

ES6 PPT1606281: 生成器 Generator 函数

Thursday, November 5, 2015 10:15 AM

生成器 Generator 函数：

1. 生成器函数是一个状态机，封装了多个可遍历产出(Yield)的内部状态，生成器函数执行后返回一个可遍历(Iterable)的遍历器(Iterator)对象
2. 生成器函数基本语法：`function* () { yield myValue; return; }`

```
function* myIteratorGenerator(text) {  
  var count = 1;  
  yield `${count} 来自${text}`;  
  count++;  
  yield `${count} 来自${text}`;  
  count++;  
  return `${count} 结束${text}`;  
  yield `${count} 我不会被遍历到`;  
}  
  
var myIterator = myIteratorGenerator('Iterator'); // 调用生成器函数生成一个遍历器  
console.log(myIterator.next()); // {value: "1来自Iterator", done: false} -> 第一次调用遍历器 next 方法，生成器函数自始执行至首个  
yield 语句产出一值为止  
console.log(myIterator.next()); // {value: "2来自Iterator", done: false} -> 第二次调用遍历器 next 方法，生成器函数自上个 yield 后执  
行至下个 yield 为止  
console.log(myIterator.next()); // {value: "3结束Iterator", done: true} -> 第三次调用遍历器 next 方法，生成器函数自上个 yield 后执  
行至 return 结束遍历  
console.log([...myIteratorGenerator('Iterable')]); // ["1来自Iterable", "2来自Iterable"]  
var myGenerators = {  
  generator1: function* () {  
    yield '1-1';  
    yield '1-2';  
  },  
  * generator2() {  
    yield '2-1';  
    yield '2-2';  
  }  
}  
console.log([...myGenerators.generator1()]); // ["1-1", "1-2"]  
for(let i of myGenerators.generator2()) console.log(i); // "2-1" // "2-2"  
  
3. 给遍历器的 next 方法传参数，该参数将被当作生成器函数执行时上一个 yield 语句的返回值  
function* dataConsumer() {  
  console.log('Started');  
  console.log(`1. ${yield}`);  
  console.log(`2. ${yield}`);  
  return 'result';  
}  
  
let myGenObj2 = dataConsumer();  
myGenObj2.next(); // "Started"  
myGenObj2.next('a') // "1. a"  
myGenObj2.next('b') // "2. b"  
//  
var myGenObj3 = (function* f() {  
  for(var i=0; true; i++) {  
    var reset = yield i;  
    if(reset) { i = -1; }  
  }  
})();  
console.log(myGenObj3.next()); // { value: 0, done: false }  
console.log(myGenObj3.next()); // { value: 1, done: false }  
console.log(myGenObj3.next(true)); // { value: 0, done: false }
```

4. 生成器函数内 throw 抛错与其返回遍历器的 throw 方法

- a. 生成器函数内 throw 抛错，遍历就会终止

```
function *myGtrThrow() {  
  var x = yield 111;  
  var y = x.toUpperCase();  
  yield y;  
}  
  
var myItr = myGtrThrow();  
console.log(myItr.next()); // {value: 111, done: false}  
try {
```

```

    console.log(myItr.next(222));
  } catch (err) {
    console.log('错误: ' + err); // 错误: TypeError: x.toUpperCase is not a function
    console.log(myItr.next(333)); // {done: true} -> 一旦抛错, 遍历就结束
  }
}

```

- b. 生成器函数内部会先尝试捕捉生成遍历器的 throw 方法, 如内部无捕捉程序再尝试生成器函数外部捕捉

```

var myGtr = function* () {
  try {
    yield 123;
    yield 456;
  } catch (e) {
    console.log('内部捕获', e); // "内部捕获 a" -> 第2行输出
  }
};
var myItrThrow = myGtr();
console.log(myItrThrow.next()); // {value: 123, done: false} -> 第1行输出
try {
  myItrThrow.throw('a');
  myItrThrow.throw('b');
} catch (e) {
  console.log('外部捕获', e); // "外部捕获 b" -> 第3行输出
}
console.log(myItrThrow.next()); // {done: true} -> 第4行输出, 一旦抛错, 遍历就结束

