0：语法上，generator函数是一个状态机，封装了Diogenes内部状态

1：执行generator 函数会返回一个遍历器对象，可以依次遍历generator函数的每一个状态

3：形式上，generator函数是一个普通函数，但是有两个特征，

特征一：function 关键字 与函数名 之间有一个 星号* ，

特征二：函数体内部使用yield表达式,定义不同的内部状态（yield 产出的意思）

4：

```
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。

5：yieid表达式

1:与 return函数的区别

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，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）。

2: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

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

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

6: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。而**执行 Generator 函数会返回一个遍历器对象，所以正好用Generator 函数去接收一个（任意一个对象的`Symbol.iterator`方法）；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 myIteratble = {};
myIteratble[Symbol.iterator] = function *() {
  yield 1;
  yield 2;
  yield 3;
  return 4;
};
console.log([...myIteratble]); //(3) [1, 2, 3]
```

```
function* gen(){
  // some code
}

var g = gen();

g[Symbol.iterator]() === g
// true
```

7：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 }
```

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。

7： for...of循环

可以自动遍历generator函数运行时生成的iterator对象，不需要再调用next 方法

8：Generator 的throw方法

```
var g = function*() {
  try{
    yield 1;
  }catch(e){
    console.log('内部捕获',e);
  }
};
var i = g();
console.log(i.next());
console.log(i.next("a"));
console.log(i.next("b"));
try{
  i.throw("b");
  i.throw('a');

}catch(e){
  console.log("外部捕获",e)
}
```

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。

9：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。

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。