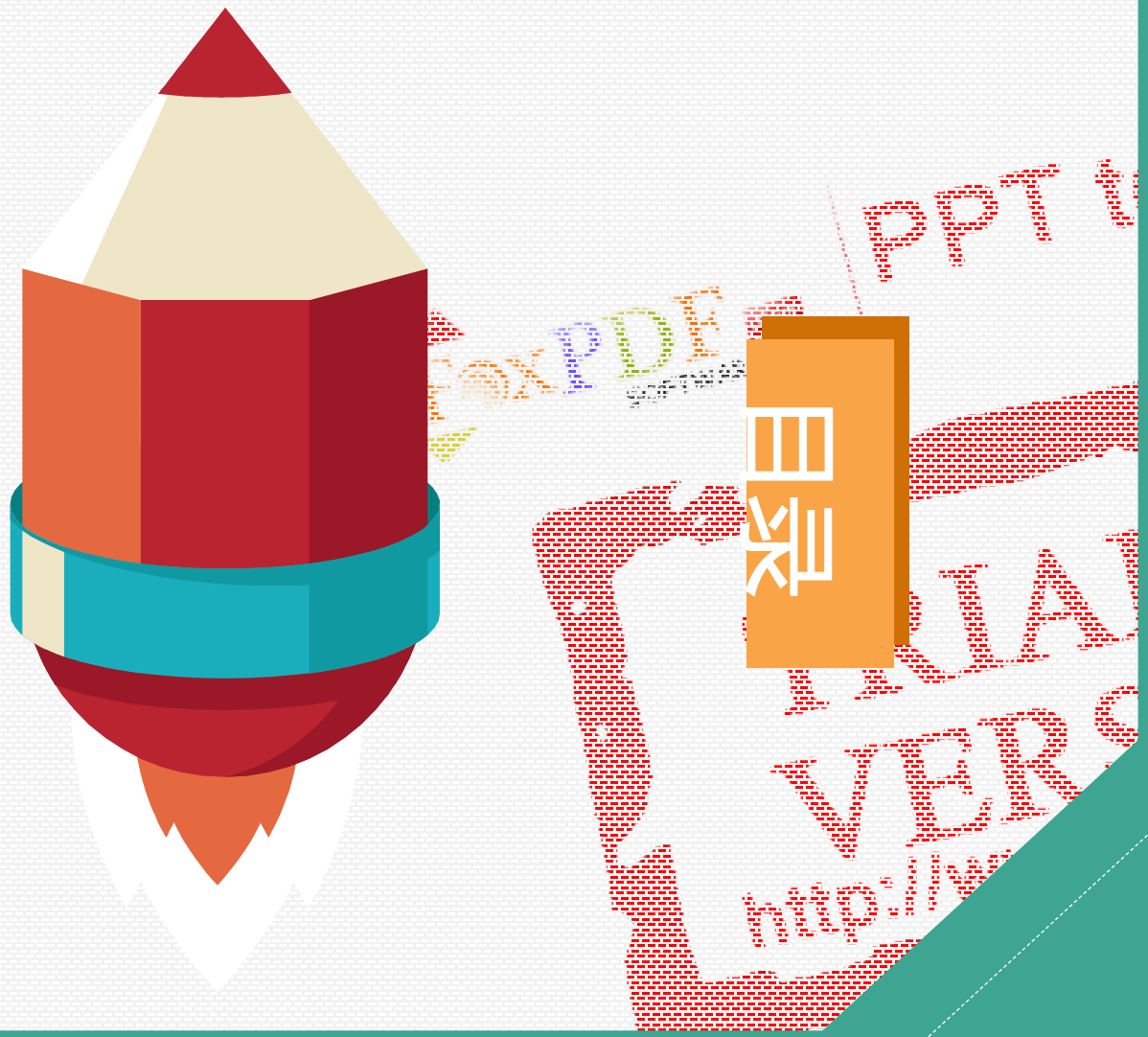
第36课：迭代器Iterator和for...of循环

主讲老师：万章



Iterator的概念

默认的迭代器接口

Iterator的常用领域

Iterator的其他方法

Iterator的概念

Iterator的概念

遍历器（Iterator）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，之前只有数组和NodeList等比较特殊的类数组才拥有该特性）

