**Generator**函数是ES6提供的一种异步编程解决方案。

Generator函数有多种理解角度。从语法上，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语句，定义不同的内部状态（yield语句在英语里的意思就是“产出”）

function\* helloWorldGenerator() {

yield'hello';

yield'world';

return'ending';

yield'world111';

yield'world222';

}

var hw = helloWorldGenerator();

上面代码定义了一个Generator函数helloWorldGenerator，它内部有两个yield语句“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，也就是遍历器对象（Iterator Object）以后会介绍到。

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语句（或return语句）为止。换言之，Generator函数是分段执行的，yield语句是暂停执行的标记，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语句为止。next方法返回一个对象，它的value属性就是当前yield语句的值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语句停下的地方，一直执行到下一个yield语句。next方法返回的对象的value属性就是当前yield语句的值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语句停下的地方，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语句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。

2 yield语句

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语句就是暂停标志。

遍历器对象的next方法的运行逻辑如下。

（1） 遇到yield语句，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语句。

（3）如果没有再遇到新的yield语句，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语句后面的表达式，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语句与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，而return语句不具备位置记忆的功能。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语句。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（在英语中，generator这个词是“生成器”的意思）。

Generator函数可以不用yield语句，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语句不能用在普通函数中，否则会报错。

(function (){

yield1;

})()

// SyntaxError: Unexpected number

上面代码在一个普通函数中使用yield语句，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语句（这个函数里面还使用了yield\*语句，这里可以不用理会，详细说明见后文）。一种修改方法是改用for循环。

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

var flat = function\* (a) {

varlength = a.length;

for(var i = 0; i < length; i++) {

varitem = 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语句如果用在一个表达式之中，必须放在圆括号里面。

console.log('Hello' + yield); // SyntaxError

console.log('Hello' + yield 123); //SyntaxError

console.log('Hello' + (yield)); // OK

console.log('Hello' + (yield 123)); // OK

yield语句用作函数参数或赋值表达式的右边，可以不加括号。

foo(yield 'a', yield 'b'); // OK

let input = yield; // OK

与Iterator接口的关系

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\* () {

yield1;

yield2;

yield3;

};

[...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(){

// somecode

}

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。

**3 next方法的参数**

yield句本身没有返回值，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语句的返回值。

function\* f() {

for(vari=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语句，变量reset的值总是undefined。当next方法带一个参数true时，当前的变量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语句的值设为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语句的值设为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语句的返回值，所以第一次使用next方法时，不能带有参数。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。

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。

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。

**4 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() {

yield1;

yield2;

yield3;

yield4;

yield5;

return6;

}

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语句的值。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，实现斐波那契数列的例子。

function\* fibonacci() {

let[prev, curr] = [0, 1];

for (;;) {

[prev, curr] = [curr, prev + curr];

yieldcurr;

}

}

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) {

letpropKeys = 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() {

letpropKeys = 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

return3

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

**5 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 {

thrownew Error('a');

thrownew 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方法被捕获以后，会附带执行下一条yield语句。也就是说，会附带执行一次next方法。

var gen = function\* gen(){

try {

yieldconsole.log('a');

} catch(e) {

//...

}

yieldconsole.log('b');

yieldconsole.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。

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(){

yieldconsole.log('hello');

yieldconsole.log('world');

}

var g = gen();

g.next();

try {

thrownew 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语句，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，就不得不为每个函数内部写一个错误处理语句，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();

yieldy;

}

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() {

yield1;

console.log('throwing an exception');

thrownew Error('generator broke!');

yield2;

yield3;

}

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。

**6 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() {

yield1;

yield2;

yield3;

}

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() {

yield1;

yield2;

yield3;

}

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，那么return方法会推迟到finally代码块执行完再执行。

function\* numbers () {

yield1;

try {

yield2;

yield3;

} finally{

yield4;

yield5;

}

yield6;

}

var g = numbers()

g.next() // { done: false, value: 1 }

g.next() // { done: false, value: 2 }

g.return(7) // { done: false, value: 4 }

g.next() // { done: false, value: 5 }

g.next() // { done: true, value: 7 }

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。

7 yield\*语句

如果在Generater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\*语句，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()) {

yieldv;

}

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';

yieldinner();

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命令后面跟的是一个遍历器对象，需要在yield命令后面加上星号，表明它返回的是一个遍历器对象。这被称为yield\*语句。

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) {

yieldvalue;

}

for(var value of iter2) {

yieldvalue;

}

}

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，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语句返回整个字符串，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() {

yield2;

yield3;

return"foo";

}

function \*bar() {

yield1;

var v =yield \*foo();

console.log( "v: " + v );

yield4;

}

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) {

letresult = 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(leti=0; i < tree.length; i++) {

yield\* iterTree(tree[i]);

}

} else {

yieldtree;

}

}

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);

yieldt.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 newTree(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，表示这个属性是一个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.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bind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语句），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。协程与其不同，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ECMAScript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ECMAScript6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语句交换控制权。

**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语句里面，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语句下面，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()

上面代码表示，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，并且异步加载数据。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() {

varresult = yield request("http://some.url");

varresp = 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语句构成的表达式，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语句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。

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 (vari=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 (vari=0; i< jobs.length; i++){

varjob = 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) {

letkeys = Object.keys(obj);

for (leti=0; i < keys.length; i++) {

letkey = 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){

varnextIndex = 0;

while(nextIndex< array.length){

yieldarray[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。