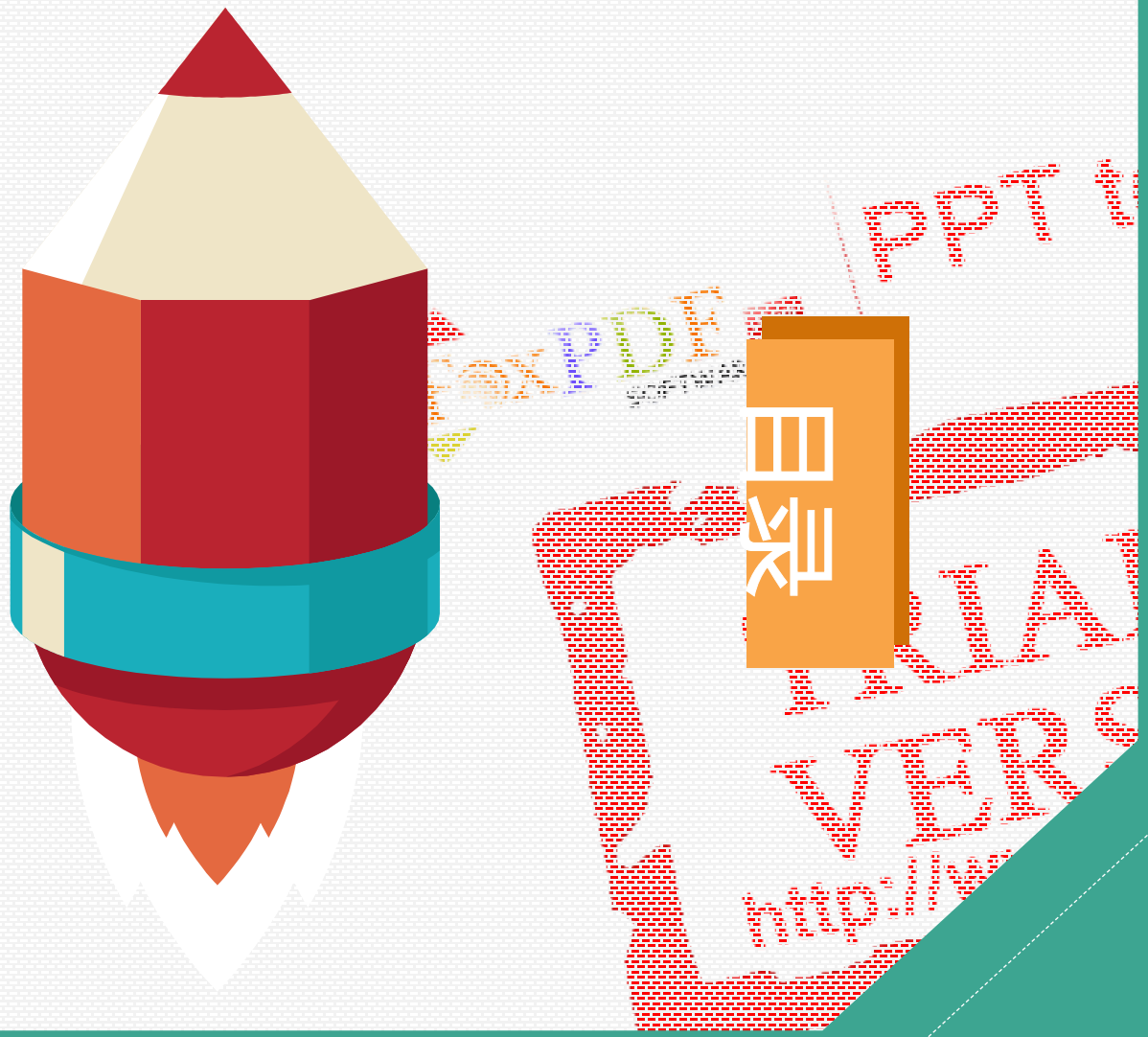
第38课：Generator的语法进阶版和异步

主讲老师：万章

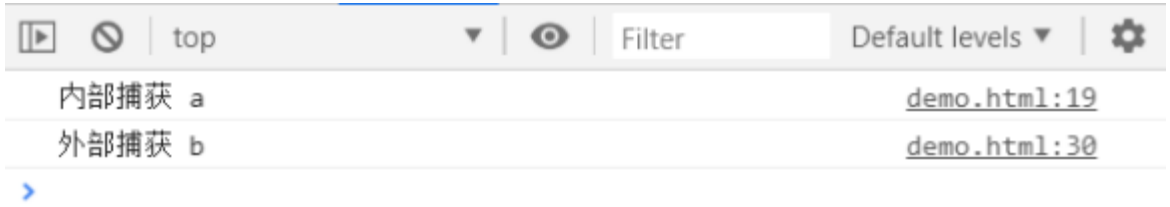


- Generator函数的throw
- Generator函数的return
- yield* 表达式
- Generator函数的其他特性

Generator函数的throw

Generator函数的throw

```
15        var g = function* () {
16            try {
17                yield;
18            } catch (err) {
19                console.log('内部捕获', err);
20            }
21        };
22
23        var i = g();
24        i.next();
25
26        try {
27            i.throw('a');
28            i.throw('b');
29        } catch (err) {
30            console.log('外部捕获', err);
31        }
```



