



第37课：生成器Generator 函数的语法

主讲老师：万章



目录

Generator函数的概念

yield表达式

for...of循环与生成器

函数的报错处理



Generator函数的概念

Generator函数的概念

Generator 函数有多种理解角度。首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

Generator 函数是一个普通函数，但是有几个特征。

- function关键字与函数名之间有一个星号；
- 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”）；
- 执行 Generator 函数会返回一个遍历器对象，可调用.next方法进行遍历，遍历的结果是Generator函数里面通过yield或是return定义的值，按顺序输出

```
> function* helloWorldGenerator() {  
  yield 'hello';  
  yield 'world';  
  return 'ending';  
}  
  
var hw = helloWorldGenerator();  
< undefined  
> hw.next()  
< ▶ {value: "hello", done: false}  
> hw.next()  
< ▶ {value: "world", done: false}  
> hw.next()  
< ▶ {value: "ending", done: true}  
>
```

Generator函数的概念

Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，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(如下图)，也就是上一章介绍的遍历器对象（Iterator Object）。

下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。每次调用next方法，内部指针从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。

```
> function* helloWorldGenerator() {  
  yield 'hello';  
  yield 'world';  
  return 'ending';  
}  
  
var hw = helloWorldGenerator();  
< undefined  
> hw.next()  
< ▶ {value: "hello", done: false}  
> hw.next()  
< ▶ {value: "world", done: false}  
> hw.next()  
< ▶ {value: "ending", done: true}  
>
```



成员1

成员2

成员3

成员4

成员5

成员6

成员7

Generator函数的概念

```
> function* helloWorldGenerator() {  
  yield 'hello';  
  yield 'world';  
}  
  
var hw = helloWorldGenerator();  
< undefined  
> hw.next()  
< ▶ {value: "hello", done: false}  
> hw.next()  
< ▶ {value: "world", done: false}  
> hw.next()  
< ▶ {value: undefined, done: true}  
>
```

```
> function* helloWorldGenerator() {  
  yield 'hello';  
  return "万章";  
  yield 'world';  
}  
  
var hw = helloWorldGenerator();  
< undefined  
> hw.next()  
< ▶ {value: "hello", done: false}  
> hw.next()  
< ▶ {value: "万章", done: true}  
> hw.next()  
< ▶ {value: undefined, done: true}  
>
```

注意!!!

如果return语句之后还有yield语句，那么return之后所有yield语句都将不会被遍历



成员1

成员2

成员3

return
成员4

成员5

成员6

成员7

Generator函数的格式

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 * foo(x, y) { ... }  
function *foo(x, y) { ... }  
function* foo(x, y) { ... }  
function*foo(x, y) { ... }
```

由于 Generator 函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在function关键字后面。



yield表达式

yield表达式的运行步骤

遍历器对象的next方法的运行逻辑如图所示。

```
> function* helloWorldGenerator() {  
  yield 'hello';  
  return "万章";  
  yield 'world';  
}
```

```
var hw = helloWorldGenerator();
```

```
< undefined
```

```
> hw.next()
```

```
< ▶ {value: "hello", done: false}
```

```
> hw.next()
```

```
< ▶ {value: "万章", done: true}
```

```
> hw.next()
```

```
< ▶ {value: undefined, done: true}
```

```
>
```

1) 遇到yield表达式，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

2) 下一次调用next方法时，再继续往下执行，直到遇到下一个yield表达式。

3) 如果没有再遇到新的yield表达式，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。结束遍历

4) 如果该函数没有return语句，则返回的对象的value属性值为undefined。

yield表达式的惰性求值(Lazy Evaluation)

yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

```
function* gen() {  
  yield 123 + 456;  
}
```

上面代码中，yield后面的表达式123 + 456，不会立即求值，只会在next方法将指针移到这一句时，才会求值。

yield表达式的暂缓执行

```
function* f() {  
  console.log('执行了! ');  
}  
  
var generator = f();  
  
setTimeout(function () {  
  generator.next()  
}, 2000);
```

```
> function* helloWorldGenerator() {  
  console.log("这个语句会和 hello一起输出 \n");  
  yield 'hello';  
  return "万章";  
  yield 'world';  
}
```

