迭代器与生成器

生成器

ES6新增结构，拥有在一个函数快内暂停和回复代码执行的能力

生成器基础

函数名前加*表示是一个生成器，能定义函数的地方就能定义生成器。箭头函数不能用来定义生成器函数。

```
// 生成器函数声明
function* generatorFn() {}
// 生成器函数表达式
let generatorFn = function* () {}
// 作为对象字面量方法的生成器函数
let foo = {
 * generatorFn() {}
}
// 作为类实例方法的生成器函数
class Foo {
 * generatorFn() {}
}
// 作为类静态方法的生成器函数
class Bar {
 static * generatorFn() {}
}
```

标识生成器函数的星号不受两侧空格的影响：

```
// 等价的生成器函数：
function* generatorFnA() {}
function *generatorFnB() {}
function * generatorFnC() {}
// 等价的生成器方法：
class Foo {
 *generatorFnD() {}
 * generatorFnE() {}
}
```

- 调用生成器函数会产生一个生成器对象(与迭代器工厂函数类似)。
- 生成器对象一开始处于暂停执行suspended状态
- 生成器对象也实现了Iterator接口，所以有next()方法，调用这个方法会让生成器开始或者恢复执行

```
function* generatorFn() {}
const g = generatorFn();
```

```
console.log(g); // generatorFn {<suspended>}
console.log(g.next); // f next() { [native code] }
```

- next()返回值类似迭代器，done属性，value属性
- 函数体为空的生成器中间不会停留，调用一次就done：true
- value属性是生成器函数的返回值，默认undefined，可以通过生成器函数返回值指定

```
function* generatorFn() {
 return 'foo';
}
let generatorObject = generatorFn();
console.log(generatorObject); // generatorFn {<suspended>}
console.log(generatorObject.next()); // { done: true, value: 'foo' }
```

- 生成器函数只会初次调用next()方法后开始执行
- 生成器对象也实现了Iterable接口

```
function* generatorFn() {}
console.log(generatorFn);
// f* generatorFn() {}
console.log(generatorFn()[Symbol.iterator]);
// f [Symbol.iterator]() {native code}
console.log(generatorFn());
// generatorFn {<suspended>}
console.log(generatorFn()[Symbol.iterator]());
// generatorFn {<suspended>}
const g = generatorFn();
console.log(g === g[Symbol.iterator]());
// true
```

总而言之，生成器除了声明时候不大一样，别的都跟迭代器基本一样

通过yield中断执行

yield可以让生成器停止和开始执行
生成器函数遇到yield就停止，需要生成器对象调用next()恢复执行

```
function* generatorFn() {
 yield;
}
let generatorObject = generatorFn();
console.log(generatorObject.next()); // { done: false, value: undefined }
console.log(generatorObject.next()); // { done: true, value: undefined }
```

yield 关键字有点像函数的中间返回语句，它生成的值会出现在 next()方法返回的对象里

```
function* generatorFn() {
 yield 'foo';
```

```
 yield 'bar';
 return 'baz';
}
let generatorObject = generatorFn();
console.log(generatorObject.next()); // { done: false, value: 'foo' }
console.log(generatorObject.next()); // { done: false, value: 'bar' }
console.log(generatorObject.next()); // { done: true, value: 'baz' }
```

一个生成器对象调用next()不会影响其他生成器
yield关键字只能在生成器函数内部使用，且必须直接位于生成器函数定义中，否则抛出错误

```
// 有效
function* validGeneratorFn() {
 yield;
}
// 无效
function* invalidGeneratorFnA() {
 function a() {
 yield;
 }
}
// 无效
function* invalidGeneratorFnB() {
 const b = () => {
 yield;
 }
}
// 无效
function* invalidGeneratorFnC() {
 (() => {
 yield;
 })();
}
```

生成器用法：

- 生成器对象作为可迭代对象

  ```
  function* generatorFn() {
   yield 1;
   yield 2;
   yield 3;
  }
  for (const x of generatorFn()) {
   console.log(x);
  }
  // 1
  // 2
  // 3
  ```

  需要一个自定义迭代对象时，使用生成器会特别有用

  ```
  function* nTimes(n) {
   while(n--) {
  ```

```
 yield;
 }
}
for (let _ of nTimes(3)) {
 console.log('foo');
}
// foo
// foo
// foo
```

