08-第八章 ECMAScript6 Iterator 迭代器 generator 生成器

一、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。

模拟`next`方法

```
1.  var it = makeIterator([1,2,3,4]);
2.
3.          function makeIterator(array){
4.              var index = 0;
5.              return {
6.                  next:function(){
7.                      return index<array.length? {value:array[inde
    x++],done:false}:{value:undefined,done:true}
8.
9.
10.                 }
11.             }
12.         }
13.         console.log( it.next() );//{value: 1, done: false}
14.         console.log( it.next() );//{value: 2, done: false}
15.         console.log( it.next() );//{value: 3, done: false}
16.         console.log( it.next() );//{value: 4, done: false}
17.         console.log( it.next() );//{value: undefined, done: true}
```

二、默认Iterator 接口

terator 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）。

```
1.  var arr = ['a','b','c'];
2.          var iter = arr[Symbol.iterator]();
3.          console.log( iter.next());//{value: "a", done: false}
4.          console.log( iter.next());//{value: "b", done: false}
5.          console.log( iter.next());//{value: "c", done: false}
6.          console.log( iter.next());//{value: undefined, done: tru
    e}
```

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`。

`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

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`

三、自定义数组`Symbol.iterator`接口

```
var arr=['a','b','c'];
 arr[Symbol.iterator] = function(){
            var _this = this;
            var index = 0;
            return {
                next:function(){
                    return index<_this.length?{value:_this[index++]}:{done:true};
                }
            }
        }
        var iter = arr[Symbol.iterator]();
       console.log(  iter.next() );//a
       console.log(  iter.next() );//b
       console.log(  iter.next() );//c

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

四、自定义对象`Symbol.iterator`接口

```
var obj = {age:20,name:'二狗'};
        obj[Symbol.iterator] = function(){
            var index = 0;
            var _this = this;
            keys = Object.keys(this);
            _this.length = keys.length;
            return {
                next:function(){
                    return index<_this.length?{value:_this[keys[index++]]}:{done:true};
                }
            }
        }

  for (key of obj) {
      console.log(key);// 20 二狗
  }
```

五、字符串`Symbol.iterator`接口

```
var str = 'hello';
        var ite = str[Symbol.iterator]();
        console.log( ite.next() );//{value: "h", done: false}
        console.log( ite.next() );//{value: "e", done: false}
        console.log( ite.next() );//{value: "l", done: false}
        console.log( ite.next() );//{value: "l", done: false}
        console.log( ite.next() );//{value: "o", done: false}
         console.log( ite.next() );//{value: undefined, done: true}
```

```
var str = 'hello';
        str[Symbol.iterator] = function(){
            var _this = this;
            var index = 0;
            console.log('ok');
            return {
                next:function(){

                    return index<_this.length?{value:_this[index++]}:{done:true};
                },

            };

        }

        for (key of str) {
            console.log(key);
            break;
        }
```

六、for...of

> 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

数据结构遍历时，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`

```
var obj = {};
        var map = new Map([[{},1],[{},2]]);
        for ([key,val] of map) {
            console.log(key);
            //{}
            //1
            console.log(val);
            //{}
            //2
        }
```

ECMAScript6 generator 生成器

ECMAscript 6

一、Generator概念

> `Generator 函数` 是 ES6 提供的一种异步编程解决方案，Generator 函数是一个遍历器对象生成函数，会返回一个 `遍历器对象` 。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

Generator 函数是一个普通函数，但是有两个特征：

> 一是，function关键字与函数名之间有一个星号(`*`)；
> 二是，函数体内部使用 `yield` 表达式，定义不同的内部状态（ yield在英语里的意思就是“产出” ）。

```
function* generator(){
            yield 'hello';
            yield 'world';
            return 'ending';