```
var hw = helloWorldGenerator();
```

```
< undefined
```

```
> hw.next()
```

```
这个语句会和 hello一起输出
```

```
< ▶ {value: "hello", done: false}
```

```
> hw.next()
```

```
< ▶ {value: "万章", done: true}
```

```
> |
```

Generator 函数可以不用yield表达式,这样当我们在调用这个Generator函数返回的迭代器对象的next方法时就会执行Generator函数内的代码,就好似是这个函数被临时暂停了,然后得到某个指令之后再去执行

注意:这条代码也只会执行一次

yield表达式的使用规范

- 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。
- 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
- yield表达式用作函数参数或放在赋值表达式的右边，可以不加括号。

```
(function (){  
  yield 1;  
})();  
// SyntaxError: Unexpected number
```

```
function* demo() {  
  console.log('Hello' + yield); // SyntaxError  
  console.log('Hello' + yield 123); // SyntaxError  
  
  console.log('Hello' + (yield)); // OK  
  console.log('Hello' + (yield 123)); // OK  
}
```

```
function* demo() {  
  foo(yield 'a', yield 'b'); // OK  
  let input = yield; // OK  
}
```

yield表达式的使用场景

可以把 Generator 赋值给对象的Symbol.iterator属性，从而使得该对象具有 Iterator 接口。

Generator 函数执行后，返回一个遍历器对象。该对象本身也具有Symbol.iterator属性，执行后返回自身。

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

```
function* gen(){  
  // some code  
}  
  
var g = gen();  
  
g[Symbol.iterator]() === g  
// true
```

yield表达式的使用场景之next函数参数

yield表达式本身没有返回值，或者说总是返回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* f() {  
  for(var i = 0; true; i++) {  
    var reset = yield i;  
    if(reset) { i = -1; }  
  }  
}  
  
var g = f();  
  
g.next() // { value: 0, done: false }  
g.next() // { value: 1, done: false }  
g.next(true) // { value: 0, done: false }
```

如果next方法没有参数，每次运行到yield表达式，变量reset的值总是undefined。

当next方法带一个参数true时，变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

注意，要牢牢记住，每次调用返回的迭代器对象时，运行到yield就会终止的，下次调用next方法才会继续往下运行，所以这个for循环每次调用一次next方法就会只执行一次循环

yield表达式的使用场景之next函数参数

```
function* foo(x) {  
  var y = 2 * (yield (x + 1));  
  var z = yield (y / 3);  
  return (x + y + z);  
}
```

```
var a = foo(5);  
a.next() // Object{value:6, done:false}  
a.next() // Object{value:NaN, done:false}  
a.next() // Object{value:NaN, done:true}
```

```
var b = foo(5);  
b.next() // { value:6, done:false }  
b.next(12) // { value:8, done:false }  
b.next(13) // { value:42, done:true }
```

第一次运行返回只会执行
(x+1), 所以返回的
是5+1=6

第二次调用, 上次的yield返回的
是undefined, 所以继续执
行Y=2*undefined, 所以 y是NaN, 再返
回下一个yield后面的 (Y/2)自然就
是(NaN/2)结果就是返回的依然是NaN

第三次运行, x是5,y
是NaN, z是undefined,
结果就是
5+NaN+undefined -
>NaN

第一次运行没有传递参数的时候

第一次运行返回只会执行
(x+1), 所以返回的是5+1 6

第二次调用, 传入了参数12, 就相当于第一
条yield语句(yield(x+1))返回的值是
12, 那么继续往下计算, y=2*12 => y就
是24, 再往下, 返回第二天yield语句后面
的(y/3), 那么就是24/3 8

第三次运行, 传入的参数
是13, 那么相当表达式 (yield(y/3))的值是13,
那么z就是13, 最后返回的
就是5+24+13 42

第二次运行有传递参数的时候

V8 引擎直接忽略第一次使用next方法时的参数, 只有从第二次使用next方法开始, 参数才是有效的。从语义上讲, 第一个next方法用来启动遍历器对象, 所以不用带有参数。

yield表达式的使用场景之next函数参数

```
function* dataConsumer() {  
  console.log('Started');  
  console.log(`1. ${yield}`);  
  console.log(`2. ${yield}`);  
  return 'result';  
}
```

```
let genObj = dataConsumer();  
genObj.next();  
// Started  
genObj.next('a')  
// 1. a  
genObj.next('b')  
// 2. b
```

