

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证



目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

其他

- 源码
- 修订历史
- 反馈意见

Generator 函数的语法

- 1.简介
- 2.next 方法的参数
- 3.for...of 循环
- 4.Generator.prototype.throw()
- 5.Generator.prototype.return()
- 6.next()、throw()、return() 的共同点
- 7.yield* 表达式
- 8.作为对象属性的 Generator 函数

上一章

下一章

1. 简介

基本概念

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍 Generator 函数的语法和 API，它的异步编程应用请看《Generator 函数的异步应用》一章。

Generator 函数有多种理解角度。语法上，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。

执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。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。

形式上，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* helloWorldGenerator() {  
  yield 'hello';  
  yield 'world';  
  return 'ending';  
}  
  
var hw = helloWorldGenerator();
```

上面代码定义了一个 Generator 函数 `helloWorldGenerator`，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）。

然后，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，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。也就是说，每次调用 `next` 方法，内部指针就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。换言之，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。

```
hw.next()  
// { value: 'hello', done: false }  
  
hw.next()  
// { value: 'world', done: false }  
  
hw.next()  
// { value: 'ending', done: true }  
  
hw.next()  
// { value: undefined, done: true }
```

上面代码一共调用了四次 `next` 方法。

第一次调用，Generator 函数开始执行，直到遇到第一个 `yield` 表达式为止。`next` 方法返回一个对象，它的 `value` 属性就是当前 `yield` 表达式的值 `hello`，`done` 属性的值 `false`，表示遍历还没有结束。

第二次调用，Generator 函数从上次 `yield` 表达式停下的地方，一直执行到下一个 `yield` 表达式。`next` 方法返回的对象的 `value` 属性就是当前 `yield` 表达式的值 `world`，`done` 属性的值 `false`，表示遍历还没有结束。

第三次调用，Generator 函数从上次 `yield` 表达式停下的地方，一直执行到 `return` 语句（如果没有 `return` 语句，就执行到函数结束）。`next` 方法返回的对象的 `value` 属性，就是紧跟在 `return` 语句后面的表达式的值（如果没有 `return` 语句，则 `value` 属性的值为 `undefined`），`done` 属性的值 `true`，表示遍历已经结束。

第四次调用，此时 Generator 函数已经运行完毕，`next` 方法返回对象的 `value` 属性为 `undefined`，`done` 属性为 `true`。以后再调用 `next` 方法，返回的都是这个值。

总结一下，调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的 `next` 方法，就会返回一个有着 `value` 和 `done` 两个属性的对象。`value` 属性表示当前的内部状态的值，是 `yield` 表达式后面那个表达式的值；`done` 属性是一个布尔值，表示是否遍历结束。

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。本书也采用这种写法。

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。

遍历器对象的 `next` 方法的运行逻辑如下。

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

需要注意的是，`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“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` 表达式与 `return` 语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到 `yield`，函数暂停执行，下一次再从该位置继续向后执行。这有点像“记忆”的功能。一个函数里面，只能执行一次（或者说

一个) `return` 语句，但是可以执行多次（或者说多个） `yield` 表达式。正常函数只能返回一个值，因为只能执行一次 `return`；

`Generator` 函数可以返回一系列的值，因为可以有任意多个 `yield`。从另一个角度看，也可以说 `Generator` 生成了一系列的值，这也就是它的名称的来历（英语中，`generator` 这个词是“生成器”的意思）。

`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。

```
function* f() {
  console.log('执行了! ')
}

var generator = f();

setTimeout(function () {
  generator.next()
}, 2000);
```

上面代码中，函数 `f` 如果是普通函数，在为变量 `generator` 赋值时就会执行。但是，函数 `f` 是一个 `Generator` 函数，就变成只有调用 `next` 方法时，函数 `f` 才会执行。

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

上面代码在一个普通函数中使用 `yield` 表达式，结果产生一个句法错误。

下面是另外一个例子。

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a) {
  a.forEach(function (item) {
    if (typeof item !== 'number') {
      yield* flat(item);
    } else {
      yield item;
    }
  });
};

for (var f of flat(arr)){
  console.log(f);
}
```

上面代码也会产生句法错误，因为 `forEach` 方法的参数是一个普通函数，但是在里面使用了 `yield` 表达式（这个函数里面还使用了 `yield*` 表达式，详细介绍见后文）。一种修改方法是改用 `for` 循环。

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a) {
  var length = a.length;
  for (var i = 0; i < length; i++) {
    var item = a[i];
    if (typeof item !== 'number') {
      yield* flat(item);
    } else {
      yield item;
    }
  }
};
```

```
for (var f of flat(arr)) {  
  console.log(f);  
}  
// 1, 2, 3, 4, 5, 6
```

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

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

与 **Iterator** 接口的关系

上一章说过，任意一个对象的 `Symbol.iterator` 方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于 **Generator** 函数就是遍历器生成函数，因此可以把 **Generator** 赋值给对象的 `Symbol.iterator` 属性，从而使得该对象具有 **Iterator** 接口。

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

上面代码中，**Generator** 函数赋值给 `Symbol.iterator` 属性，从而使得 `myIterable` 对象具有了 **Iterator** 接口，可以被 `...` 运算符遍历了。

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。

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

上面代码中，`gen` 是一个 **Generator** 函数，调用它会生成一个遍历器对象 `g`。它的 `Symbol.iterator` 属性，也是一个遍历器对象生成函数，执行后返回它自己。

2. next 方法的参数

`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 }
```

上面代码先定义了一个可以无限运行的 **Generator** 函数 `f`，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。

这个功能有很重要的语法意义。**Generator** 函数从暂停状态到恢复运行，它的上下文状态（**context**）是不变的。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。

再看一个例子。

```
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 }
```

上面代码中，第二次运行 `next` 方法的时候不带参数，导致 `y` 的值等于 `2 * undefined`（即 `NaN`），除以 `3` 以后还是 `NaN`，因此返回对象的 `value` 属性也等于 `NaN`。第三次运行 `Next` 方法的时候不带参数，所以 `z` 等于 `undefined`，返回对象的 `value` 属性等于 `5 + NaN + undefined`，即 `NaN`。

如果向 `next` 方法提供参数，返回结果就完全不一样了。上面代码第一次调用 `b` 的 `next` 方法时，返回 `x+1` 的值 `6`；第二次调用 `next` 方法，将上一次 `yield` 表达式的值设为 `12`，因此 `y` 等于 `24`，返回 `y / 3` 的值 `8`；第三次调用 `next` 方法，将上一次 `yield` 表达式的值设为 `13`，因此 `z` 等于 `13`，这时 `x` 等于 `5`，`y` 等于 `24`，所以 `return` 语句的值等于 `42`。

