

## 预习资料

名称	链接	备注
可迭代协议 迭代器协议	<a href="https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Iteration_protocols#%E5%8F%AF%E8%BF%AD%E4%BB%A3%E5%8D%8F%E8%AE">https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Iteration_protocols#%E5%8F%AF%E8%BF%AD%E4%BB%A3%E5%8D%8F%E8%AE</a>	介绍可迭代协议和迭代器协议
协程	<a href="https://cnodejs.org/topic/58dd7a303d476b42d34c911">https://cnodejs.org/topic/58dd7a303d476b42d34c911</a>	介绍协程
co源码	<a href="https://github.com/tj/co">https://github.com/tj/co</a>	课后作业分析的源码

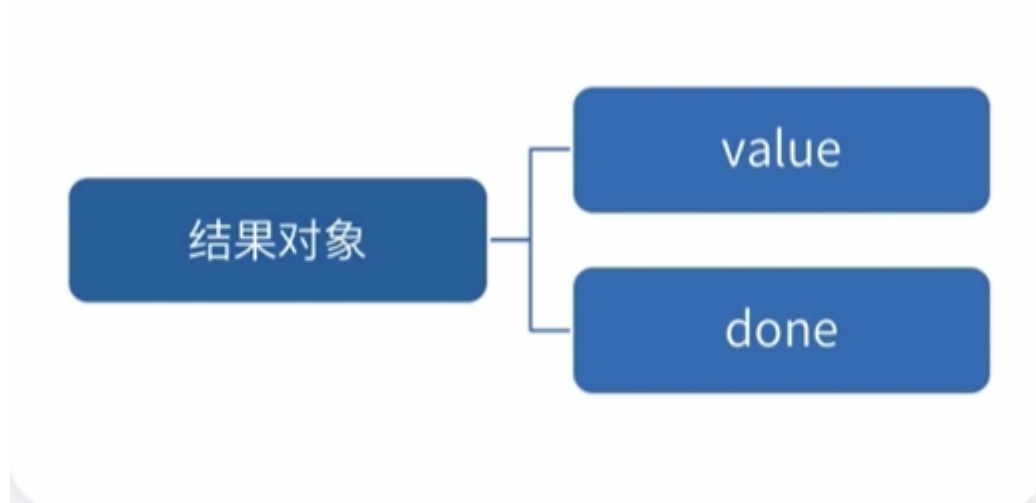
## 一. Generator 函数

generator函数也是解决异步的一个重要解决方法，也是ES6内一块重要的内容。

在学习generator函数前，我们先看两个概念迭代器和生成器

generator的意思是产生，生成的意思，生成器就是generator函数

迭代器：迭代器需要有next方法，执行返回结果对象



我们来看一段代码是用ES5写的一个生成迭代器的函数

```

1 function createIterator(items) {
2     var i = 0;
3     return {
4         next: function() {
5             var done = i >= items.length;
6             var value = !done ? items[i++] : undefined;
7             return {
8                 done: done,
9                 value: value
10            };
11        }
12    };
13 }
14
15 var iterator = createIterator([1, 2, 3]);
16
17 iterator.next();
18 iterator.next();
19 iterator.next();
20 iterator.next();

```

这段代码是生成一个迭代器函数，代码解析：

1. 从`iterator.next()` 看起，`iterator`是用的`createIterator`函数的返回值
2. 再看一下`createIterator`函数，参数`items`是一个数组，定义一个变量`i`，返回值是一个大对象
3. 这个大对象里面有`next`方法，`next`方法里面定义了一个 `done`，`done`是布尔值，`i >= items.length`，`done`为`true`的话，`value`为`undefined`，`done`为`false`的时候，取`items[i++]`的
4. `next`方法再返回一个对象，`{done:done,value:value}`
5. 多次调用`next()`，都是下一次的值

ES6里面又区分了

可迭代协议：将`[Symbol.iterator]`属性定义为一个迭代器对象 内置可迭代对象(`String` `Array` `Map` `Set`) 可满足：

例如代码：

```

1 var someString = "hi";
2 var iterator = someString[Symbol.iterator]();

```

```
3 iterator.next(); // { value: "h", done: false }
4 iterator.next(); // { value: "i", done: false }
5 iterator.next();
```