- 使用yield实现输入和输出
  yield 关键字还可以作为函数的中间参数使用

```
function* generatorFn(initial) {
 console.log(initial);
 console.log(yield);
 console.log(yield);
}
let generatorObject = generatorFn('foo');
// 第一次next参数不会传入，第一次next只是单纯的启动
generatorObject.next('bar'); // foo
generatorObject.next('baz'); // baz
generatorObject.next('qux'); // qux
```

输入输出

```
function* generatorFn() {
 return yield 'foo';
}
let generatorObject = generatorFn();
console.log(generatorObject.next()); // { done: false, value: 'foo' }
console.log(generatorObject.next('bar')); // { done: true, value: 'bar' }
```

yield 关键字并非只能使用一次。比如，以下代码就定义了一个无穷计数生成器函数

```
function* generatorFn() {
 for (let i = 0;;++i) {
 yield i;
 }
}
let generatorObject = generatorFn();
console.log(generatorObject.next().value); // 0
console.log(generatorObject.next().value); // 1
console.log(generatorObject.next().value); // 2
console.log(generatorObject.next().value); // 3
console.log(generatorObject.next().value); // 4
console.log(generatorObject.next().value); // 5
...
```

迭代生成索引

```
function* nTimes(n) {
 let i = 0;
 while(n--) {
 yield i++;
 }
}
for (let x of nTimes(3)) {
 console.log(x);
}
// 0
// 1
// 2
```

范围索引，填充数组

```
function* range(start, end) {
 while(end > start) {
 yield start++;
 }
}
for (const x of range(4, 7)) {
 console.log(x);
}
// 4
// 5
// 6
function* zeroes(n) {
 while(n--) {
 yield 0;
 }
}
console.log(Array.from(zeroes(8))); // [0, 0, 0, 0, 0, 0, 0, 0]
```

- 产生可迭代对象
可以使用星号增强 yield 的行为，让它能够迭代一个可迭代对象，从而一次产出一个值

```
// 等价的 generatorFn：
// function* generatorFn() {
// for (const x of [1, 2, 3]) {
// yield x;
// }
// }
function* generatorFn() {
 yield* [1, 2, 3];
}
let generatorObject = generatorFn();
for (const x of generatorFn()) {
 console.log(x);
}
// 1
// 2
// 3
```

yield 星号两侧的空格不影响其行为
yield* 在done：true，普通迭代器value是undefined
生成器函数迭代器value是生成器函数返回值

```
function* innerGeneratorFn() {
 yield 'foo';
 return 'bar';
}
function* outerGeneratorFn(genObj) {
 console.log('iter value:', yield* innerGeneratorFn());
}
for (const x of outerGeneratorFn()) {
 console.log('value:', x);
}
// value: foo
// iter value: bar
```

- 使用yield实现递归
  yield 最有用的地方是实现递归

```
function* nTimes(n) {
 if (n > 0) {
 yield* nTimes(n - 1);
 yield n - 1;
 }
}
for (const x of nTimes(3)) {
 console.log(x);
}
// 0
// 1
// 2
```

遍历图！！！

```
class Node {
 constructor(id) {
 this.id = id;
 this.neighbors = new Set();
 }
 connect(node) {
 if (node !== this) {
 this.neighbors.add(node);
 node.neighbors.add(this);
 }
 }
}
class RandomGraph {
 constructor(size) {
 this.nodes = new Set();
 // 创建节点
 for (let i = 0; i < size; ++i) {
 this.nodes.add(new Node(i));
 }
 // 随机连接节点
```

```
 const threshold = 1 / size;
 for (const x of this.nodes) {
 for (const y of this.nodes) {
 if (Math.random() < threshold) {
 x.connect(y);
 }
 }
 }
 }
 // 这个方法仅用于调试
 print() {
 for (const node of this.nodes) {
 const ids = [...node.neighbors]
 .map((n) => n.id)
 .join(',');
 console.log(`${node.id}: ${ids}`);
 }
 }
}
const g = new RandomGraph(6);
g.print();
// 示例输出：
// 0: 2,3,5
// 1: 2,3,4,5
// 2: 1,3
// 3: 0,1,2,4
// 4: 2,3
// 5: 0,4


class Node {
 constructor(id) {
 ...
 }
 connect(node) {
 ...
 }
}
class RandomGraph {
 constructor(size) {
 ...
 }
 print() {
 ...
 }
 isConnected() {
 const visitedNodes = new Set();
 function* traverse(nodes) {
 for (const node of nodes) {
 if (!visitedNodes.has(node)) {
 yield node;
 yield* traverse(node.neighbors);
 }
 }
 }
 // 取得集合中的第一个节点
 const firstNode = this.nodes[Symbol.iterator]().next().value;
 // 使用递归生成器迭代每个节点
 for (const node of traverse([firstNode])) {
 visitedNodes.add(node);
 }
```

```
 return visitedNodes.size === this.nodes.size;
 }
}
```