注意，由于 `next` 方法的参数表示上一个 `yield` 表达式的返回值，所以在第一次使用 `next` 方法时，传递参数是无效的。**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。

再看一个通过 `next` 方法的参数，向 **Generator** 函数内部输入值的例子。

```
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
```

上面代码是一个很直观的例子，每次通过 **next** 方法向 **Generator** 函数输入值，然后打印出来。

如果想要第一次调用 **next** 方法时，就能够输入值，可以在 **Generator** 函数外面再包一层。

```
function wrapper(generatorFunction) {
  return function (...args) {
    let generatorObject = generatorFunction(...args);
    generatorObject.next();
    return generatorObject;
  };
}

const wrapped = wrapper(function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
});

wrapped().next('hello!')
// First input: hello!
```

上面代码中，**Generator** 函数如果不用 **wrapper** 先包一层，是无法第一次调用 **next** 方法，就输入参数的。

3. 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* 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
```

上面代码使用 **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，所以上面代码的 **return** 语句返回的 **6**，不包括在 **for...of** 循环之中。

下面是一个利用 **Generator** 函数和 **for...of** 循环，实现非同步遍历的例子

```
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);
}
```

从上面代码可见，使用 `for...of` 语句时不需要使用 `next` 方法。

利用 `for...of` 循环，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，就可以用了。

```
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
```

上面代码中，对象 `jane` 原生不具备 **Iterator** 接口，无法用 `for...of` 遍历。这时，我们通过 Generator 函数 `objectEntries` 为它加上遍历器接口，就可以用 `for...of` 遍历了。加上遍历器接口的另一种写法是，将 Generator 函数加到对象的 `Symbol.iterator` 属性上面。

```
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
```

除了 `for...of` 循环以外，扩展运算符（`...`）、解构赋值和 `Array.from` 方法内部调用的，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。

```
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

```

4. Generator.prototype.throw()

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。

```

var g = function* () {
  try {
    yield;
  } catch (e) {
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 内部捕获 a
// 外部捕获 b

```

上面代码中，遍历器对象 `i` 连续抛出两个错误。第一个错误被 Generator 函数体内的 `catch` 语句捕获。`i` 第二次抛出错误，由于 Generator 函数内部的 `catch` 语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了 Generator 函数体，被函数体外的 `catch` 语句捕获。

`throw` 方法可以接受一个参数，该参数会被 `catch` 语句接收，建议抛出 `Error` 对象的实例。

```

var g = function* () {
  try {
    yield;
  } catch (e) {
    console.log(e);
  }
};

var i = g();
i.next();
i.throw(new Error('出错了!'));
// Error: 出错了! (...)

```

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。

```
var g = function* () {
  while (true) {
    try {
      yield;
    } catch (e) {
      if (e !== 'a') throw e;
      console.log('内部捕获', e);
    }
  }
};

var i = g();
i.next();

try {
  throw new Error('a');
  throw new Error('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 [Error: a]
```

上面代码之所以只捕获了 `a`，是因为函数体外的 `catch` 语句块，捕获了抛出的 `a` 错误以后，就不会再继续 `try` 代码块里面剩余的语句了。

如果 `Generator` 函数内部没有部署 `try...catch` 代码块，那么 `throw` 方法抛出的错误，将被外部 `try...catch` 代码块捕获。

```
var g = function* () {
  while (true) {
    yield;
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 a
```

上面代码中，`Generator` 函数 `g` 内部没有部署 `try...catch` 代码块，所以抛出的错误直接被外部 `catch` 代码块捕获。

如果 `Generator` 函数内部和外部，都没有部署 `try...catch` 代码块，那么程序将报错，直接中断执行。

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();
g.throw();
// hello
// Uncaught undefined
```

上面代码中，`g.throw` 抛出错误以后，没有任何 `try...catch` 代码块可以捕获这个错误，导致程序报错，中断执行。

`throw` 方法抛出的错误要被内部捕获，前提是必须至少执行过一次 `next` 方法。

```
function* gen() {
  try {
    yield 1;
  } catch (e) {
    console.log('内部捕获');
  }
}

var g = gen();
g.throw(1);
// Uncaught 1
```

上面代码中，`g.throw(1)` 执行时，`next` 方法一次都没有执行过。这时，抛出的错误不会被内部捕获，而是直接在外部抛出，导致程序出错。这种行为其实很好理解，因为第一次执行 `next` 方法，等同于启动执行 **Generator** 函数的内部代码，否则 **Generator** 函数还没有开始执行，这时 `throw` 方法抛错只可能抛出在函数外部。

`throw` 方法被捕获以后，会附带执行下一条 `yield` 表达式。也就是说，会附带执行一次 `next` 方法。

```
var gen = function* gen(){
  try {
    yield console.log('a');
  } catch (e) {
    // ...
  }
  yield console.log('b');
  yield console.log('c');
}

var g = gen();
g.next() // a
g.throw() // b
g.next() // c
```

上面代码中，`g.throw` 方法被捕获以后，自动执行了一次 `next` 方法，所以会打印 `b`。另外，也可以看到，只要 **Generator** 函数内部部署了 `try...catch` 代码块，那么遍历器的 `throw` 方法抛出的错误，不影响下一次遍历。

另外，`throw` 命令与 `g.throw` 方法是无关的，两者互不影响。

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();

try {
  throw new Error();
} catch (e) {
  g.next();
}
// hello
// world
```

上面代码中，`throw` 命令抛出的错误不会影响到遍历器的状态，所以两次执行 `next` 方法，都进行了正确的操作。

这种函数体内捕获错误的机制，大大方便了对错误的处理。多个 `yield` 表达式，可以只用一个 `try...catch` 代码块来捕获错误。如果使用回调函数的写法，想要捕获多个错误，就不得不为每个 `yield` 语句，现在只在 **Generator** 函数内部写一次 `catch` 语句

就可以了。

Generator 函数体外抛出的错误，可以在函数体内捕获；反过来，Generator 函数体内抛出的错误，也可以被函数体外的 `catch` 捕获。

```
function* foo() {
  var x = yield 3;
  var y = x.toUpperCase();
  yield y;
}

var it = foo();

it.next(); // { value:3, done:false }

try {
  it.next(42);
} catch (err) {
  console.log(err);
}
```

上面代码中，第二个 `next` 方法向函数体内传入一个参数 `42`，数值是没有 `toUpperCase` 方法的，所以会抛出一个 `TypeError` 错误，被函数体外的 `catch` 捕获。

一旦 Generator 执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。如果此后还调用 `next` 方法，将返回一个 `value` 属性等于 `undefined`、`done` 属性等于 `true` 的对象，即 JavaScript 引擎认为这个 Generator 已经运行结束了。

