* [**Posts**](https://viblo.asia/)

**[Questions](https://viblo.asia/questions)**

* [**Discussions**](https://viblo.asia/discussion)



 Sign In/Sign up

[[](https://viblo.asia/s/state-of-chain-2018-zD5dBoqOZjY)](https://viblo.asia/s/state-of-chain-2018-zD5dBoqOZjY" \t "_blank)

+6

[](https://viblo.asia/u/nguyen.thanh.tungb)

[**Nguyen Thanh Tung B**](https://viblo.asia/u/nguyen.thanh.tungb)@nguyen.thanh.tungb

Follow

 517  25  27

Published Aug 28th, 2016 10:40 PM

1.2K

3

2

**Generator trong Javasccript**

[JavaScript](https://viblo.asia/tags/javascript)[generator](https://viblo.asia/tags/generator)

**Mở đầu**

Trong javascript một khi function được thực thi thì nó sẽ được đảm bảo run-to-completion tức là những phần code khác không thể can thiệp, làm gián đoạn quá trình chạy của function đó. Tuy nhiên ES6 đã cho ra mắt 1 loại function mới mà không hành xử theo lẽ thông thường như thế - Generator

Hãy xem xét ví dụ sau đây:

var x = 1;

function foo() {

x++;

bar();

console.log( "x:", x );

}

function bar() {

x++;

}

foo();

Chúng ta có thể thấy function bar được gọi bên trong function foo và giá trị của x sau khi gọi foo() là 3. Trong trường hợp không có lời gọi bar() bên trong foo giá trị của x sẽ là 2. Hãy tưởng tượng bằng 1 cách nào đó mặc dù chúng ta không gọi bar() ở bên trong foo mà kết quả trả về vẫn là 3 ??? Đó chính là lúc mà chúng ta sử dụng generator

Hãy thay đổi đoạn code 1 chút

var x = 1;

function \*foo() {

x++;

yield;

console.log( "x:", x );

}

function bar() {

x++;

}

Bạn có để ý thấy sự khác biệt ?

Chúng ta đã có 1 generator declaration với khai báo function \*foo() - chú ý dấu \* trong phần khai báo.

Và sử dụng generator như sau

var it = foo();

it.next();

x; // 2

bar();

x; // 3

it.next();

Xem xét từng bước một của quá trình trên:

1. Phép toán it = foo() không thực thi \*foo() ngay lập tức mà tạo ra 1 iterator để kiểm soát quá trình thực thi này
2. Dòng it.next() đầu tiên bắt đầu chạy genertor và thực hiện phép toán x++
3. generator dừng lại ở lệnh yield sau khi lời gọi it.next() đầu tiên hoàn thành. Tại thời điểm này generator vẫn còn hoạt động nhưng chuyển sang trạng thái dừng
4. Chúng ta kiểm tra giá trị của x và kết quả trả về là 2
5. Chúng ta gọi function bar() để tăng giá trị của x
6. Chúng ta kiểm tra giá trị của x lần nữa và lần này kết quả là 3
7. Lời gọi it.next() cuối cùng khiến generator hoạt động trở lại, thực hiện lệnh console.login ra giá trị của x

Như vậy chúng ta có thể gọi bar bên ngoài foo và kết quả cuối cùng của x vẫn là 3. Chúng ta có thể coi điều này như là 1 sự phá vỡ rule run-to-completion của function

**Input và output**

generator là 1 function đặc biệt nhưng nó vẫn là 1 function nên nó vẫn có thể có tham số truyền vào - input và giá trị trả về - output như 1 function thông thường.

function \*foo(x,y) {

return x \* y;

}

var it = foo( 6, 7 );

var res = it.next();

res.value;

Chúng ta vẫn có thể truyền giá trị đầu vào 6 và 7 cho x và y Cách gọi function cũng khá quen thuộc - foo( 6, 7 ) nhưng cách mà chúng ta nhận về kết quả có đôi chút khác biệt.