        }
        var f = generator();
        console.log( f.next() );//{value: "hello", done: false}
        console.log( f.next() );//{value: "world", done: false}
        console.log( f.next() );//{value: "ending", done: true}
        console.log( f.next() );//{value: "undefined", done: tru
e}
```

上面代码定义了一个 `Generator` 函数generator，它内部有两个`yield`表达式（hello和world），即该函数有三个状态：`hello`，`world` 和 `return` 语句（结束执行）。

调用 Generator 函数后，`该函数并不执行`，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的`遍历器对象`（Iterator Object）

二、next方法

调用遍历器对象的`next`方法，使得指针移向下一个状态。

```
function* generator(){
            console.log('ok');
            yield 'hello';
            yield 'world';
            return 'ending';

        }
        var f = generator();
        console.log( f.next() );//ok {value: "hello", done: fals
e}
        console.log( f.next() );//{value: "world", done: false}
        console.log( f.next() );//{value: "ending", done: true}
        console.log( f.next() );//{value: "undefined", done: tru
e}
```

> 第一次调用next，Generator 函数开始执行，从头部开始，执行 `第一个yield语句` 。 `遇到第二个yield停下` 。返回一个对象（它的value属性就是当前yield表达式的值hello，）（ `done` 属性的值 `false` ）。

> 第二次调用next，Generator 函数开始执行，从上次yied结束为止开始，执行 `第二个yield语句` 。返回一个对象（它的value属性就是当前yield表达式的值world，）（ `done` 属性的值 `false` ）。

> 第三次调用next，Generator 函数开始执行，从上次yied结束为止开始。执行 `return语句` 。返回一个对象（它的value属性就是当前yield表达式的值ending，）（ `done` 属性的值 `true` ）。

> 第四次调用next，Generator 函数开始执行，从上次yied结束为止开始。后面没有语句，返回一个对象，（它的value属性就是 `undefined` ，）（ `done` 属性的值 `true` ）。

总结:

> `每次调用next，就执行一个yield语句` ，直到遇到下一个 `yield` 表达式（或 `return` 语句）为止。
> Generator 函数是 `分段执行` 的， `yield` 表达式是 `暂停执行` 的标记，而 `next` 方法可以恢复执行。
> `done` 属性是一个布尔值，表示是否遍历结束。

> return 后, `done` 的值会马上为 `true`

三、星号(*)

ES6 没有规定，function关键字与函数名之间的星号（ * ），写在哪个位置。这导致下面的写法都能通过。

```
        function*  generator(){}
        function * generator(){}
        function  *generator(){}
        function*generator(){}
var fn = function*(){
```

表达式写法

```
var fn = function* (){
     yield 'hello';
 }
 var f = fn();
 console.log(f.next() );//{value: "hello", done: false}
  console.log(f.next() );//{value: "hello", done: false}
```

三、yield 表达式

由于 Generator 函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种`可以暂停执行的函数`。`yield`表达式就是`暂停标志`。

（1）

> Generator 函数可以不用yield表达式，这时就变成了一个单纯的暂缓执行函数。就变成只有调用next方法时，函数fn才会执行

```
function* fn(){
  console.log('generator')
}
var f = fn();
```

(2)

> 另外需要注意，yield表达式只能用在 Generator 函数里面，用在其他地方都会报错。

```
function fn(){
            yield 'hello';//SyntaxError: Unexpected string
        }
```

(3)

> yield表达式如果用在另一个表达式之中，必须放在 `圆括号` 里面。

```
function* generator(){
           console.log('hello' +  (yield 'world') );
            console.log('hello' +  yield 'world' );//SyntaxError: Unexpected identifier

        }
        var g =generator();
        console.log( g.next() );//{value: "world", done: false}
```

注意'hello'并不会打印

四、yield语句返回值

```
function* fn(){
            var a = yield 10;//yield返回undefined
            console.log(a);//undefined

        }
        var f = fn();
        f.next();
        f.next();//undefined
```

五、与Iterator关系

`Symbol.iterator`方法，就是遍历器生成函数，调用该函数会返回该对象的一个遍历器对象，`Generator`函数就是遍历器生成函数

```
    function* generator(){
            yield 'hello';
            yield 'world';
            yield 'ending';
        }

        for (key of generator() ) {//需要调用generator
            console.log(key);//hello world ending
        }
```

因此可以把 Generator 赋值给对象的`Symbol.iterator`属性