```
function* g() {
  yield 1;
  console.log('throwing an exception');
  throw new Error('generator broke!');
  yield 2;
  yield 3;
}

function log(generator) {
  var v;
  console.log('starting generator');
  try {
    v = generator.next();
    console.log('第一次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第二次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第三次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  console.log('caller done');
}

log(g());
// starting generator
// 第一次运行next方法 { value: 1, done: false }
// throwing an exception
// 捕捉错误 { value: 1, done: false }
// 第三次运行next方法 { value: undefined, done: true }
// caller done
```

上面代码一共三次运行 `next` 方法，第二次运行的时候会抛出错误，然后第三次运行的时候，`Generator` 函数就已经结束了，不再执行下去了。

5. Generator.prototype.return()

`Generator` 函数返回的遍历器对象，还有一个 `return` 方法，可以返回给定的值，并且终结遍历 `Generator` 函数。

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

var g = gen();

g.next()          // { value: 1, done: false }
g.return('foo')   // { value: "foo", done: true }
g.next()          // { value: undefined, done: true }
```

上面代码中，遍历器对象 `g` 调用 `return` 方法后，返回值的 `value` 属性就是 `return` 方法的参数 `foo`。并且，`Generator` 函数的遍历就终止了，返回值的 `done` 属性为 `true`，以后再调用 `next` 方法，`done` 属性总是返回 `true`。

如果 `return` 方法调用时，不提供参数，则返回值的 `value` 属性为 `undefined`。

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

var g = gen();

g.next()          // { value: 1, done: false }
g.return()         // { value: undefined, done: true }
```

如果 `Generator` 函数内部有 `try...finally` 代码块，且正在执行 `try` 代码块，那么 `return` 方法会推迟到 `finally` 代码块执行完再执行。

```
function* numbers () {
  yield 1;
  try {
    yield 2;
    yield 3;
  } finally {
    yield 4;
    yield 5;
  }
  yield 6;
}

var g = numbers();
g.next() // { value: 1, done: false }
g.next() // { value: 2, done: false }
g.return(7) // { value: 4, done: false }
g.next() // { value: 5, done: false }
g.next() // { value: 7, done: true }
```

上面代码中，调用 `return` 方法后，就开始执行 `finally` 代码块，然后等到 `finally` 代码块执行完，再执行 `return` 方法。

6. 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。

```
const g = function* (x, y) {
  let result = yield x + y;
  return result;
};

const gen = g(1, 2);
gen.next(); // Object {value: 3, done: false}

gen.next(1); // Object {value: 1, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = 1;
```

上面代码中，第二个 `next(1)` 方法就相当于将 `yield` 表达式替换成一个值 `1`。如果 `next` 方法没有参数，就相当于替换成 `undefined`。

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

```
gen.throw(new Error('出错了')); // Uncaught Error: 出错了
// 相当于将 let result = yield x + y
// 替换成 let result = throw(new Error('出错了'));
```

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

```
gen.return(2); // Object {value: 2, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = return 2;
```

7. 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。

```
function* foo() {
  yield 'a';
  yield 'b';
}

function* bar() {
  yield 'x';
  foo();
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// "x"
// "y"
```

上面代码中，`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，用来在一个 Generator 函数里面执行另一个 Generator 函数。

```
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  yield 'a';
  yield 'b';
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  for (let v of foo()) {
    yield v;
  }
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// "x"
// "a"
// "b"
// "y"
```

再来看一个对比的例子。

```
function* inner() {
  yield 'hello!';
}

function* outer1() {
  yield 'open';
  yield inner();
  yield 'close';
}

var gen = outer1()
gen.next().value // "open"
gen.next().value // 返回一个遍历器对象
gen.next().value // "close"

function* outer2() {
  yield 'open'
  yield* inner()
  yield 'close'
}

var gen = outer2()
gen.next().value // "open"
gen.next().value // "hello!"
gen.next().value // "close"
```

上面例子中，`outer2` 使用了 `yield*`，`outer1` 没使用。结果就是，`outer1` 返回一个遍历器对象，`outer2` 返回该遍历器对象的内部值。

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。

```

let delegatedIterator = (function* () {
  yield 'Hello!';
  yield 'Bye!';
})();

let delegatingIterator = (function* () {
  yield 'Greetings!';
  yield* delegatedIterator;
  yield 'Ok, bye.';
})();

for(let value of delegatingIterator) {
  console.log(value);
}
// "Greetings!
// "Hello!"
// "Bye!"
// "Ok, bye."

```

上面代码中，`delegatingIterator` 是代理者，`delegatedIterator` 是被代理者。由于 `yield* delegatedIterator` 语句得到的值，是一个遍历器，所以要用星号表示。运行结果就是使用一个遍历器，遍历了多个 **Generator** 函数，有递归的效果。

`yield*` 后面的 **Generator** 函数（没有 `return` 语句时），等同于在 **Generator** 函数内部，部署一个 `for...of` 循环。

```

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

// 等同于

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

```

上面代码说明，`yield*` 后面的 **Generator** 函数（没有 `return` 语句时），不过是 `for...of` 的一种简写形式，完全可以用后者替代前者。反之，在有 `return` 语句时，则需要用 `var value = yield* iterator` 的形式获取 `return` 语句的值。

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。

```

function* gen(){
  yield* ["a", "b", "c"];
}

gen().next() // { value:"a", done:false }

```

上面代码中，`yield` 命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器对象。

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

```

let read = (function* () {
  yield 'hello';
  yield* 'hello';
})();

read.next().value // "hello"
read.next().value // "h"

```


上面代码中，`yield` 表达式返回整个字符串，`yield*` 语句返回单个字符。因为字符串具有 `Iterator` 接口，所以被 `yield*` 遍历。

如果被代理的 `Generator` 函数有 `return` 语句，那么就可以向代理它的 `Generator` 函数返回数据。

```
function* foo() {
  yield 2;
  yield 3;
  return "foo";
}

function* bar() {
  yield 1;
  var v = yield* foo();
  console.log("v: " + v);
  yield 4;
}

var it = bar();

it.next()
// {value: 1, done: false}
it.next()
// {value: 2, done: false}
it.next()
// {value: 3, done: false}
it.next();
// "v: foo"
// {value: 4, done: false}
it.next()
// {value: undefined, done: true}
```

上面代码在第四次调用 `next` 方法的时候，屏幕上会有输出，这是因为函数 `foo` 的 `return` 语句，向函数 `bar` 提供了返回值。

再看一个例子。

