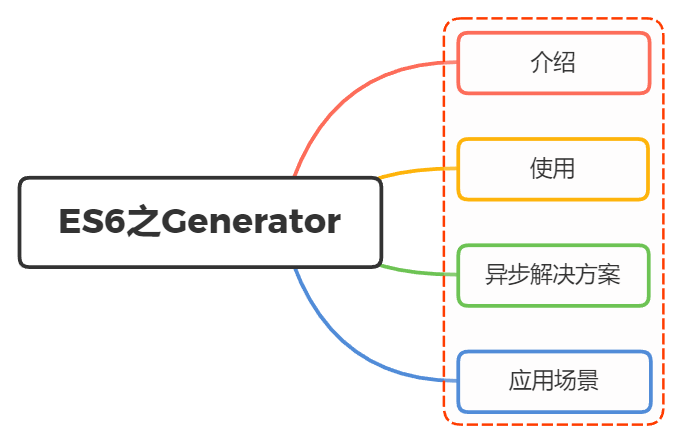
# 面试官：你是怎么理解ES6中 Generator的？使用场景？



## 一、介绍

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

回顾下上文提到的解决异步的手段：

* 回调函数
* promise

那么，上文我们提到promsie已经是一种比较流行的解决异步方案，那么为什么还出现Generator？甚至async/await呢？

该问题我们留在后面再进行分析，下面先认识下Generator

### Generator函数

执行 Generator 函数会返回一个遍历器对象，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：

* 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

function\* helloWorldGenerator() {  
 yield 'hello';  
 yield 'world';  
 return 'ending';  
}

## 二、使用

Generator 函数会返回一个遍历器对象，即具有Symbol.iterator属性，并且返回给自己

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

通过yield关键字可以暂停generator函数返回的遍历器对象的状态

function\* helloWorldGenerator() {  
 yield 'hello';  
 yield 'world';  
 return 'ending';  
}  
var hw = helloWorldGenerator();

上述存在三个状态：hello、world、return

通过next方法才会遍历到下一个内部状态，其运行逻辑如下：

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

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

done用来判断是否存在下个状态，value对应状态值

yield表达式本身没有返回值，或者说总是返回undefined

通过调用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\* 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 }

正因为Generator函数返回Iterator对象，因此我们还可以通过for...of进行遍历

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

原生对象没有遍历接口，通过Generator函数为它加上这个接口，就能使用for...of进行遍历了

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

## 三、异步解决方案

回顾之前展开异步解决的方案：

* 回调函数
* Promise 对象
* generator 函数
* async/await

这里通过文件读取案例，将几种解决异步的方案进行一个比较：

### 回调函数

所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，再调用这个函数

fs.readFile('/etc/fstab', function (err, data) {  
 if (err) throw err;  
 console.log(data);  
 fs.readFile('/etc/shells', function (err, data) {  
 if (err) throw err;  
 console.log(data);  
 });  
});

readFile函数的第三个参数，就是回调函数，等到操作系统返回了/etc/passwd这个文件以后，回调函数才会执行

### Promise

Promise就是为了解决回调地狱而产生的，将回调函数的嵌套，改成链式调用

const fs = require('fs');  
  
const readFile = function (fileName) {  
 return new Promise(function (resolve, reject) {  
 fs.readFile(fileName, function(error, data) {  
 if (error) return reject(error);  
 resolve(data);  
 });  
 });  
};  
  
  
readFile('/etc/fstab').then(data =>{  
 console.log(data)  
 return readFile('/etc/shells')  
}).then(data => {  
 console.log(data)  
})

这种链式操作形式，使异步任务的两段执行更清楚了，但是也存在了很明显的问题，代码变得冗杂了，语义化并不强

### generator

yield表达式可以暂停函数执行，next方法用于恢复函数执行，这使得Generator函数非常适合将异步任务同步化

const gen = function\* () {  
 const f1 = yield readFile('/etc/fstab');  
 const f2 = yield readFile('/etc/shells');  
 console.log(f1.toString());  
 console.log(f2.toString());  
};

### async/await

将上面Generator函数改成async/await形式，更为简洁，语义化更强了

const asyncReadFile = async function () {  
 const f1 = await readFile('/etc/fstab');  
 const f2 = await readFile('/etc/shells');  
 console.log(f1.toString());  
 console.log(f2.toString());  
};

### 区别：

通过上述代码进行分析，将promise、Generator、async/await进行比较：

* promise和async/await是专门用于处理异步操作的
* Generator并不是为异步而设计出来的，它还有其他功能（对象迭代、控制输出、部署Interator接口...）
* promise编写代码相比Generator、async更为复杂化，且可读性也稍差
* Generator、async需要与promise对象搭配处理异步情况
* async实质是Generator的语法糖，相当于会自动执行Generator函数
* async使用上更为简洁，将异步代码以同步的形式进行编写，是处理异步编程的最终方案

## 四、使用场景

Generator是异步解决的一种方案，最大特点则是将异步操作同步化表达出来

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

包括redux-saga中间件也充分利用了Generator特性

import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'  
import Api from '...'  
  
function\* fetchUser(action) {  
 try {  
 const user = yield call(Api.fetchUser, action.payload.userId);  
 yield put({type: "USER\_FETCH\_SUCCEEDED", user: user});  
 } catch (e) {  
 yield put({type: "USER\_FETCH\_FAILED", message: e.message});  
 }  
}  
  
function\* mySaga() {  
 yield takeEvery("USER\_FETCH\_REQUESTED", fetchUser);  
}  
  
function\* mySaga() {  
 yield takeLatest("USER\_FETCH\_REQUESTED", fetchUser);  
}  
  
export default mySaga;

还能利用Generator函数，在对象上实现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

## 参考文献

* https://es6.ruanyifeng.com/#docs/generator-async