```

5. 生成器函数返回遍历器的 return 方法

- a. 返回传入 return 方法的参数并结束遍历

```

function* myGtr2() {
  yield 1;
  yield 2;
  yield 3;
}
var myItrReturn = myGtr2();
console.log(myItrReturn.next()); // { value: 1, done: false }
console.log(myItrReturn.return('foo')); // { value: "foo", done: true }
console.log(myItrReturn.next()); // { value: undefined, done: true }

```

- b. 如果生成器函数内有 try...finally 代码块, return 方法会推迟到 finally 代码块执行完再执行

```

function* myGtrTryFinally() {
  yield 1;
  try {
    yield 2;
    yield 3;
  } finally {
    yield 4;
    yield 5;
  }
  yield 6;
}
var myItrReturn2 = myGtrTryFinally();
console.log(myItrReturn2.next()); // {value: 1, done: false}
console.log(myItrReturn2.next()); // {value: 2, done: false}
console.log(myItrReturn2.return(7)); // {value: 4, done: false}
console.log(myItrReturn2.next()); // {value: 5, done: false}
console.log(myItrReturn2.next()); // {value: 7, done: true}

```

6. 使用生成器函数给对象的 Symbol.iterator 方法赋值, 使该对象成为可遍历对象:

```

var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  return 3;
};
console.log([...myIterable]); // [1, 2]
for(var i of myIterable) console.log(i); // 1 // 2

```

7. 使用 yield* myIterable 进行批量产出:

- a. 一般用法:

```

function* myGtrA() {
  yield 'A-1';
  yield 'A-2';
}
function* myGtrB() {
  yield 'B-1';
  yield* myGtrA();
}

```

```

    yield 'B-2';
    yield* ['B-3-1', 'B-3-2'];
    yield 'B-4';
}
var myItrStar = myGtrB();
console.log(...myItrStar); // ["B-1", "A-1", "A-2", "B-2", "B-3-1", "B-3-2", "B-4"]

```

- b. 使用 return 语句向代理它的生成器函数返回数据，yield * 返回值即为 return 后面跟的值：

```

function *myGtrC() {
    yield 2;
    return "返回值C";
}
function *myGtrD() {
    yield 1;
    var v = yield *myGtrC();
    console.log("v: " + v);
    yield 3;
}
var myItrStarReturn = myGtrD();
console.log(myItrStarReturn.next()); // {value: 1, done: false}
console.log(myItrStarReturn.next()); // {value: 2, done: false}
console.log(myItrStarReturn.next()); // "v: 返回值C" // {value: 3, done: false}
console.log(myItrStarReturn.next()); // {done: true}

```

8. 生成器函数下的 this 指向：

- a. 生成器函数返回的遍历器是该生成器函数的实例，但 this 不指向该遍历器，且不能对其使用 new 关键字：

```

function* myGtrD() {
    this.val = 123;
}
myGtrD.prototype.hello = function () {
    return '哈喽';
};
let myItrD = myGtrD();
console.log(myItrD instanceof myGtrD); // true
console.log(myItrD.hello()); // "哈喽"
console.log(myItrD.val); // undefined
new myGtrD(); // Uncaught TypeError: myGtrD is not a constructor -> 报错，不能对生成器函数使用 new 语句

```

- b. 使用 myGenerator.call(myGenerator.prototype) 变通解决生成器函数内 this 指向问题：

```

function* myGtrThisOK() {
    this.a = 1;
    yield this.b = 2;
    yield this.c = 3;
}
var myItrThisOK = myGtrThisOK.call(myGtrThisOK.prototype);
console.log(myItrThisOK.next()); // Object {value: 2, done: false}
console.log(myItrThisOK.next()); // Object {value: 3, done: false}
console.log(myItrThisOK.next()); // Object {done: true}
console.log(myItrThisOK.a); // 1
console.log(myItrThisOK.b); // 2
console.log(myItrThisOK.c); // 3