```
function* genFuncWithReturn() {
  yield 'a';
  yield 'b';
  return 'The result';
}

function* logReturned(genObj) {
  let result = yield* genObj;
  console.log(result);
}

[...logReturned(genFuncWithReturn())]
// The result
// 值为 [ 'a', 'b' ]
```

上面代码中，存在两次遍历。第一次是扩展运算符遍历函数 `logReturned` 返回的遍历器对象，第二次是 `yield*` 语句遍历函数 `genFuncWithReturn` 返回的遍历器对象。这两次遍历的效果是叠加的，最终表现为扩展运算符遍历函数 `genFuncWithReturn` 返回的遍历器对象。所以，最后的数据表达式得到的值等于 `['a', 'b']`。但是，函数 `genFuncWithReturn` 的 `return` 语句的返回值 `The result`，会返回给函数 `logReturned` 内部的 `result` 变量，因此会有终端输出。

`yield*` 命令可以很方便地取出嵌套数组的所有成员。

```
function* iterTree(tree) {
  if (Array.isArray(tree)) {
    for(let i=0; i < tree.length; i++) {
      yield* iterTree(tree[i]);
    }
  } else {
    yield tree;
  }
}
```

```

}

const tree = [ 'a', ['b', 'c'], ['d', 'e'] ];

for(let x of iterTree(tree)) {
  console.log(x);
}
// a
// b
// c
// d
// e

```

下面是一个稍微复杂的例子，使用 `yield*` 语句遍历完全二叉树。

```

// 下面是二叉树的构造函数，
// 三个参数分别是左树、当前节点和右树
function Tree(left, label, right) {
  this.left = left;
  this.label = label;
  this.right = right;
}

// 下面是中序（inorder）遍历函数。
// 由于返回的是一个遍历器，所以要用generator函数。
// 函数体内采用递归算法，所以左树和右树要用yield*遍历
function* inorder(t) {
  if (t) {
    yield* inorder(t.left);
    yield t.label;
    yield* inorder(t.right);
  }
}

// 下面生成二叉树
function make(array) {
  // 判断是否为叶节点
  if (array.length == 1) return new Tree(null, array[0], null);
  return new Tree(make(array[0]), array[1], make(array[2]));
}
let tree = make([['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]);

// 遍历二叉树
var result = [];
for (let node of inorder(tree)) {
  result.push(node);
}

result
// ['a', 'b', 'c', 'd', 'e', 'f', 'g']

```

8. 作为对象属性的 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。

```

let obj = {
  * myGeneratorMethod() {
    ...
  }
};

```

上面代码中，`myGeneratorMethod` 属性前面有一个星号，[上一章](#) [生成器函数](#) [下一节](#) `generator` 函数。

它的完整形式如下，与上面的写法是等价的。

```
let obj = {
  myGeneratorMethod: function* () {
    // ...
  }
};
```

9. Generator 函数的 `this`

Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的 `prototype` 对象上的方法。

```
function* g() {}

g.prototype.hello = function () {
  return 'hi!';
};

let obj = g();

obj instanceof g // true
obj.hello() // 'hi!'
```

上面代码表明，Generator 函数 `g` 返回的遍历器 `obj`，是 `g` 的实例，而且继承了 `g.prototype`。但是，如果把 `g` 当作普通的构造函数，并不会生效，因为 `g` 返回的总是遍历器对象，而不是 `this` 对象。

```
function* g() {
  this.a = 11;
}

let obj = g();
obj.next();
obj.a // undefined
```

上面代码中，Generator 函数 `g` 在 `this` 对象上面添加了一个属性 `a`，但是 `obj` 对象拿不到这个属性。

Generator 函数也不能跟 `new` 命令一起用，会报错。

```
function* F() {
  yield this.x = 2;
  yield this.y = 3;
}

new F()
// TypeError: F is not a constructor
```

上面代码中，`new` 命令跟构造函数 `F` 一起使用，结果报错，因为 `F` 不是构造函数。

那么，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`？

下面是一个变通方法。首先，生成一个空对象，使用 `call` 方法绑定 Generator 函数内部的 `this`。这样，构造函数调用以后，这个空对象就是 Generator 函数的实例对象了。

```
function* F() {
  this.a = 1;
  yield this.b = 2;
```

```

    yield this.c = 3;
  }
  var obj = {};
  var f = F.call(obj);

  f.next(); // Object {value: 2, done: false}
  f.next(); // Object {value: 3, done: false}
  f.next(); // Object {value: undefined, done: true}

  obj.a // 1
  obj.b // 2
  obj.c // 3

```

上面代码中，首先是 **F** 内部的 **this** 对象绑定 **obj** 对象，然后调用它，返回一个 **Iterator** 对象。这个对象执行三次 **next** 方法（因为 **F** 内部有两个 **yield** 表达式），完成 **F** 内部所有代码的运行。这时，所有内部属性都绑定在 **obj** 对象上了，因此 **obj** 对象也就成了 **F** 的实例。

上面代码中，执行的是遍历器对象 **f**，但是生成的对象实例是 **obj**，有没有办法将这两个对象统一呢？

一个办法就是将 **obj** 换成 **F.prototype**。

```

function* F() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
var f = F.call(F.prototype);

f.next(); // Object {value: 2, done: false}
f.next(); // Object {value: 3, done: false}
f.next(); // Object {value: undefined, done: true}

f.a // 1
f.b // 2
f.c // 3

```

再将 **F** 改成构造函数，就可以对它执行 **new** 命令了。

```

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

function F() {
  return gen.call(gen.prototype);
}

var f = new F();

f.next(); // Object {value: 2, done: false}
f.next(); // Object {value: 3, done: false}
f.next(); // Object {value: undefined, done: true}

f.a // 1
f.b // 2
f.c // 3

```

10. 含义

Generator 与状态机

Generator 是实现状态机的最佳结构。比如，下面的 `clock` 函数就是一个状态机。

```
var ticking = true;
var clock = function() {
  if (ticking)
    console.log('Tick!');
  else
    console.log('Tock!');
  ticking = !ticking;
}
```

上面代码的 `clock` 函数一共有两种状态（`Tick` 和 `Tock`），每运行一次，就改变一次状态。这个函数如果用 **Generator** 实现，就是下面这样。