生成器作为默认迭代器

生成器对象实现了Iterable接口，生成器函数和默认迭代器被调用之后都生成迭代器，所以生成器适合做默认迭代器

```
class Foo {
 constructor() {
 this.values = [1, 2, 3];
 }
 * [Symbol.iterator]() {
 yield* this.values;
 }
}
const f = new Foo();
for (const x of f) {
 console.log(x);
}
// 1
// 2
// 3
```

提前终止生成器

与迭代器类似，一个实现Iterator接口对象一定有next(),return(),生成器对象除了这俩方法还有第三个方法，throw()
return()和throw()都可以强制让生成器进入关闭状态
return(),value就是终止迭代器的值

return

生成器对象都有return(),进入关闭状态后就无法恢复了

```
function* generatorFn() {
 for (const x of [1, 2, 3]) {
 yield x;
 }
}
const g = generatorFn();
console.log(g.next()); // { done: false, value: 1 }
console.log(g.return(4)); // { done: true, value: 4 }
console.log(g.next()); // { done: true, value: undefined }
console.log(g.next()); // { done: true, value: undefined }
console.log(g.next()); // { done: true, value: undefined }
```

for-of 循环等内置语言结构会忽略状态为 done: true 的 IteratorObject 内部返回的值

throw

会在暂停的时候将一个提供的错误注入到生成器对象中。如果错误未被处理，生成器就会关闭：

```
function* generatorFn() {
 for (const x of [1, 2, 3]) {
 yield x;
 }
}
const g = generatorFn();
console.log(g); // generatorFn {<suspended>}
try {
 g.throw('foo');
} catch (e) {
 console.log(e); // foo
}
console.log(g); // generatorFn {<closed>}
```

但是如果处理了即有catch，则生成器不会关闭

```
function* generatorFn() {
 for (const x of [1, 2, 3]) {
 try {
 yield x;
 } catch(e) {}
 }
}
const g = generatorFn();
console.log(g.next()); // { done: false, value: 1}
g.throw('foo');
console.log(g.next()); // { done: false, value: 3}
```

小结

ES6正式支持迭代模式，并引入两个新的语言特性：迭代器，生成器

迭代器是一个可以由任意对象实现的接口，支持连续获取对象产出的每一个值。任何实现 Iterable接口的对象都有一个 Symbol.iterator 属性，这个属性引用默认迭代器。默认迭代器就像一个迭代器工厂，也就是一个函数，调用之后会产生一个实现 Iterator 接口的对象

迭代器必须通过连续调用 next()方法才能连续取得值，这个方法返回一个 IteratorObject。这个对象包含一个 done 属性和一个 value 属性。前者是一个布尔值，表示是否还有更多值可以访问；后者包含迭代器返回的当前值。这个接口可以通过手动反复调用 next()方法来消费，也可以通过原生消费者，比如 for-of 循环来自动消费

生成器是一种特殊的函数，调用之后会返回一个生成器对象。生成器对象实现了 Iterable 接口，因此可用在任何消费可迭代对象的地方。生成器的独特之处在于支持 yield 关键字，这个关键字能够暂停执行生成器函数。使用 yield 关键字还可以通过 next()方法接收输入和产生输出。在加上星号之后，yield 关键字可以将跟在它后面的可迭代对象序列化为一连串值