```

9. 应用：

- a. 暂缓执行函数

```

function* myYieldFreeGenerator() {
    console.log('执行了！')
}
var myEmptyIterator = myYieldFreeGenerator();
setTimeout(function () { myEmptyIterator.next(); }, 2000); // 于2秒后打印出“执行了”，没有 yield 语句产生的生成器函数即一个简单的暂缓执行函数

```

- b. 可遍历属性对象

```

class Cat {
    constructor(name, age, gender = 'unknown') {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this[Symbol.iterator] = function* () {
            for (var key in this) yield `${key}: ${this[key]}`;
        }
    }
}
console.log(...new Cat('Tom', 3)); // {name: "Tom", age: 3, gender: "unknown"}

```

- c. 快速实现切换状态机

```

var stateGenerator = function* () {
  while (true) {
    yield true;
    yield false;
  }
};
var myState = stateGenerator();
console.log(myState.next().value); // true
console.log(myState.next().value); // false
console.log(myState.next().value); // true
console.log(myState.next().value); // false
console.log(myState.next().value); // true

```

d. 异步操作的同步化表达

// 以下依次发起 AJAX 请求，该示例可在 BABEL 在线编辑器下 (<https://babeljs.io/repl/>) 运行

```

function myAjaxRequest(url) {
  $.get(url).done(function(response) {
    myItr.next(response);
  }).fail(function(err) {
    myItr.throw('加载' + url + '失败');
  });
}

function* myGen() {
  var data = null;
  try {
    data = yield myAjaxRequest('https://babeljs.io/favicon-16x16.png');
    console.log('favicon-16x16.png 数据已 get 载入:' + data.length + '字节'); // "favicon-16x16.png 数据已 get 载入: 479字
    data = yield myAjaxRequest('https://babeljs.io/favicon-160x160.png');
    console.log('favicon-160x160.png 数据已 get 载入:' + data.length + '字节'); // "favicon-160x160.png 数据已 get 载入: 5912
    data = yield myAjaxRequest('http://不存在的地址/'); // "错误: 加载 http://不存在的地址/ 失败" -> 第三行输出
    console.log('不存在的地址数据已载入:' + data.length + '字节'); // 此行不会被输出
    data = yield myAjaxRequest('https://babeljs.io/favicon-192x192.png'); // 此行不会被输出
    console.log('favicon-192x192.png 数据已 get 载入:' + data.length + '字节'); // 此行不会被输出
  } catch(err) {
    console.log('错误:' + err);
  }
}

var myItr = myGen();
myItr.next(); // 开始运行myGen函数内代码

```

e. 斐波那契数列

```

function* fibonacci() {
  let [prev, curr] = [0, 1];
  for (;;) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (let n of fibonacci()) {
  if (n > 100) break;
  console.log(n);
} // 1 // 2 // 3 // 5 // 8 // 13 // 21 // 34 // 55 // 89

```

f. 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' ] ];
const flatTree = [...iterTree(tree)];
console.log(flatTree); // ["a","b","c","d","e"]

```

10. 生成器语法注意事项：

a. 非生成器函数不能使用 yield 语句

```

(function () {

```

```
    yield 1;
  })() // Uncaught SyntaxError: Unexpected number -> 报错, yield 语句不能用于非生成器函数
```

b. 生成器函数内如果无限产出 yield 语句, 对生成的遍历器使用扩展运算符时将产生灾难性后果——死循环

```
function* myGtrEndlessLoop() {
  var count = 0;
  while(true) {
    yield count++;
  }
}
var myEndlessItr = myGtrEndlessLoop();
// console.log([...myEndlessItr]); // 如果取消本行开始的注释, 程序将无线循环下去而造成系统崩溃
```