Generator 函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在 Generator 函数体内捕获。

右侧代码中，遍历器对象i连续抛出两个错误：
第一个错误被 Generator 函数体内的catch语句捕获。
第二次抛出错误，由于 Generator 函数内部的catch语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了 Generator 函数体，被函数体外的catch语句捕获

Generator函数的throw

```
var g = function* () {
  while (true) {
    try {
      yield;
    } catch (e) {
      if (e != 'a') throw e;
      console.log('内部捕获', e);
    }
  }
};

var i = g();
i.next();

try {
  throw new Error('a');
  throw new Error('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 [Error: a]
```

这个抛出错误不是迭代器对象抛出的，就只是一个普通的错误抛出，所以错误信息直接就抛到下面的catch里面了，不会调用迭代器里面的catch

```
var g = function* () {
  while (true) {
    yield;
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 a
```

如果 Generator 函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。

Generator函数的throw启动方式

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();
g.throw();
// hello
// Uncaught undefined
```

如果 Generator 函数内部和外部，都没有部署try...catch代码块，那么程序将报错，直接中断执行。

```
function* gen() {
  try {
    yield 1;
  } catch (e) {
    console.log('内部捕获');
  }
}

var g = gen();
g.throw(1);
// Uncaught 1
```

throw方法抛出的错误要被内部捕获，前提是必须至少执行过一次next方法。

这种行为其实很好理解，因为第一次执行next方法，等同于启动执行 Generator 函数的内部代码，否则 Generator 函数还没有开始执行，这时throw方法抛错只可能抛出在函数外部

Generator函数的throw启动方式

```
var gen = function* gen(){
  try {
    yield console.log('a');
  } catch (e) {
    // ...
  }
  yield console.log('b');
  yield console.log('c');
}

var g = gen();
g.next() // a
g.throw() // b
g.next() // c
```

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();

try {
  throw new Error();
} catch (e) {
  g.next();
}
// hello
// world
```

throw方法被捕获以后，会附带执行**下一条yield表达式**。也就是说，会附带执行一次next方法。

只要 Generator 函数内部部署了try...catch代码块，那么遍历器的throw方法抛出的错误，不影响下一

throw命令与g.throw方法是无关的，两者互不影响。

Generator函数的throw启动方式

```
function* foo() {
  var x = yield 3;
  var y = x.toUpperCase();
  yield y;
}

var it = foo();

it.next(); // { value:3, done:false }

try {
  it.next(42);
} catch (err) {
  console.log(err);
}
```

```
TypeError: x.toUpperCase is not a function                    VM517:14
    at foo (<anonymous>:3:13)
    at foo.next (<anonymous>)
    at <anonymous>:12:6
```

Generator 函数体外抛出的错误，可以在函数体内捕获；反过来，Generator 函数体内抛出的错误，也可以被函数体外的catch捕获。

Generator函数的throw启动方式

```
function* g() {
  yield 1;
  console.log('throwing an exception');
  throw new Error('generator broke!');
  yield 2;
  yield 3;
}
```

```
function log(generator) {
  var v;
  console.log('starting generator');
  try {
    v = generator.next();
    console.log('第一次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第二次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第三次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  console.log('caller done');
}

log(g());
```

```
// starting generator
// 第一次运行next方法 { value: 1, done: false }
// throwing an exception
// 捕捉错误 { value: 1, done: false }
// 第三次运行next方法 { value: undefined, done: true }
// caller done
```

一旦 Generator 执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。如果此后还调用next方法，将返回一个value属性等于undefined、done属性等于true的对象，即 JavaScript 引擎认为这个 Generator 已经运行结束了。

Generator函数的return

Generator函数的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。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()        // { value: 1, done: false }
g.return('foo') // { value: "foo", done: true }
g.next()        // { value: undefined, done: true }
```

上面代码中，遍历器对象g调用return方法后，返回值的value属性就是return方法的参数foo。并且，Generator 函数的遍历就终止了，返回值的done属性为true，以后再调用next方法，done属性总是返回true。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()        // { value: 1, done: false }
g.return() // { value: undefined, done: true }
```

如果return方法调用时，不提供参数，则返回值的value属性为undefined。

Generator函数的return

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。

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

Generator函数的return/throw/next辨析

next()、throw()、return()这三个方法本质上是同一件事，可以放在一起理解。它们的作用都是让 Generator 函数恢复执行，并且使用不同的语句替换yield表达式。

```
const g = function* (x, y) {
  let result = yield x + y;
  return result;
};

const gen = g(1, 2);
gen.next(); // Object {value: 3, done: false}

gen.next(1); // Object {value: 1, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = 1;
```

next()是将yield表达式替换成一个值

上面代码中，第二个next(1)方法就相当于将yield表达式替换成一个值1。如果next方法没有参数，就相当于替换成undefined。

```
gen.throw(new Error('出错了')); // Uncaught Error: 出错了
// 相当于将 let result = yield x + y
// 替换成 let result = throw(new Error('出错了'));
```

throw()是将yield表达式替换成一个throw语句

```
gen.return(2); // Object {value: 2, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = return 2;
```

return()是将yield表达式替换成一个return语句。

yield* 表达式

yield* 表达式

如果在 Generator 函数内部，调用另一个 Generator 函数。需要在前者的函数体内部，自己手动完成遍历。

```
function* foo() {
  yield 'a';
  yield 'b';
}

function* bar() {
  yield 'x';
  // 手动遍历 foo()
  for (let i of foo()) {
    console.log(i);
  }
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// x
// a
// b
// y
```

