async + await 原理

今天我们要说的主人公是ES7中新引入的语法“async + await”，被称为**异步的完美解决方案**。当然了也不是脑门一热，一拍脑门，说我们搞一个异步的完美解决方案吧，然后就叫async，await。如果凡事可以通过拍脑们的方式，我们现在进展远远不止于此。（PS: 一拍脑门，征服宇宙，↖(^ω^)↗）

好了，说回来，今天的主人公 async await .他们的前身，来源于 generator + co，这样的一个组合，来实现了一个异步解决的最优解，不单纯的靠回调函数和Promise。而是把**异步代码，同步化**。这个时候就会有人问，那是怎么做到的捏？这个就是我们今天要研究的内容，当然了研究懂了，async和await的原理也就懂了。

generator直译过来叫做生成器。生成器干嘛的，用来生成迭代器。那迭代器又是什么？
迭代器：ES6中提出的一种统一话迭代数据结构的解决方案。

好了啊，说了半天，好像也没说明白是干啥滴，接下来我们们用代码说话，这样对于上面的概念会解释的更清晰。

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

这样的一个函数就叫做生成器函数，但是我们看这个函数结构好像和我们的普通函数不一样，里面出现了，两个我们之前不在函数中使用的两个标志
一个 *
一个 yield 关键字。
*** 作用： 用于说明，我接下来要声明一个函数，不是普通函数了，叫做生成器函数了。**
yield关键字必须陪在生成器函数使用，不能单独使用。
接下来看看这个函数该怎么用和我们的普通函数有啥不一样的东西。

```
function *generator() {
    yield 1;
    yield 2;
    yield 3;
```

```
}

let it = generator(); // 这个it 就是一个迭代器。
console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

执行结果：

```
{ value: 1, done: false }
{ value: 2, done: false }
{ value: 3, done: false }
{ value: undefined, done: true }
{ value: undefined, done: true }
```

从这里我们可以看出来generator函数返回结果是一个迭代器。迭代器函数，可以**无限次的调用next函数**，来依次输出 yield 后面的值。上面我特意执行了五次，前三次next把对应的三个yield后面的内容输出，之后两次，返回的值是undefined。 done表示是否迭代完毕。作为迭代器本身是一个类似链表的结构，可以通过next不断的指针后移，然后读取相关的内容。直到最后一个 {value: undefined , done: true} 。

其实从结果上面我们能看出来迭代器中的代码是可以分片执行的。接下来写一个基于上面的例子的改写;

```
function *generator() {
    console.log("第一次next")
    yield 1;
    console.log("第二次next")
    yield 2;
    console.log("第三次next")
    yield 3;
}

let it = generator(); // 这个it 就是一个迭代器。
console.log("start")
console.log(it.next());
console.log("1")
console.log(it.next());
```

```
15 console.log("2")
16 console.log(it.next());
17 console.log("3")
```

执行顺序是

所以说 对于generator函数是可以分步执行的。

接下来说说 yield 返回值。

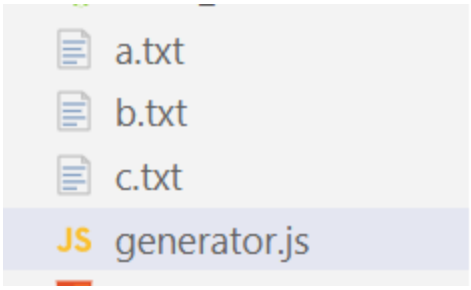
```
1 function *generator() {
2     let a = yield 1;
3     console.log("a的值:" + a )
4     let b = yield 2;
5     console.log("b的值:" + b )
6     yield 3;
7 }
8
9 let it = generator(); // 这个it 就是一个迭代器。
10 console.log("第1次执行",it.next(10000));
11 console.log("第2次执行",it.next(1));
12 console.log("第3次执行",it.next(2));
```