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的实现步骤

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。

Iterator的实现原理

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};
    }
  };
}
```

Iterator的实现原理

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。

```
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};
    }
  };
}
```

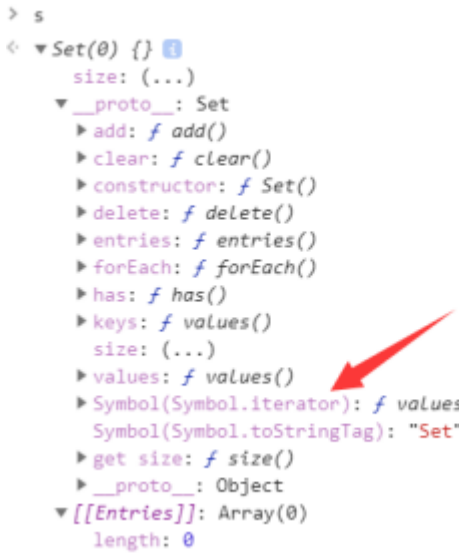
上面是一个无限运行的遍历器对象的例子。

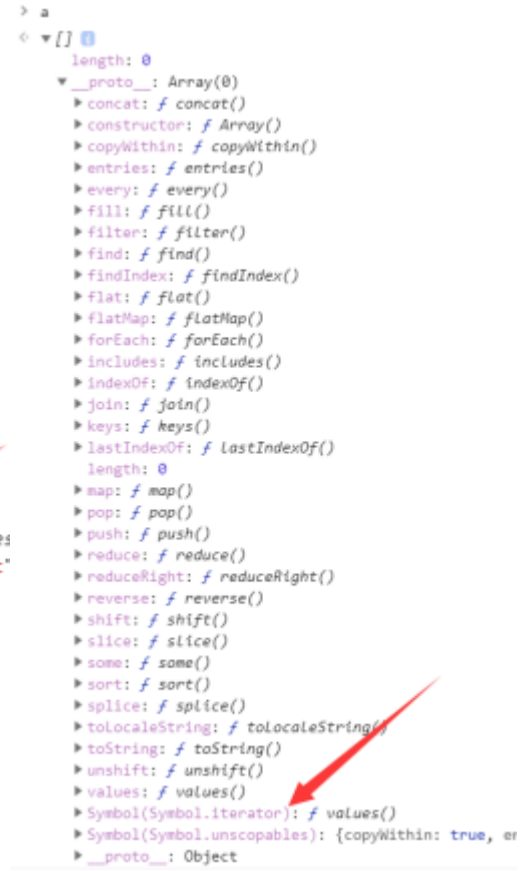
默认的迭代器接口

默认的迭代器(Iterator)接口

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)

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) 。





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（。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。

默认的迭代器(Iterator)接口

原生具备 Iterator 接口的数据结构如下。

- Array
- Map
- Set
- String
- TypedArray(一个底层二进制数据缓存区的类数组，了解一下就阔以)
- 函数的 arguments 对象
- NodeList 对象

```
> let arr = ['a', 'b', 'c'];
  let iter = arr[Symbol.iterator]();
< undefined
> iter
< ▼Array Iterator {}
    ▼__proto__: Array Iterator
      ▶next: ƒ next()
        Symbol(Symbol.toStringTag): "Array Iterator"
      ▶__proto__: Object
> iter.next();
< ▶{value: "a", done: false}
> iter.next();
< ▶{value: "b", done: false}
> iter.next();
< ▶{value: "c", done: false}
> iter.next();
< ▶{value: undefined, done: true}
>
```

自行设置迭代器(Iterator)接口

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。

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。

```
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
}
```

自行设置迭代器(Iterator)接口

→是通过遍历器实现指针结构的例子

右侧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。

```
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
}
```

自行设置迭代器(Iterator)接口

右侧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
                    };
                }
            }
        };
    }
};
```