Đầu tiên foo( 6, 7 ) không thực thi function từ đầu đến cuối mà như đã đề cập lúc trước, nó tạo ra 1 iterator. Khi it.next(); được gọi, nó thực thi các lệnh trong function cho đến khi gặp yieldhoặc kết thúc function. Giá trị trả về của lênh it.next(); này là 1 object có thuộc tính value và chúng ta sẽ lấy giá trị từ thuộc tính này.

**Iteration Messaging**

Với lệnh yield đứng độc lập, ta có thể coi đó như 1 điểm dừng để thực hiện những xử lí chen vào giữa quá trình thực thi của function (có vẻ giống debugger mà mình vẫn hay dùng) nhưng thực thế là cách sử dụng của nó còn linh hoạt hơn thế.

function \*foo(x) {

var y = x \* (yield);

return y;

}

var it = foo( 6 );

it.next();

var res = it.next( 7 );

res.value; // 42

Ở đây ta có thể sử dụng yield như 1 nơi để truyền tham số vào Chuôĩ câu lệnh trên truyền 6 là giá trị đầu vào cho x và truyền 7 như là gía trị cho yield và gía trị sau khi tính toán là 6\*7=42.

Câu chuyện ở đây là gì ? Câu lệnh next đầu tiên bắt đầu generator và chạy cho đến khi gặp yield. Khi gặp lệnh yield, yield đặt ra câu hỏi giá trị truyền vào ở đây là gì. Câu lệnh next đầu tiên đã hoàn thành xong nhiệm vụ của mình vậy nên trách nhiệm trả lời này sẽ nằm trong câu lệnh next tiếp theo. Khi lệnh next tiếp theo được thực hiện, nó trả lời cho câu hỏi mà yield đặt ra mà thực thi nốt phần còn lại.

**Muliple iterators**

Mỗi khi chúng ta tạo một iterator thì chúng ta cũng đã tạo 1 generator instance. Khi có nhiều instance của cùng 1 generator thì chúng có thể tương tác với nhau.

function \*foo() {

var x = yield 2;

z++;

var y = yield (x \* z);

console.log( x, y, z );

}

var z = 1;

var it1 = foo();

var it2 = foo();

var val1 = it1.next().value; // 2 <-- yield 2

var val2 = it2.next().value; // 2 <-- yield 2

val1 = it1.next( val2 \* 10 ).value; // 40 <-- x:20, z:2

val2 = it2.next( val1 \* 5 ).value; // 600 <-- x:200, z:3

it1.next( val2 / 2 ); // y:300

// 20 300 3

it2.next( val1 / 4 ); // y:10

// 200 10 3

Trong đoạn code trên thì khi khởi tạo it1 và it2 chúng ta có 2 generator instance của generator foo. Chú ý lệnh yield param có tác dụng như lệnh yield không có tham số nhưng có thêm tác dụng gán param vào thuộc tính value của iterator tại thời điểm câu lệnh next thực thi xong. Trong ví dụ trên giá trị của it1 và it2 có thể nhận iterator còn lại làm tham số trong quá trình tính toán của mình. Đối với trường hợp 2 iterator của 2 generator khác nhau tùy theo cách thực hiện các step trong function của các generator mà kết quả trả về rất khác nhau. Hiện tượng race condition xảy ra và nó tương tự như việc xung đột tài nguyên giữa các thread trong các ngôn ngữ hỗ trợ multi-thread vậy.

Ví dụ với 2 generator sau:

var a = 1;

var b = 2;

function \*foo() {

a++;

yield;

b = b \* a;

a = (yield b) + 3;

}

function \*bar() {

b--;

yield;

a = (yield 8) + b;

b = a \* (yield 2);

}

Nếu foo và bar là 2 function thông thường thì kết quả khi thực thiện foo, bar sẽ chỉ có 2 case:

* thực hiện foo trước, bar sau
* thực hiện bar trước, foo sau

Nó chính là các case có thể xảy ra khi xử lí không đồng bộ foo và bar. Nhưng với generator - thứ phá vỡ luật run-to-completion, việc xen kẽ các step của foo và bar là điều có thể. Số lượng cách sắp xếp các step trộn lẫn với nhau cũng như số lượng kết quả trả về là khá nhiều và phức tạp.