```
var clock = function* () {
  while (true) {
    console.log('Tick!');
    yield;
    console.log('Tock!');
    yield;
  }
};
```

上面的 **Generator** 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量 `ticking`，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。**Generator** 之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

Generator 与协程

协程（`coroutine`）是一种程序运行的方式，可以理解成“协作的线程”或“协作的函数”。协程既可以用单线程实现，也可以用多线程实现。前者是一种特殊的子例程，后者是一种特殊的线程。

（1）协程与子例程的差异

传统的“子例程”（`subroutine`）采用堆栈式“后进先出”的执行方式，只有当调用的子函数完全执行完毕，才会结束执行父函数。协程与之不同，多个线程（单线程情况下，即多个函数）可以并行执行，但是只有一个线程（或函数）处于正在运行的状态，其他线程（或函数）都处于暂停态（`suspended`），线程（或函数）之间可以交换执行权。也就是说，一个线程（或函数）执行到一半，可以暂停执行，将执行权交给另一个线程（或函数），等到稍后收回执行权的时候，再恢复执行。这种可以并行执行、交换执行权的线程（或函数），就称为协程。

从实现上看，在内存中，子例程只使用一个栈（`stack`），而协程是同时存在多个栈，但只有一个栈是在运行状态，也就是说，协程是以多占用内存为代价，实现多任务的并行。

（2）协程与普通线程的差异

不难看出，协程适合用于多任务运行的环境。在这个意义上，它与普通的线程很相似，都有自己的执行上下文、可以分享全局变量。它们的不同之处在于，同一时间可以有多个线程处于运行状态，但是运行的协程只能有一个，其他协程都处于暂停状态。此外，普通的线程是抢先式的，到底哪个线程优先得到资源，必须由运行环境决定，但是协程是合作式的，执行权由协程自己分配。

由于 **JavaScript** 是单线程语言，只能保持一个调用栈。引入协程以后，每个任务可以保持自己的调用栈。这样做的最大好处，就是抛出错误的时候，可以找到原始的调用栈。不至于像异步操作的回调函数那样，一旦出错，原始的调用栈早就结束。

Generator 函数是 ES6 对协程的实现，但属于不完全实现。Generator 函数被称为“半协程”（semi-coroutine），意思是只有 Generator 函数的调用者，才能将程序的执行权还给 Generator 函数。如果是完全执行的协程，任何函数都可以让暂停的协程继续执行。

如果将 Generator 函数当作协程，完全可以将多个需要互相协作的任务写成 Generator 函数，它们之间使用 **yield** 表达式交换控制权。

Generator 与上下文

JavaScript 代码运行时，会产生一个全局的上下文环境（**context**，又称运行环境），包含了当前所有的变量和对象。然后，执行函数（或块级代码）的时候，又会在当前上下文环境的上层，产生一个函数运行的上下文，变成当前（**active**）的上下文，由此形成一个上下文环境的堆栈（**context stack**）。

这个堆栈是“后进先出”的数据结构，最后产生的上下文环境首先执行完成，退出堆栈，然后再执行完成它下层的上下文，直至所有代码执行完成，堆栈清空。

Generator 函数不是这样，它执行产生的上下文环境，一旦遇到 **yield** 命令，就会暂时退出堆栈，但是并不消失，里面的所有变量和对象会冻结在当前状态。等到对它执行 **next** 命令时，这个上下文环境又会重新加入调用栈，冻结的变量和对象恢复执行。

```
function* gen() {
  yield 1;
  return 2;
}

let g = gen();

console.log(
  g.next().value,
  g.next().value,
);
```

上面代码中，第一次执行 **g.next()** 时，Generator 函数 **gen** 的上下文会加入堆栈，即开始运行 **gen** 内部的代码。等遇到 **yield 1** 时，**gen** 上下文退出堆栈，内部状态冻结。第二次执行 **g.next()** 时，**gen** 上下文重新加入堆栈，变成当前的上下文，重新恢复执行。

11. 应用

Generator 可以暂停函数执行，返回任意表达式的值。这种特点使得 Generator 有多种应用场景。

（1）异步操作的同步化表达

Generator 函数的暂停执行的效果，意味着可以把异步操作写在 **yield** 表达式里面，等到调用 **next** 方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在 **yield** 表达式下面，反正要等到调用 **next** 方法时再执行。所以，Generator 函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

```
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
  hideLoadingScreen();
}

var loader = loadUI();
// 加载UI
```

```
loader.next()
```

```
// 卸载UI
```

```
loader.next()
```

上面代码中，第一次调用 `loadUI` 函数时，该函数不会执行，仅返回一个遍历器。下一次对该遍历器调用 `next` 方法，则会显示 `Loading` 界面（`showLoadingScreen`），并且异步加载数据（`loadUIDataAsynchronously`）。等到数据加载完成，再一次使用 `next` 方法，则会隐藏 `Loading` 界面。可以看到，这种写法的好处是所有 `Loading` 界面的逻辑，都被封装在一个函数，按部就班非常清晰。

Ajax 是典型的异步操作，通过 `Generator` 函数部署 Ajax 操作，可以用同步的方式表达。

```
function* main() {
  var result = yield request("http://some.url");
  var resp = JSON.parse(result);
  console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

var it = main();
it.next();
```

上面代码的 `main` 函数，就是通过 Ajax 操作获取数据。可以看到，除了多了一个 `yield`，它几乎与同步操作的写法完全一样。注意，`makeAjaxCall` 函数中的 `next` 方法，必须加上 `response` 参数，因为 `yield` 表达式，本身是没有值的，总是等于 `undefined`。

下面是另一个例子，通过 `Generator` 函数逐行读取文本文件。

```
function* numbers() {
  let file = new FileReader("numbers.txt");
  try {
    while(!file.eof) {
      yield parseInt(file.readLine(), 10);
    }
  } finally {
    file.close();
  }
}
```

上面代码打开文本文件，使用 `yield` 表达式可以手动逐行读取文件。

（2）控制流管理

如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```

采用 `Promise` 改写上面的代码。

[上一章](#)

[下一章](#)

```

Promise.resolve(step1)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // Do something with value4
  }, function (error) {
    // Handle any error from step1 through step4
  })
  .done();

```

上面代码已经把回调函数，改成了直线执行的形式，但是加入了大量 **Promise** 的语法。**Generator** 函数可以进一步改善代码运行流程。

```

function* longRunningTask(value1) {
  try {
    var value2 = yield step1(value1);
    var value3 = yield step2(value2);
    var value4 = yield step3(value3);
    var value5 = yield step4(value4);
    // Do something with value4
  } catch (e) {
    // Handle any error from step1 through step4
  }
}

```

然后，使用一个函数，按次序自动执行所有步骤。