迭代器协议：生成器协议，该协议定义了什么是迭代器对象。其实迭代器协议很简单，只要实现`.next()`方法（并具有对应语义）即可。该方法返回的对象除`.value`属性外，还应有一个`.done`属性来标识迭代器是否已越过最后一个元素。

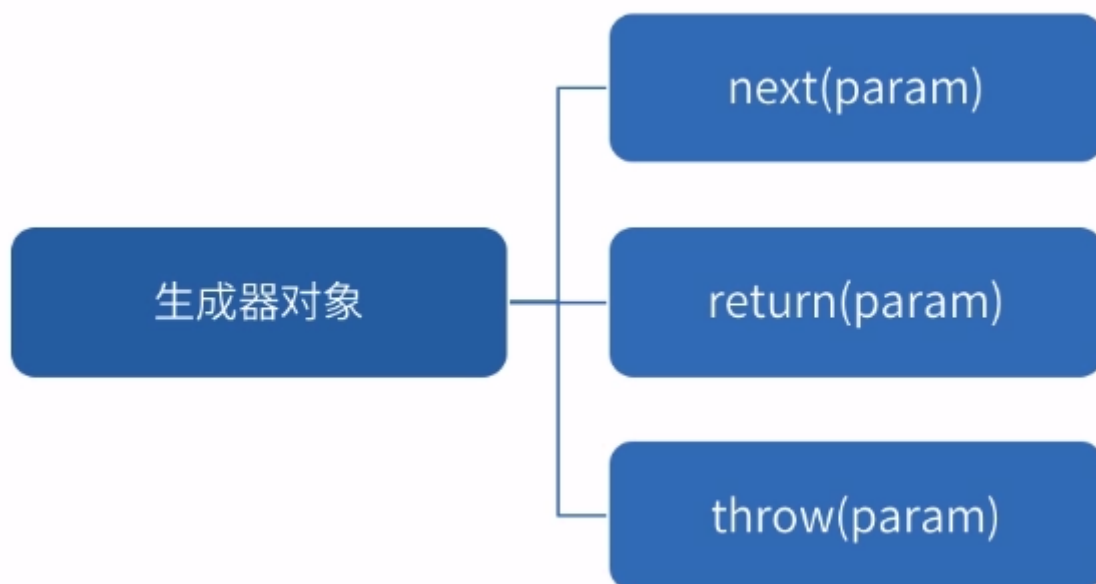
例如代码：

```
1 function* g(){
2   yield 1;
3   yield 2;
4 }
5 var iterator = g();
6 console.log(iterator.next()); // { value: 1, done: false }
7 console.log(iterator.next()); // { value: 2, done: false }
8 console.log(iterator.next()); // { value: undefined, done: true }
```

## generator函数

上面我们用ES5模拟了一个产生迭代器的函数，ES6我们有一个函数，他就是generator函数，可以直接生成迭代器。

1. ES6异步编程解决方案
2. 声明：通过`function *` 声明
3. 返回值：符合可迭代协议和迭代器协议的生成器对象(两种协议都满足)
4. 在执行时暂停，又能从暂停处继续执行(这个和普通函数还是有区别的)



执行了generator函数就生产了一个生成器对象，生成器对象的原型上面有三个方法

1. next方法(就是上面自定义迭代器中next方法)
2. return方法
3. throw方法

### next(param)方法

generator函数除了上面说的四个特别的地方，还有一个不同的地方就是yield关键字，yield关键字只能出现在Generator函数，如果在普通的函数内适用yield函数是会报语法错误的。

yield关键字用来暂停和恢复生成器函数，它是通过什么来实现的呢？代码层面的话就是通过yield关键字来实现。

然后我们在说一下next执行：

1. 遇到yield就暂停，将紧跟yield表达式的值作为返回的对象的value
2. 没有yield，一直执行到return，将return的值作为返回对象的value
3. 没有return，将undefined作为返回对象的value

接着我们说一下next参数

1. next方法可以带参数，该参数会被当作上一个yield表达式的值

下面我看一下相关generator函数的示例代码

```
1 function* createIterator(){
2   let first = yield 1
3   let second = yield first + 1
4   yield second + 3
5
6 }
```

```

7
8 let iterator = createIterator()
9 iterator.next() //{value: 1, done: false}
10 iterator.next() //{value: NaN, done: false}
11 iterator.next(10) //{value:13,done:false}
12 iterator.next(10) //{value:undefined,done:true}