右侧代码中，foo和bar都是 Generator 函数，在bar里面调用foo，就需要手动遍历foo。如果有多个 Generator 函数嵌套，写起来就非常麻烦。

yield* 表达式

但是遍历一个就需要一个for循环，太过余麻烦,ES6 提供了yield*表达式，作为解决办法，用来在一个 Generator 函数里面执行另一个 Generator 函数。

```
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}
```

等价于

```
function* bar() {
  yield 'x';
  yield 'a';
  yield 'b';
  yield 'y';
}
```

等价于

```
function* bar() {
  yield 'x';
  for (let v of foo()) {
    yield v;
  }
  yield 'y';
}
```

```
for (let v of bar()){
  console.log(v);
}
// "x"
// "a"
// "b"
// "y"
```

从语法角度看，如果yield表达式后面跟的是一个遍历器对象，需要在yield表达式后面加上星号，表明它返回的是一个遍历器对象。这被称为yield*表达式。

yield* 表达式

```
function* gen(){
  yield* ["a", "b", "c"];
}

gen().next() // { value:"a", done:false }
```

如果yield*后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员。
上面代码中，yield命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器对象。

```
let read = (function* () {
  yield 'hello';
  yield* 'hello';
})();

read.next().value // "hello"
read.next().value // "h"
```

实际上，任何数据结构(比如说字符串)只要有 Iterator 接口，就可以被yield*遍历

Generator函数的其他特性

Generator函数的其他特性之作为对象的属性

```
let obj = {
  * myGeneratorMethod() {
    ...
  }
};
```

等价于

```
let obj = {
  myGeneratorMethod: function* () {
    // ...
  }
};
```

如果一个对象的属性是 Generator 函数，可以简写成上面的形式。

上面代码中，myGeneratorMethod属性前面有一个星号，表示这个属性是一个 Generator 函数。

Generator函数的其他特性之this

Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。

```
function* g() {}

g.prototype.hello = function () {
  return 'hi!';
};

let obj = g();

obj instanceof g // true
obj.hello() // 'hi!'
```

右侧代码表明，Generator 函数g返回的遍历器obj，是g的实例，而且继承了g.prototype。

但是，如果把g当作普通的构造函数，并不会生效，因为g返回的总是遍历器对象，而不是this对象。所以也不能在创造了遍历器对象之后在自己添加写

```
function* g() {
  this.a = 11;
}

let obj = g();
obj.next();
obj.a // undefined
```

Generator函数的其他特性之this

```
function* F() {
  yield this.x = 2;
  yield this.y = 3;
}

new F()
// TypeError: F is not a constructor
```

Generator 函数也不能跟new命令一起用，会报错。

一个无法绑定新的this的函数都没法作为构造函数使用,比如箭头函数

Generator函数的其他特性之this

```
function* F() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
var obj = {};
var f = F.call(obj);

f.next();  // Object {value: 2, done: false}
f.next();  // Object {value: 3, done: false}
f.next();  // Object {value: undefined, done: true}

obj.a // 1
obj.b // 2
obj.c // 3
```

进化

```
function* F() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
var f = F.call(F.prototype);

f.next();  // Object {value: 2, done: false}
f.next();  // Object {value: 3, done: false}
f.next();  // Object {value: undefined, done: true}

f.a // 1
f.b // 2
f.c // 3
```

变通方法:使用一个终极原理，函数的this指的是调用该函数的对象

上面的代码其实简单理解就是，在对象obj上调用了一个生成器函数，那么这个生成器内部代码的this就是这个obj对象

注意:这个返回的f和obj没毛线关系

将obj换成F.prototype。

这就阔以搞定

Generator函数的其他特性之this

```
function* gen() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}

function F() {
  return gen.call(gen.prototype);
}

var f = new F();

f.next();  // Object {value: 2, done: false}
f.next();  // Object {value: 3, done: false}
f.next();  // Object {value: undefined, done: true}

f.a // 1
f.b // 2
f.c // 3
```

实现把一个生成器函数改造成构造函数

Generator函数的其他特性之状态机

```
var ticking = true;
var clock = function() {
  if (ticking)
    console.log('Tick!');
  else
    console.log('Tock!');
  ticking = !ticking;
}
```

```
var clock = function* () {
  while (true) {
    console.log('Tick!');
    yield;
    console.log('Tock!');
    yield;
  }
};
```

上面代码的clock函数一共有两种状态（Tick和Tock），每运行一次，就改变一次状态

上面的 Generator 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