```

scheduler(longRunningTask(initialValue));

function scheduler(task) {
  var taskObj = task.next(task.value);
  // 如果Generator函数未结束，就继续调用
  if (!taskObj.done) {
    task.value = taskObj.value
    scheduler(task);
  }
}

```

注意，上面这种做法，只适合同步操作，即所有的 **task** 都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。如果要控制异步的操作流程，详见后面的《异步操作》一章。

下面，利用 **for...of** 循环会自动依次执行 **yield** 命令的特性，提供一种更一般的控制流管理的方法。

```

let steps = [step1Func, step2Func, step3Func];

function* iterateSteps(steps){
  for (var i=0; i< steps.length; i++){
    var step = steps[i];
    yield step();
  }
}

```

上面代码中，数组 **steps** 封装了一个任务的多个步骤，**Generator** 函数 **iterateSteps** 则是依次为这些步骤加上 **yield** 命令。

将任务分解成步骤之后，还可以将项目分解成多个依次执行的任务。

```

let jobs = [job1, job2, job3];

function* iterateJobs(jobs){
  for (var i=0; i< jobs.length; i++){
    var job = jobs[i];
    yield* iterateSteps(job.steps);
  }
}

```



```
}  
}
```

上面代码中，数组 `jobs` 封装了一个项目的多个任务，Generator 函数 `iterateJobs` 则是依次为这些任务加上 `yield*` 命令。

最后，就可以用 `for...of` 循环一次性依次执行所有任务的所有步骤。

```
for (var step of iterateJobs(jobs)){  
  console.log(step.id);  
}
```

再次提醒，上面的做法只能用于所有步骤都是同步操作的情况，不能有异步操作的步骤。如果想要依次执行异步的步骤，必须使用后面的《异步操作》一章介绍的方法。

`for...of` 的本质是一个 `while` 循环，所以上面的代码实质上执行的是下面的逻辑。

```
var it = iterateJobs(jobs);  
var res = it.next();  
  
while (!res.done){  
  var result = res.value;  
  // ...  
  res = it.next();  
}
```

（3）部署 **Iterator** 接口

利用 Generator 函数，可以在任意对象上部署 **Iterator** 接口。

```
function* iterEntries(obj) {  
  let keys = Object.keys(obj);  
  for (let i=0; i < keys.length; i++) {  
    let key = keys[i];  
    yield [key, obj[key]];  
  }  
}  
  
let myObj = { foo: 3, bar: 7 };  
  
for (let [key, value] of iterEntries(myObj)) {  
  console.log(key, value);  
}  
  
// foo 3  
// bar 7
```

上述代码中，`myObj` 是一个普通对象，通过 `iterEntries` 函数，就有了 **Iterator** 接口。也就是说，可以在任意对象上部署 `next` 方法。

下面是一个对数组部署 **Iterator** 接口的例子，尽管数组原生具有这个接口。

```
function* makeSimpleGenerator(array){  
  var nextIndex = 0;  
  
  while(nextIndex < array.length){  
    yield array[nextIndex++];  
  }  
}  
  
var gen = makeSimpleGenerator(['yo', 'ya']);
```

```
gen.next().value // 'yo'
gen.next().value // 'ya'
gen.next().done  // true
```

（4）作为数据结构

Generator 可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为 Generator 函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

```
function* doStuff() {
  yield fs.readFile.bind(null, 'hello.txt');
  yield fs.readFile.bind(null, 'world.txt');
  yield fs.readFile.bind(null, 'and-such.txt');
}
```

上面代码就是依次返回三个函数，但是由于使用了 Generator 函数，导致可以像处理数组那样，处理这三个返回的函数。

```
for (task of doStuff()) {
  // task是一个函数，可以像回调函数那样使用它
}
```

实际上，如果用 ES5 表达，完全可以用数组模拟 Generator 的这种用法。

```
function doStuff() {
  return [
    fs.readFile.bind(null, 'hello.txt'),
    fs.readFile.bind(null, 'world.txt'),
    fs.readFile.bind(null, 'and-such.txt')
  ];
}
```

上面的函数，可以用一模一样的 `for...of` 循环处理！两相一比较，就不难看出 Generator 使得数据或者操作，具备了类似数组的接口。

留言

124 Comments

ECMAScript 6 入门

1 Login ▾

♥ Recommend 17

🐦 Tweet

f Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



vision57 • a year ago

阮老师，我觉得描述throw的部分可以精简、归纳下。

我的理解，`next`/`throw`/`return`三个方法，^{上一章}为generator函数复执行，效果上的不同在于，`it.next(value)`是

将generator里面的yield表达式 *替换* 为一个值, `it.throw(value)`是将yield表达式替换为一个throw语句, `it.return(value)`是将yield表达式替换为一个return语句。

基于这个理解, generator内部的各种`return`/`throw`行为就按一般情况来预期。如果`return`/`throw`出现在try内, 那么有捕获就捕获, 有finally就执行finally; 如果最后`return`/`throw`到了函数外部, 那么这个generator自然就变成`complete`状态, 而外部caller自然就应该收到返回值或抛出异常。

其实这里面唯一需要提到的特殊情况, 就是generator尚未开始之前和complete之后, 如果尝试执行`next`/`throw`/`return`, 会是什么效果。

35 ^ | v • Reply • Share ›



ruanyf Mod → vision57 • a year ago

你这个理解好, 我要更新一下文档。

2017-09-05 19:35 GMT+08:00 Disqus <notifications@disqus.net>:

1 ^ | v • Reply • Share ›



youxiachai • 3 years ago

```
var clock = function*(_) {
```

```
  while (true) {
```

```
    yield _;
```

```
    console.log("Tick!");
```

```
    yield _;
```

```
    console.log("Tock!");
```

```
  }
```

```
};
```

console.log 的顺序有问题吧?

