环主要是为遍历对象而设计的，不适用于遍历数组。

`for...of` 循环相比上面几种做法，有一些显著的优点。

```
for (let value of myArray) {
  console.log(value);
}
```

- 有着同 `for...in` 一样的简洁语法，但是没有 `for...in` 那些缺点。
- 不同于 `forEach` 方法，它可以与 `break` 、 `continue` 和 `return` 配合使用。
- 提供了遍历所有数据结构的统一操作接口。

下面是一个使用 break 语句，跳出 `for...of` 循环的例子。

```
for (var n of fibonacci) {
  if (n > 1000)
    break;
  console.log(n);
}
```

上面的例子，会输出斐波纳契数列小于等于 1000 的项。如果当前项大于 1000，就会使用 `break` 语句跳出 `for...of` 循环。

留言