注意，要牢牢记住：

- 每次调用返回的迭代器对象时，运行到yield就会终止的，下次调用next方法才会继续往下运行
- 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。



for...of循环与生成器


for...of循环与生成器

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

```
function* foo() {  
  yield 1;  
  yield 2;  
  yield 3;  
  yield 4;  
  yield 5;  
  return 6;  
}  
  
for (let v of foo()) {  
  console.log(v);  
}  
// 1 2 3 4 5
```

左侧代码使用for...of循环，依次显示 5 个yield表达式的值。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，所以左侧代码的return语句返回的6，不包括在for...of循环之中。

```
> function* foo() {  
  yield 1;  
  yield 2;  
  yield 3;  
  yield 4;  
  yield 5;  
  return 6;  
}  
< undefined  
> let o=foo()  
< undefined  
> o.next()  
< {value: 1, done: false}  
> o.next()  
< {value: 2, done: false}  
> o.next()  
< {value: 3, done: false}  
> o.next()  
< {value: 4, done: false}  
> o.next()  
< {value: 5, done: false}  
> o.next()  
< {value: 6, done: true}  
> |
```



举个栗子：斐波那契数列的实现



0 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

p	c
---	---



我举个栗子

```
> function* fibonacci() {  
  let [prev, curr] = [0, 1];  
  for (;;) {  
    yield curr;  
    [prev, curr] = [curr, prev + curr];  
  }  
}  
  
for (let n of fibonacci()) {  
  if (n > 1000) break;  
  console.log(n);  
}
```

2 1

2

3

5

8

13

21

34

55

89

144

233

377

610

987

< undefined

>

举个栗子:for...of循环实现对象的遍历

```
function* objectEntries(obj) {  
  let propKeys = Reflect.ownKeys(obj);  
  
  for (let propKey of propKeys) {  
    yield [propKey, obj[propKey]];  
  }  
}  
  
let jane = { first: 'Jane', last: 'Doe' };  
  
for (let [key, value] of objectEntries(jane)) {  
  console.log(`${key}: ${value}`);  
}  
// first: Jane  
// last: Doe
```

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，就可以用了。



```
function* objectEntries() {  
  let propKeys = Object.keys(this);  
  
  for (let propKey of propKeys) {  
    yield [propKey, this[propKey]];  
  }  
}  
  
let jane = { first: 'Jane', last: 'Doe' };  
  
jane[Symbol.iterator] = objectEntries;  
  
for (let [key, value] of jane) {  
  console.log(`${key}: ${value}`);  
}  
// first: Jane  
// last: Doe
```

另一种写法是，将 Generator 函数加到对象的Symbol.iterator属性上面。

这种方式更方便一些，建议使用

其他的遍历与生成器的关系

```
function* numbers () {  
  yield 1  
  yield 2  
  return 3  
  yield 4  
}  
  
// 扩展运算符  
[...numbers()] // [1, 2]  
  
// Array.from 方法  
Array.from(numbers()) // [1, 2]  
  
// 解构赋值  
let [x, y] = numbers();  
x // 1  
y // 2  
  
// for...of 循环  
for (let n of numbers()) {  
  console.log(n)  
}  
// 1  
// 2
```

- for...of循环
- 扩展运算符 (...)
- 解构赋值
- Array.from方法

等内部调用的，都是遍历器接口。这意味着，它们都可以将 Generator 函数返回的 Iterator 对象，作为参数。



函数的报错处理

try, catch语句

```
> function sayName(s) {  
    console.log("hello");  
}  
  
sayname("万章"); //注意, 这个函数名称错误了  
  
console.log("这一行代码无法正常执行");
```