```
var clock = function*(_) {
```

```
  while (true) {
```

[see more](#)

14 ^ | v • Reply • Share ›



ruanyf Mod → youxiachai • 3 years ago

谢谢指出, 已经改过来了。

^ | v • Reply • Share ›



Hongye Qi • 4 years ago

论证生成器的iterator属性指向其本身的代码, 通过阅读规范中的类图, 是否应该是这样: `function* f(){};let a = new f();console.log(a[Symbol.iterator]() === a);`

2 ^ | v • Reply • Share ›



Hongye Qi → Hongye Qi • 4 years ago

[上一章](#)

[下一章](#)

或者`let a = f(); a[Symbol.iterator]() === a;function*`本身并不是generator, 他只是generator的构造器



2 ^ | v • Reply • Share ›



1 ^ | v • Reply • Share ›



1 ^ | v • Reply • Share ›



^ | v • Reply • Share ›



1 ^ | v • Reply • Share ›



^ | v • Reply • Share ›



kiinlam • 5 years ago

阮兄，第3节里的第二个例子，`function`后面的星号放错位置了吧。

原文：`function *numbers() {`

应该是：`function* numbers`

1 ^ | v • Reply • Share ›



Albert Yu ➔ kiinlam • 4 years ago

实际上星号的位置不重要，在前在后在中间甚至没有空格都是可以的。

^ | v • Reply • Share ›



ruanyf Mod ➔ kiinlam • 5 years ago

谢谢指出，已经修改了。

^ | v • Reply • Share ›



沈志敏 • 8 months ago

阮老师，关于 `throw` 的场景下，如果一开始就`throw`的话，内部捕获也是无效的

```
function* gen() {
  try {
    yield 1
  } catch (e) {
    console.log('内部捕获')
  }
}

var g = gen();
try {
  g.throw(1)
} catch (e) {
  console.log('外部捕获')
  g.next() // { value: undefined, done: true }
}
```

^ | v • Reply • Share ›



ruanyf Mod ➔ 沈志敏 • 8 months ago

谢谢指出，确实如此。

我应该说明一下，第一次执行 `next` 方法 等同于启动 `Generator` 函数的内部代码。所以，`throw` 方法抛错被内部捕获，必须执行过一次 `Generator` 方法。

我已经改了原文了。

^ | v • Reply • Share ›



wu andrew • 9 months ago

第6小节：

原文：

`gen.throw(new Error('出错了'));` // Uncaught Error: 出错了

// 相当于将 `let result = yield x + y`

// 替换成 `let result = throw(new Error('出错了'));`

上一章

下一章

以下这段代码执行就会无法捕获错误：

```
const g = function* (x, y) {  
  try{  
    var result = yield x + y;  
  }catch(err){  
    console.log(err);  
  }  
}
```

```
return result;  
};
```

```
const gen = g(1, 2);  
gen.throw(new Error('err'));
```

我的理解：

我觉得原文不够严谨，`gen.throw`不完全是`yield`替换成`throw`，`throw`前必须先执行一次`gen.next()`，不然无法捕获错误，我觉得是`yield`必须执行到所在`try`代码块，然后`throw`异常才可以被捕获。

希望阮老师或各位大牛指正下我的理解对不对

^ | v • Reply • Share ›



ruanyf Mod ➔ wu andrew • 9 months ago

是的，必须执行一次 `next()`，然后 `gen.throw` 才有效果。

我改一下。

1 ^ | v • Reply • Share ›



Cyclone • 9 months ago

“网友 [vision57](#) 提出”，建议删除，显得突兀。是否可以另起一章来感谢那些意见提供者的名字？

^ | v • Reply • Share ›



ruanyf Mod ➔ Cyclone • 9 months ago

这个想法很好，我以前没想到，后面就改掉，这么处理。

^ | v • Reply • Share ›



khflx • a year ago

阮兄, 你在描述“Generator函数的 `this`”时例子有误

```
function* g() {  
  this.a = 11;  
}
```

```
let obj = g();  
obj.a // undefined
```

你想用这个例子来表明Generator函数的`this`，但是你这里只是返回一个遍历器而没有执行，虽然结果是对的

^ | v • Reply • Share ›



ruanyf Mod ➔ khflx • a year ago

谢谢指出，已经改正。

^ | v • Reply • Share ›



宋九重 • a year ago

[上一章](#)

[下一章](#)



阮大师,我还有一个问题想问一下,16章第10节的'控制流管理'部分,您举了一个例子:

采用 Promise 改写上面的代码。

```
Promise.resolve(step1)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // Do something with value4
  }, function (error) {
    // Handle any error from step1 through step4
  })
  .done();
```

我认为step1,step2...都代表了一个耗时的函数,需要上一个函数的返回值做为参数才能继续运行,那么是不是将Promise.resolve(step1)改为Promise.resolve(step1())更加准确呢? 谢谢!

补充一点,这个例子的下一个例子中:

[see more](#)

^ | v • Reply • Share ›



ruanyf Mod ➔ 宋九重 • a year ago

这一段其实是伪码,表示多个步骤。实际编程中,必须看 step1、step2、.....如何定义,再决定调用的写法。

^ | v • Reply • Share ›



宋九重 ➔ ruanyf • a year ago

好的,知道了,谢谢!

^ | v • Reply • Share ›



宋九重 • a year ago

阮大师,我有一个疑问:

16章第10节的'异步操作的同步化表达'部分,第一个例子:

```
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
  hideLoadingScreen();
}
var loader = loadUI();
// 加载UI
loader.next()

// 卸载UI
loader.next()
```

您写道:'...这实际上等同于不需要写回调函数了,因为异步操作的后续操作可以放在yield表达式下面,反正要等到调用next方法时再执行... 等到数据加载完成,再一次使用next方法,则会隐藏Loading界面...'

我的问题是调用函数者是怎么判断这个异步的loadUIDataAsynchronously()已经完成了呢? 他应该在哪个时刻去调用这第二次next方法呢? 如果没有收到返回就去调用第二次loader.next()方法卸载UI,这样不是会有问

题吗? 谢谢!

^ | v • Reply • Share ›



ruanyf Mod ➔ 宋九重 • a year ago

这里是简化问题，为了说明 **Generator** 函数的作用。实际上，是要通过外部手段，才能知道 **UI** 是否加载成功。比如，**UI** 加载成功后，出现了“卸载”按钮，用户点击该按钮，触发第二次调用 **next** 方法。

2017-08-09 9:35 GMT+08:00 Disqus <notifications@disqus.net>:

^ | v • Reply • Share ›



宋九重 ➔ ruanyf • a year ago

我知道了,这里只是为了介绍**Generator**函数的用法,谢谢!

^ | v • Reply • Share ›



ruiyu lin • 2 years ago

```
var arr = [1, [[2, 3], 4], [5, 6]];
```

```
var flat = function* (a) {  
  var length = a.length;  
  for (var i = 0; i < length; i++) {  
    var item = a[i];  
    if (typeof item !== 'number') {  
      yield* flat(item);  
    } else {  
      yield item;  
    }  
  }  
};
```