```
var arr=['a','b','c'];
        arr[Symbol.iterator]=generator;
        function* generator(){
            yield this[0];
            yield this[1];
            yield this[2];
        }

        for (key of arr ) {
            console.log(key);//a b c
        }
console.log([...arr]);//["a", "b", "c"]
```

六、next 方法参数

> yield表达式本身默认返回 `undefined` 。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值

```
function* fn(){
            var a = yield 'ok';
            var b = yield 'ok';
            console.log(a);
            console.log(b);

        }
        var f = fn();
        f.next(10);
        f.next(20);//20
        f.next(30);//30
```

通过next方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

```
function* fn(){
            console.log('ok');
            console.log(`1.${yield}`);
            console.log(`2.${yield}`);
            return 'result';
        }
        var f = fn();
        f.next();//ok
        f.next(10);//1.10
        f.next(20);//2.20
```

七、`for...of`循环

`for...of` 循环可以自动遍历 Generator 函数时生成的Iterator对象，且此时不再需要调用next方法

```
function* fn(){
            yield 1;
            yield 2;
            yield 3;
            yield 4;
            return  5;
        }

        for (key of fn()) {
            console.log(key);//1 2 3 4
        }
```

这里需要注意，一旦next方法的返回对象的`done`属性为`true`，for…of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的5，不包括在`for...of`循环之中

利用iterator，可以写出遍历任意对象（`object`）的方法。原生的 JavaScript 对象没有遍历接口，无法使用for…of循环，通过 Generator 函数为它加上这个接口

```
function* generator(obj){
            let propArr = Object.keys(obj);
            for(key of propArr){
                yield [key,propArr[key]];
            }
        }
        var obj = {
            age:20,
            name:'二狗'
        }
        var g = generator(obj);
        console.log( g.next() );//{value: Array[2], done: false}
        console.log( g.next() );//{value: Array[2], done: false}
        console.log( g.next() );//{value: undefined, done: false}
```

上代码中`for`循环一样会暂停，而不是一次执行的，下面我们可以写一个对象的`Symbol.iterator`

```
function* generator(){
            let propArr = Object.keys(this);
            for(key of propArr){
                yield [key,this[key]];
            }
        }
        var obj = {age:20,name:'二狗',[Symbol.iterator]:generator
        }
        for ([key,val] of obj) {
           console.log(key,val);
        }
```

八、`throw` 抛出错误

> Generator 函数返回的遍历器对象，都有一个`throw`方法，可以在函数体外抛出错误，然后在`Generator`函数体内捕获。

```
 function* fn(){
            try{
                yield;
            }catch(e){
                console.log('内部',e);
            }

        }
       var f = fn();
       f.next();
       f.throw('抛出了一个错误');//Uncaught 抛出了一个错误
```

注意，不要混淆遍历器对象的throw方法和全局的throw命令。上面代码的错误，是用遍历器对象的throw方法抛出的，而不是用throw命令抛出的。后者只能被函数体外的catch语句捕获。

九、`return` 返回给定的value，且终结遍历

> return方法，可以返回给定的value，并且终结遍历 Generator 函数

```
function* fn(){
          yield 1;
             yield 2;
             yield 3;
             yield 4;
    }

    var f = fn();
    console.log( f.next() );//{value: 1, done: false}
    console.log( f.next() );//{value: 2, done: false}
    console.log( f.return('ok') );//{value: 'ok', done: false}
    console.log( f.next() );//{value: undefined, done: true}
```

十、next() throw() return的共同点

> `next()`、`throw()`、`return()`这三个方法本质上是同一件事，可以放在一起理解。它们的作用都是`让 Generator 函数恢复执行`，并且使用不同的语句`替换`yield表达式

`next()`是将yield表达式替换成一个值。
`throw()`是将yield表达式替换成一个throw语句。
`return()`是将yield表达式替换成一个return语句。

十一、yield* 表达式

> 如果在 Generator 函数内部，调用另一个 Generator 函数，默认情况下是没有效果的。即yield*用来在generator中调用`嵌套的generator`，

> 从语法角度看，如果yield表达式后面跟的是`一个遍历器对象`，需要在yield表达式后面加上星号，表明它返回的是一个遍历器对象。这被称为`yield*表达式`。