✖ ▶ Uncaught ReferenceError: sayname is not defined
at <anonymous>:5:1

[VM65:5](#)

```
> function sayName(s){  
    console.log("hello");  
}  
  
try{  
    sayname("万章");//注意, 这个函数名称错误了  
    console.log("这一行错误代码后的代码是没法执行的");  
}  
catch(err){  
    console.log("函数执行出错, 错误信息为: \n");  
    console.log(err);  
}  
  
console.log("这一行代码正常执行");
```

函数执行出错, 错误信息为:	←	VM95:10
ReferenceError: sayname is not defined at <anonymous>:6:5	←	VM95:11
这一行代码正常执行	←	VM95:14

try :定义一个代码块, 这个代码块就和普通代码一样正常执行,但是一旦当里面的代码出错了, 那么就会终止执行, 并且向catch语句中传入报错信息,但是不会影响try代码块以外代码的执行

catch 接受try代码块中的报错信息,并执行catch中的代码

throw语句

```
1 function sayName(s) {
2     console.log("hello");
3 }
4
5 try {
6     throw "你要的啊, 函数啊, 并不存在 ... .";
7 } catch (err) {
8     console.log("函数执行出错, 错误信息为: \n");
9     console.log(err);
10 }
11
12 console.log("这一行代码正常执行");
```

```
> function sayName(s) {
    console.log("hello");
}

try {
    throw "你要的啊, 函数啊, 并不存在 ... .";
} catch (err) {
    console.log("函数执行出错, 错误信息为: \n");
    console.log(err);
}

console.log("这一行代码正常执行");
```

函数执行出错, 错误信息为:	VM69:8
你要的啊, 函数啊, 并不存在	VM69:9
这一行代码正常执行	VM69:12

throw 语句允许创建自定义错误提示。

不过要留意, 必须是在**throw**语句被执行了, 才会向catch提交通过throw自定义的err数据

finally语句

```
> function sayName(s) {  
  console.log("hello");  
}  
  
try {  
  sayname("wanz");//函数名称错误  
} catch (err) {  
  console.log("函数执行出错, 错误信息为: \n");  
  console.log(err);  
}  
  
finally{  
  console.log("这一行代码正常执行");  
}
```

函数执行出错, 错误信息为:	VM114:8
ReferenceError: sayname is not defined at <anonymous>:6:4	VM114:9
这一行代码正常执行	VM114:13

一样



```
> function sayName(s) {  
  console.log("hello");  
}  
  
try {  
  sayname("wanz");//函数名称错误  
} catch (err) {  
  console.log("函数执行出错, 错误信息为: \n");  
  console.log(err);  
}
```

console.log("这一行代码正常执行");	
函数执行出错, 错误信息为:	VM84:8
ReferenceError: sayname is not defined at <anonymous>:6:4	VM84:9
这一行代码正常执行	VM84:12



finally 语句允许在 try 和 catch 之后执行代码，不管try和catch的运行结果咋样，都会执行finally里的代码。

不过，这玩意没啥卵用

错误信息对象

通过Error的构造器可以创建一个错误对象。当运行时错误产生时，Error的实例对象会被抛出

Error 对象属性

属性	描述
name	设置或返回错误名
message	设置或返回错误消息（一条字符串）

error 的 name 属性可返回六个不同的值：

错误名	描述
EvalError	已在 eval() 函数中发生的错误
RangeError	已发生超出数字范围的错误
ReferenceError	已发生非法引用
SyntaxError	已发生语法错误
TypeError	已发生类型错误
URIError	在 encodeURIComponent() 中已发生的错误

```
function sayName(s) {
  console.log("hello");
}

try {
  sayname("wanz");//函数名称错误
} catch (err) {
  console.log("函数执行出错,错误信息为: \n");
  console.log(err.name);
  console.log(err.message);
}
```

```
> function sayName(s) {
  console.log("hello");
}

try {
  sayname("wanz");//函数名称错误
} catch (err) {
  console.log("函数执行出错,错误信息为: \n");
  console.log(err.name);
  console.log(err.message);
}
```

函数执行出错,错误信息为:

ReferenceError

sayname is not defined

VM144:8

VM144:9

VM144:10