类数组的迭代器(Iterator)接口设置方式

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

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。

```
> HTMLCollection
ƒ HTMLCollection() { [native code] }
> HTMLCollection.prototype[Symbol.iterator]=Array.prototype[Symbol.iterator]
ƒ values() { [native code] }
> let o=document.getElementsByTagName("div");
undefined
> o
  HTMLCollection(11) [div#sidebar, div#content, div#disqus_thread, div#back_to_top, div#edit, div#loading, div#error, div#flip, div#pageup, div#pagedown, div.progress-indicator-2, sidebar: div#sidebar, content: div#content, disqus_thread: div#disqus_thread, back_to_top: div#back_to_top, edit: div#edit, …]
> for(let item of o){
    console.log(item);
  }
  <div id="sidebar">…</div>                                    VM333:2
  <div id="content">…</div>                                    VM333:2
  <div id="disqus_thread">…</div>                              VM333:2
  <div id="back_to_top" style="display: block;">back to top
  </div>                                                       VM333:2
  <div id="edit" style="display: block;">edit</div>            VM333:2
  <div id="loading" style="display: none;">Loading ...</div>   VM333:2
  <div id="error" style="display: none;">Oops! ... File not
  found!</div>                                                 VM333:2
  <div id="flip">…</div>                                       VM333:2
  <div id="pageup" style="display: inline-block;">上一章</div>  VM333:2
  <div id="pagedown" style="display: inline-block;">下一章</div> VM333:2
  <div class="progress-indicator-2" style="width: 34.1976%;">
  </div>                                                       VM333:2
```

类数组的迭代器(Iterator)接口设置方式

```
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'
}
```

```
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
}
```

类数组的迭代器(Iterator)接口设置方式

```
> let o=document.getElementsByTagName("div");
< undefined
> o[Symbol.iterator] = () => 1;
< () => 1
> for(let item of o){
      console.log(item);
  }
✕ ▶Uncaught TypeError: Result of the Symbol.iterator method VM101:1
  is not an object
      at <anonymous>:1:17
>
```

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

Iterator的常用领域

Iterator的常用领域

（1）解构赋值

对数组和 Set 结构进行解构赋值时，会默认调用set数据的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']
```

Iterator的常用领域

(3)字符串的 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 }
```

字符串是一个类似数组的对象，也原生具有 Iterator 接口

```
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"
```

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

Iterator的其他方法

Iterator的其他方法之return

遍历器对象除了具有next方法，还可以具有return方法和throw方法。

如果是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**。

```
> function setIerator(obj) {

    obj[Symbol.iterator] = function () {
        return {
            next() {
                return {
                    value: 1,
                    done: false
                };
            },
            return () {
                console.log("遍历中断");
                return {done:true}
            }
        };
    }
}

let o = document.getElementsByTagName("div")

setIerator(o);

for (let item of o) {
    console.log(item);
    break;
}
1                                                    VM68:24
遍历中断                                             VM68:12
undefined
```

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 setIerator(obj) {

    obj[Symbol.iterator] = function () {
        return {
            next() {
                return {
                    value: 1,
                    done: false
                };
            },
            return () {
                console.log("遍历中断");
            }
        };
    }
}

let o = document.getElementsByTagName("div")

setIerator(o);

for (let item of o) {
    console.log(item);
    break;
}
1                                                    VM79:24
遍历中断                                             VM79:12
▶Uncaught TypeError: Iterator result undefined is not an object   VM79:25
    at <anonymous>:25:5
>
```

return方法一定要返回一个对象,不然就要报错

Iterator的其他方法之throw

```
> function setIerator(obj) {

      obj[Symbol.iterator] = function () {
          return {
              next() {
                  return {
                      value: 1,
                      done: false
                  };
              },
              return () {
                  console.log("遍历中断");
                  return {done:true}
              }
          };
      }
  }

  let o = document.getElementsByTagName("div")

  setIerator(o);

  for (let item of o) {
      console.log(item);
      throw new Error("throw会中断遍历，并且还会执行一次报错,报错信息就
  是这里写的");
  }
  1                                                    VM112:24
  遍历中断                                              VM112:12
  ▶Uncaught Error: throw会中断遍历，并且还会执行一次报错,报错 VM112:25
  信息就是这里写的
      at <anonymous>:25:11
>
```

Iterator的其他方法之for… …of循环

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
}
```

左侧代码中，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。

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