```
[Running] node "c:\Users\DYZ96\Desktop\前端知识\generator\generator.js"
第1次执行 { value: 1, done: false }
a的值:1
第2次执行 { value: 2, done: false }
b的值:2
第3次执行 { value: 3, done: false }
第4次执行 { value: undefined, done: true }
第5次执行 { value: undefined, done: true }
```

这三个框分别代表三次的执行性结果。
我们可以看出：
第一次的next里面传入的值，无意义。
从第二次开始。每次调用next传入值，作为上一次yeild的返回值。
这样的话，我们就可以在外边传入接下来我们要操作的值。

接下来，有一个这样的需求。我们需要陆续读取两个文件夹内的内容。作为下一次读取文件的名称。通过一个generator函数实现。

文件目录如下：



a.txt 存的内容是 "b.txt", b.txt 存的内容是 "c.txt" 通过三步读取操作去读 c.txt的内容。

```
const fs = require("fs").promises
function *read() {
    let value = yield fs.readFile("a.txt", "utf-8");
    let value2 = yield fs.readFile(value, "utf-8");
    let value3 = yield fs.readFile(value2, "utf-8");
    return value3;
}

let it = read();
let {done, value} = it.next();
value.then(data => {
    console.log(data, "1");
    let {done, value} = it.next(data);
    value.then(data => {
        console.log(data, "2");
        let {done, value} = it.next(data);
        value.then(data => {
            console.log(data, "3");
            let {done, value} = it.next(data);
            console.log(value)
```

```
        })
    })
})
```

执行结果：

```
[Running] node "c:\Users\DYZ96\Desktop\前端知识\generato
b.txt 1
c.txt 2
渡一教育Web 3
渡一教育Web
```

这里的 requrie("fs").promises 指的是 Promise化的文件API。所以可以看到如下的这种代码形式。当然了，我不想说 回调地狱这个概念。而是说这种代码，里面的的这种类似递归的结构。

其实 TJ大佬写了一个库 Co 用来解决我们上面的问题。

```
const fs = require("fs").promises
const co = require("co")

// 使用Generator函数,读取两个文件的内容写到第三个文件中，文件名称可以自行传入，例如 读取a.txt, 和 b.txt 然后写到 c.txt; co
function *read() {
    let value = yield fs.readFile("a.txt", "utf-8");
    let value2 = yield fs.readFile(value, "utf-8");
    let value3 = yield fs.readFile(value2, "utf-8");
    return value3;
}
co(read()).then(data => {
    console.log(data)
})
```

输出结果：

```
渡一教育Web
```

其实跟我们上面最终输出结果一样，只不过，我们在前面做了若干的辅助输出。

接下来我们来搞一搞CO函数到底怎么写出来的，我们来模拟一个。

```
const fs = require("fs").promises
// const co = require("co")

// 使用Generator函数,读取两个文件的内容写到第三个文件中，文件名称可以自行传入，例如 读取a.txt, 和 b.txt 然后写到 c.txt; co
function *read() {
    let value = yield fs.readFile("a.txt", "utf-8");
    let value2 = yield fs.readFile(value, "utf-8");
    let value3 = yield fs.readFile(value2, "utf-8");
    return value3;
}
function co(it) {
    return new Promise((resolve, reject) => {
        function next(data) {
            // 第一次next中可以不用传值。
            let {done, value} = it.next(data);
            if(done) {
                resolve(value)
            } else {
                Promise.resolve(value).then(data => {
                    next(data)
                })
            }
        }
        next()
    })
}
co(read()).then(data => {
    console.log(data)
})
```

我们实现Co之后，跟上面的结果其实是一样的。当然了，我们这里是一个简版的CO，如果向更深如了解CO的同学可以去github去研究研究代码量也不多不过几百行。
这就是一个 co + Generator的模型。
在这个基础上面扩展出来 async, await我们看看改完之后是什么样。

```
const fs = require("fs").promises
// const co = require("co")
```

```
3
4 // 使用Generator函数,读取两个文件的内容写到第三个文件中，文件名称可以自行传入，例
  如 读取a.txt, 和 b.txt 然后写到 c.txt; co
5 async function read() {
6     let value = await fs.readFile("a.txt", "utf-8");
7     let value2 = await fs.readFile(value, "utf-8");
8     let value3 = await fs.readFile(value2, "utf-8");
9     return value3;
10 }
11
12 read().then(data => {
13     console.log(data)
14 })
```

我们把原来的 * 改成了 async 把 yield 改成了 await。同样，通过 async声明的函数返回仍然是一个Promise。而且对于await必须和 async共同使用。