```
function* foo(){
            yield 'a';
            yield 'b';
        }
        function* bar(){
            yield 1;
            yield foo();
            yield 2;
        }
        for (key of bar()) {
            console.log(key);
            //1
            //foo {}
            //2
        }
```

foo和bar都是 Generator 函数，在bar里面调用foo，是不会有效果的

这个就需要用到`yield*`表达式

```
function* foo(){
      yield 'a';
         yield 'b';
    }
    function* bar(){
         yield 1;
         yield* foo();
         yield 2;
    }
    for (key of bar()) {
        console.log(key);
        //1
        //a
        //b
        //2
    }
```

`yield*`表达式，相对于了部署一个`for...of`循环。

```
function* concat(iter1,iter2){
        yield* iter1;
        yield* iter2;
       }

 //等同于

    function* concat(iter1,iter2){
         for(val of iter1){
             yield val;
         }
         for(val of iter2){
             yield val;
         }

    }
```

如果yield*后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员

```
var arr = ['a','b','c'];
        function* generator(arr){
         yield* arr;
        }
        for (key of generator(arr)) {
            console.log(key);
        }
```

实际上，任何数据结构只要有 Iterator 接口，就可以被yield*遍历。

```
var foo = (function*(){
   yield* 'hello';
    })();

    for (key of foo) {
       console.log(key);
    }
    //h e l l o
```

十二、作为对象的Generator函数

> 如果一个对象的属性是 Generator 函数，可以简写成下面的形式。

```
var obj ={
        generator:function *(){
            yield 1;
            yield 2;
            yield 3;
            yield 4;
        }
       }
       for (key of obj.generator()) {
           console.log(key);
           //1 2 3 4
       }
```

上面的写法等价于。

```
var obj ={
        *generator(){
            yield 1;
            yield 2;
            yield 3;
            yield 4;
        }
       }
       for (key of obj.generator()) {
           console.log(key);
           //1 2 3 4
       }
```

十三、Generator函数的this

> Generator 函数总是返回一个遍历器，这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的 `prototype` 对象上的方法，

（1）`this对象` 并非遍历器对象

```
function* generator(){
          console.log(this);//window
      }
      generator.prototype.hello = function(){
        console.log('hello');
      }
      let obj = generator();
      console.log( obj instanceof generator );//true
      obj.hello();//hello
      obj.next();//window
```

（2）Generator 函数也不能跟`new`命令一起用，会报错。

```
        function* gener(){}
           var obj = new gener();//TypeError: gener is not a cons
tructor
```

(3)

有没有办法让 Generator 函数返回一个正常的对象实例，既可以用`next`方法，又可以获得正常的this？

使用`call`方法绑定 Generator 函数内部的this

```
function* generator(){
            this.a = 1;
            yield this.b = 2;
            yield this.c = 3;
        }
        var obj = {};
        var g = generator.call(obj);

        console.log( g.next() );//{value: 2, done: false}
        console.log( g.next() );//{value: 3, done: false}
        console.log( g.a );//undefined
        console.log( obj.a );//1
```

(4)一个办法就是将obj换成F.`prototype`,将这两个对象统一

```
function* generator(){
            this.a = 1;
            yield this.b = 2;
            yield this.c = 3;
        }

        var g = generator.call(generator.prototype);

        console.log( g.next() );//{value: 2, done: false}
        console.log( g.next() );//{value: 3, done: false}
        console.log( g.a );//1
        console.log( g.b );//1
```

再将F改成构造函数，就可以对它执行`new`命令了。

```
function* generator(){
            this.a = 1;
            yield this.b = 2;
            yield this.c = 3;
        }

        function Fn(){
            return generator.call(generator.prototype);

        }
        var f = new Fn();
        console.log( f.next() );//{value: 2, done: false}
```