```

根据上面相关next执行和next参数相关规则，进行分析：

1. 调用createIterator()函数生产一个迭代器
2. 看第一个的next()执行,遇到yield就暂停，yield表达式的值为1，作为返回对象的value，done还未执行完，所有结果对象为 {value:1,done:false}
3. 看第二个next的执行，因为第一次next执行到yield就暂停,yield1的值返回。next的参数为空，默认返回一个undefined替换上一个yield表达式的值，所以first + 1 为NaN，所以结果对象的为 {value:NaN,done:false}
4. 看第三个next的执行,同第二个执行一样，但是第三个next携带了参数，该参数会被当作上一个yield表达式的值，所以结果对象为 {value:13,done:false}
5. 看第四个next的执行，没有yield，也没有显示的return，就隐式的return undefined 将默认的undefined作为返回对象的value，执行完毕。done为true。所以结果对象为{value:undefined,done:true}

**yield\*** (yield关键字是用来暂停的，yield\*关键字是用来委托给其他可迭代对象)  
作用:复用生成器,把控制权交付给其他的可迭代对象，遇到 yield\* 是不会暂停的。

我们看下示例代码:

```

1 function* generator1(){
2   yield 1
3   yield 2
4 }
5 function* generator2(){
6   yield 100
7   yield* generator1()
8   yield 200
9   return 400
10 }
11
12 let g2 = generator2()
13 g2.next() //{value: 100, done: false}
14 g2.next() //{value: 1, done: false}
15 g2.next() //{value: 2, done: false}

```

```
16 g2.next() //{value: 200, done: false}
17 g2.next() //{value: 400, done: true}
```

代码解析：

1. g2的第一个next 返回结果对象为 {value: 100, done: false}
2. g2的第二个next 是 yield \* 委托其他的可迭代对象 这里直接调用 generator1() 把控制权都交给generator1。所以返回的结果对象为 { value: 1, done: false}
3. g2的第三个next 还是generator1中，返回的结果对象 { value: 1, done: false}
4. g2的第四个next 回到了generator2，返回的结果对象 { value: 200, done: false}
5. g2的第五个next，执行完毕，看又没有return 有return，终结遍历，done为true。把return的值当成结果对象的value {value: 400, done: true}

## return(param)方法

给定param值终结遍历器 param可缺省

```
1 function* generator(){
2   yield 1
3   yield 2
4   yield 3
5 }
6
7 let g = generator()
8 g.next() //{value: 1, done: false}
9 g.return(32)//{value: 32, done: true}
```

代码解析：

1. return()方法终结遍历，返回结果对象的value为param的值 {value: 1, done: true}
2. 如果没有param，返回结果对象的value为undefined {value: 1, done: true}

## throw(param)方法

让生成器对象内部抛出错误

看代码

```

1 function* createIterator() {
2     let first = yield 1;
3     let second;
4     try {
5         second = yield first + 2;
6     } catch (e) {
7         second = 6;
8     }
9     yield second + 3;
10 }
11 let iterator = createIterator();
12
13 iterator.next();           {value: 1, done: false}
14 iterator.next(10);         {value: 12, done: false}
15 iterator.throw(new Error('error')); {value: 9, done: false}
16 iterator.next();           {value: undefined, done: true}

```

代码执行:

1. 执行第一个next的时候, 结果对象返回的是 {value:1,done:false}
2. 执第二个next的时候, 结果对象返回的是 {value:12,done:false}
3. 执行到throw的时候, 抛出异常, 继续执行catch里面, second = 6, 直到遇到yield 暂停, 返回对象为 {value:9,done:false}
4. 下一个next, 返回对象为 {value:undefined,done:true}

## 二. Generator 的实现原理

要了解generator的实现原理, 首先要了解一下协程 (等下补充, 先看下代码示例)