**Values**

**Generator Iterator**

General iterator có thể sử dụng khi cần tính toán 1 chuỗi giá trị mà giá trị sau phụ thuộc vào giá trị trước. Mỗi lần gọi next chúng ta lại có thể nhận được 1 gía trị mới. Công việc này cũng có thể thực hiện bằng closure.

var foo = (function(){

var nextVal;

return function(){

if (nextVal === undefined) {

nextVal = 1;

}

else {

nextVal = (3 \* nextVal) + 6;

}

return nextVal;

};

})();

foo(); // 1

foo(); // 9

foo(); // 33

foo(); // 105

Mỗi lần gọi foo chúng ta nhận được 1 giá trị mới của chuỗi số.

Với phong cách của generator chúng ta có thể định nghĩa 1 generator và cho chạy loop qua generator đó

function \*foo() {

var nextVal;

while (true) {

if (nextVal === undefined) {

nextVal = 1;

}

else {

nextVal = (3 \* nextVal) + 6;

}

yield nextVal;

}cj

}

for (var v of foo()) {

console.log( v );

if (v > 500) {

break;

}

}

// 1 9 33 105 321 969

Cách viết này sử dụng vòng lặp while (true) - bình thường sẽ gây infinite loop tuy nhiên với lệnh yield đặt bên trong loop generator sẽ dừng lại tại mỗi lần lặp, lệnh yield cũng giữ lại gía trị của function foo mà không cần đến closure để lưu lại biến trạng thái. Khi thực hiện lệnh for ofchúng ta vẫn có thể ngắt vòng lặp bằng cách check điều kiện và break.

**Iterating Generators Asynchronously**

Với các mẫu xử lí không đồng bộ, generator đem đến 1 cách tiếp cận mới. Dưới đây là 1 mẫu xử lí không đồng bộ kinh điển sử dụng câu lệnh ajax.

function foo(x,y,cb) {

ajax("http://some.url.1/?x=" + x + "&y=" + y, cb);

}

foo( 11, 31, function(err,text) {

if (err) {

console.error( err );

} else {

console.log( text );

}

});

Khi biểu diễn dưới kiểu sử dụng generator

function foo(x,y) {

ajax(

"http://some.url.1/?x=" + x + "&y=" + y,

function(err,data){

if (err) {

it.throw( err );

}

else {

it.next( data );

}

}

);

}

function \*main() {

try {

var text = yield foo( 11, 31 );

console.log( text );

}

catch (err) {

console.error( err );

}

}

var it = main();

it.next();

Đánh giá sơ bộ: viết lại với kiểu dùng generator dài hơn và có vẻ phức tạp hơn.

Vậy kiểu viết trông có vẻ rối rắm này có đem lại sự khác biệt gì ?

Đầu tiên, hãy xem xét đoạn code

var text = yield foo( 11, 31 );

console.log( text );

Câu lệnh foo thực ra là 1 request ajax nên 2 dòng trên trông có vẻ sẽ tương tự như

var data = ajax( "url.1..." );

console.log( text );

Vấn đề ở đây là thông thường 2 câu lệnh này sẽ không trả về giá trị mong muốn, do request ajax là xử lí không đồng bộ, trả về 1 giá trị trong tương lai còn câu lệnh log thì lại trả về giá trị ngay thời điểm hiện tại. Nhưng khi chúng ta sử dụng generator việc xử lí như trong ví dụ là vẫn ok do sự xuất hiện của yield. Khi bắt gặp lệnh yield generator main tạm thời dừng lại và chờ đợi gía trị để gán cho text. Khi request ajax success thì it.next( data ); được thực hiện, generator được resume. \*\* Điều hay nhất ở đây là chúng ta có thể biểu diễn những xử lí không đồng bộ dưới dạng 1 chuỗi xử lí tuần tự như thể tất cả đều là những xử lí đồng bộ \*\*

TABLE OF CONTENTS