```
for (var f of flat(arr)) {  
  console.log(f);  
}  
// 1, 2, 3, 4, 5, 6
```

前面说到：“需要注意的是，**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”。。这里没有**next()**，为什么可以执行？？

^ | v • Reply • Share ›



iCyMind ➔ ruiyu lin • a year ago

for (var f of flat(arr)) 会自动帮你 next， 直到 done 为止

^ | v • Reply • Share ›



常修 • 2 years ago

指针对象即**iterator**对象？？

^ | v • Reply • Share ›



常修 • 2 years ago

这是一个新篇章跟买的书的章节安排不太一样，但是很精彩。人们添加的新语法，都是期待着**js**有着更大的作用。

需要继续加油啊。

[上一章](#)

[下一章](#)

^ | v • Reply • Share ›



绫绮飒 • 2 years ago

```
function* genFuncWithReturn() {  
  yield 'a';  
  yield 'b';  
  return 'The result';  
}  
  
function* logReturned(genObj) {  
  let result = yield* genObj;  
  console.log(result);  
}
```

```
[...logReturned(genFuncWithReturn())]  
// The result  
// 值为 ['a', 'b']
```

阮老师，我想请问了一下 为何使用...运算符的输出结果是 The result 优先而不是 ['a', 'b'] 优先

因为按照我的理解 应该是先 遍历 genFuncWithReturn 返回的遍历器对象
我试着使用了 for...of 和 next() 跑了一下,如我所想



see more

^ | v • Reply • Share ›



ruanyf Mod → 绫绮飒 • 2 years ago

你的理解是对的，确实是先产生 **a** 和 **b**。但是，这里是 [...logReturned(genFuncWithReturn())]，**a** 和 **b** 只是返回值，并不会打印出来，第一个打印出来的是 the result。

^ | v • Reply • Share ›



绫绮飒 → ruanyf • 2 years ago

哦 这么一说还真是

^ | v • Reply • Share ›



yacent • 2 years ago

阮老师，指出文档一点点小错误，就是第六小节 yield* 标题下方，generator 单词拼写的错误

generater → generator

^ | v • Reply • Share ›



ruanyf Mod → yacent • 2 years ago

谢谢指出，已经改正。

^ | v • Reply • Share ›



Hongying Wang • 2 years ago

如果 yield 语句后面的参数，是一个具有遍历器接口的对象，yield 会遍历这个对象，再往下执行。

没理解！！求解释

^ | v • Reply • Share ›



Hongying Wang → Hongying Wang • 2 years ago

截图



^ | v • Reply • Share ›



ruanyf Mod → Hongying Wang • 2 years ago

原文确实写得不清楚，其实指的是，返回一个都是**Promise**成员的数组，必须等每一个成员执行完，才会往下执行。我已经把这部分改写了，将异步的部分拿掉了，你可以再看一下。

^ | v • Reply • Share ›



zhuscat • 3 years ago

关于**Generator**中的**this**，举的例子是不是有点问题。

```
function* g() {  
  this.a = 11;  
}
```

obj = g();

这里就算没有*，**this.a**也是**undefined**吧

另外是

再将**F**改成构造函数，就可以对它执行**new**命令了。

a，**b**，**c**都直接添加到原型上面，之后**new**一个实例属性就都是已经有的，这样做有什么实际意义吗？

^ | v • Reply • Share ›



ruanyf Mod → zhuscat • 3 years ago

如果不是严格模式，就会赋值成功。

^ | v • Reply • Share ›



swei • 3 years ago

你好，下面两行代码在51.0.2704.106 m 版本下的**Chrome**和 6.7.7版本下的**babel**都运行不了，是为什么呀？

```
console.log('Hello' + (yield));  
console.log('Hello' + (yield 123));
```

^ | v • Reply • Share ›



ruanyf Mod → swei • 3 years ago

yield 要放在 **generator** 之中。

^ | v • Reply • Share ›



Youxiang Lu • 3 years ago

阮老师，不太明白下面代码有啥好处

```
function* g(){  
  try {  
    var a = yield foo('a');  
    var b = yield foo('b');
```

```

var c = yield foo('c');
} catch (e) {
console.log(e);
}

console.log(a, b, c);
}

```

感觉撇开客户调用代码来说，谈这段代码的优势一点意义都没有。

外部怎么调用才是重点。

说到底，你用前面一段完整代码来和这里一段不完整代码比较来说明generator的优势，完全没可比性。

此外，感觉这个例子混淆了yield和await的用法，如果你把这里的yield变成await就可以理解了。

此外，a,b,c的值不是foo调用的返回值，如果外部next代码没有传入值，都是undefined，完全不符合逻辑。再次说明复杂度都在外部代码上了。

^ | v • Reply • Share ›



ruanyf Mod ➔ Youxiang Lu • 3 years ago

这个例子确实有问题，删掉了。

^ | v • Reply • Share ›



tanson • 3 years ago

```

function taskA(){
"use strict";
console.log("taskA");
return "A";
}
function taskB(){
"use strict";
console.log("taskB");
return "B";
}
function* parallelTasks() {
let [resultA, resultB] = yield [
taskA(),
taskB()
];
console.log(resultA, resultB);
}
按该方法运行，报这个错误
//taskA
//taskB
//{ value: [ 'A', 'B' ], done: false }
//TypeError: taskB is not a function

```

^ | v • Reply • Share ›



ranger zhao • 3 years ago

用generator实现状态机个人感觉有点勉强，虽然说generator本身带有状态，但是感觉他的执行是线性的，而状态机根据输入可以产生回路，要在generator函数中实现回路只能使用switch或者ifelse。而使用状态机的初衷不就是为了消灭它们么？

[上一章](#)

[下一章](#)

^ | v • Reply • Share ›



Youxiang Lu → ranger zhao • 3 years ago

个人感觉也是，**Generator**本身就是用状态机实现，说用它来实现状态机确实有点怪。

还有就是，能不能不要翻译成**Generator**函数，就应该是**GeneratorFunction**，还有**GeneratorFunction**返回的是**Generator**对象，而不是**Iterator**对象，而**Generator**实现了**Iterator**接口而已，所以能够有**Iterator**的效果。

1 ^ | v • Reply • Share ›

Load more comments

ALSO ON ECMASCRIPT 6 入门

编程风格

51 comments • 4 years ago

Tom Bian — mark

Class 的继承

38 comments • 2 years ago

Francie Who —

数组的扩展

71 comments • 5 years ago

liyun — 不好意思，之前是我理解错了浅拷贝和深拷贝的概念

let和const命令

222 comments • 5 years ago

Seaborn Lee —

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy