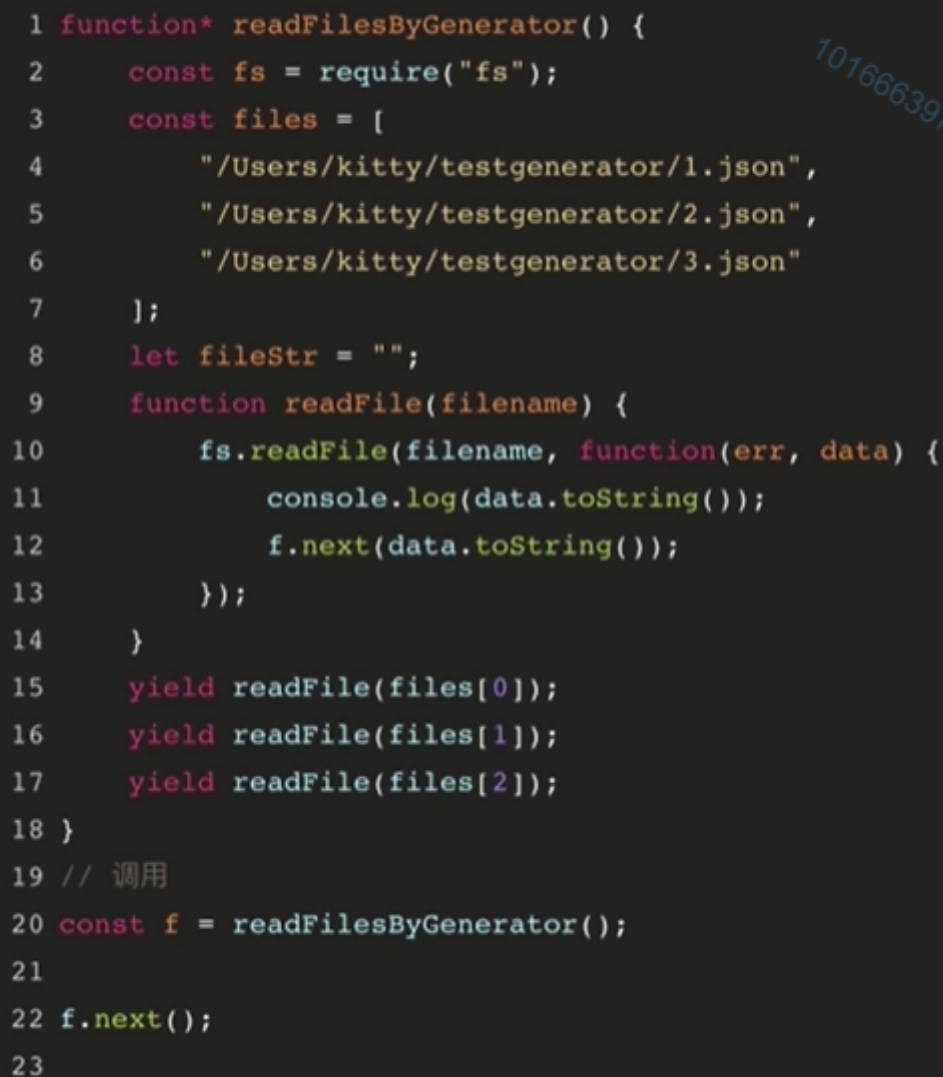
```
1 function readFilesByCallback() {
2     const fs = require("fs");
3     const files = [
4         "/Users/kitty/testgenerator/1.json",
5         "/Users/kitty/testgenerator/2.json",
6         "/Users/kitty/testgenerator/3.json"
7     ];
8     fs.readFile(files[0], function(err, data) {
9         console.log(data.toString());
10        fs.readFile(files[1], function(err, data) {
11            console.log(data.toString());
12            fs.readFile(files[2], function(err, data) {
13                console.log(data.toString());
14            });
15        });
16    });
17 }
18 // 调用
19 readFilesByCallback();
```

上述代码是想读取三个文件：然后按顺序读取三个文件，打印出来

1. 首先看readFile是异步读取文件，readFileSync是同步读取
2. 如果按照顺序调用，不放在回调里面，同步的执行的话，每个文件的大小的不同，执行时间也不一致，可能出现不是的顺序也不一样。
3. 放在回调里面虽然可以实现，但是回调嵌套多层的话就会出现回调地狱，也是不可取的

所以采用generator方式





```

1 function* readFilesByGenerator() {
2     const fs = require("fs");
3     const files = [
4         "/Users/kitty/testgenerator/1.json",
5         "/Users/kitty/testgenerator/2.json",
6         "/Users/kitty/testgenerator/3.json"
7     ];
8     let fileStr = "";
9     function readFile(filename) {
10         fs.readFile(filename, function(err, data) {
11             console.log(data.toString());
12             f.next(data.toString());
13         });
14     }
15     yield readFile(files[0]);
16     yield readFile(files[1]);
17     yield readFile(files[2]);
18 }
19 // 调用
20 const f = readFilesByGenerator();
21
22 f.next();
23

```

### 代码分析

1. 先看调用，readFilesByGenerator函数，f.next()执行
2. next 遇到yield关键字就暂停，但是yield后面的表达式 readFile(files[0])还是会执行的。
3. readFile函数的执行，里面是一个异步读取文件的过程，然后打印出文件，然后读取文件的回调里面，执行下一个next(),遇到yield暂停，就会执行readFile(files[1])。以此类推 执行完readFile(files[2])

### 缺陷:

外部分 调用函数 f 和内部的readFile函数耦合在一起了，十分不雅观。

有什么办法可以让generator函数包装一下，不需要在内部去调用外部的变量，接下来就得用到Thunk函数得作用了。

## 二. Thunk函数

我们先看一下求值策略

- **求值策略** 一种是传值调用 一种是传名调用 我们看下`sum(x+1,x+2)`

传值调用就是 先算出入参表达式得值 例如:先算出 `x+1`和`x+2`, 得到结果后再传给 `sum`函数

传名调用 等到调用`sum`函数得时候 具体用到才会去计算`x+1`和`x+2`的值

- `thunk`函数是传名调用的实现方式之一
- 可以实现自动执行的generator函数

js中的thunk函数是怎么样子的呢? JavaScript 语言是传值调用, 它的 Thunk 函数含义有所不同。在 JavaScript 语言中, Thunk 函数替换的不是表达式, 而是多参数函数, 将其替换成单参数的版本, 且只接受回调函数作为参数。如下代码

```
1 // 正常版本的readFile (多参数版本)
2 fs.readFile(fileName, callback);
3
4 // Thunk版本的readFile (单参数版本)
5 var readFileThunk = Thunk(fileName);
6 readFileThunk(callback);
7
8 var Thunk = function (fileName){
9   return function (callback){
10     return fs.readFile(fileName, callback);
11   };
12 };
```

上面代码中, `fs` 模块的 `readFile` 方法是一个多参数函数, 两个参数分别为文件名和回调函数。经过转换器处理, 它变成了一个单参数函数, 只接受回调函数作为参数。这个单参数版本, 就叫做 Thunk 函数。

下面我们用Thunk函数实现上面按顺序读取三个文件

```

const fs = require("fs");
const Thunk = function(fn) {
  return function(...args) {
    return function(callback) {
      return fn.call(this, ...args, callback);
    };
  };
};
const readFileThunk = Thunk(fs.readFile);

function run(fn) {
  var gen = fn();
  function next(err, data) {
    var result = gen.next(data);
    if (result.done) return;
    result.value(next);
  }
  next();
}

```

```

const g = function*() {
  const s1 = yield readFileThunk("/Users/kitty/testgenerator/1.json");
  console.log(s1.toString());
  const s2 = yield readFileThunk("/Users/kitty/testgenerator/2.json");
  console.log(s2.toString());
  const s3 = yield readFileThunk("/Users/kitty/testgenerator/3.json");
  console.log(s3.toString());
};

run(g);

```

代码分析：

1. 根据JS Thunk函数的定义，是将多参数函数，将其替换成单参数的版本，且只接受回调函数作为参数，所以把Thunk定义成一个Thunk函数。把fs异步读取文件的方法参数，转化成单参数版本。
2. 定义一个readFileTunk常量接收Thunk函数调用。他返回的是function(fn)

3. 再定义一个普通run函数，和generator函数g
4. run(g)调用run函数，把g函数作为参数
5. 分析一下执行过程，g函数作为run函数的参数，调用run函数，看run函数体，gen是generator函数生成器返回的结果对象。gen调用next方法，开始直接g函数，遇到yield暂停，然后执行readFileThunk函数的调用，它返回的是function(...args)函数，所以执行第一个next后，返回的结果对象的是{value:function,done:false},所以run函数中result就是该对象{value:function,done:false},result的value就是function(callback)，result.value(next) 就是调用function(callback)，并把next方法当作callback参数传入，当第一个文件读取完之后，执行回调函数next方法。
6. 因为next方法中有执行gen.next()所以这个会继续遍历，执行s2,s3直到遍历结束返回{value:undefined,done:true}
7. 这样就按照顺序读取了三个文件