

# 7张图，20分钟就能搞定的async/await原理！为什么要拖那么久？



Sunshine\_Lin Lv6

2021年09月12日 13:04 · 阅读 44200

+ 关注

 7张图，20分钟就能搞定的async/await原理！为什么要拖那么久？

## 前言

大家好，我是林三心，**以最通俗的话，讲最难的知识点**是我写文章的宗旨

之前我发过一篇[手写Promise原理，最通俗易懂的版本！！](#)，带大家基本了解了 `Promise` 内部的实现原理，而提到 `Promise`，就不得不提一个东西，



1k+



78



收藏

async/await 吧!!!

## async/await用法

其实你要实现一个东西之前，最好是先搞清楚这两样东西

- 这个东西有什么用？
- 这个东西是怎么用的？

### 有什么用？

async/await 的用处就是：**用同步方式，执行异步操作**，怎么说呢？举个例子

比如我现在有一个需求：先请求完 **接口1**，再去请求 **接口2**，我们通常会这么做

```
function request(num) { // 模拟接口请求
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(num * 2)
    }, 1000)
  })
}

request(1).then(res1 => {
  console.log(res1) // 1秒后 输出 2
})
```

js 复制代码

 1k+

 78

 收藏

```
request(2).then(res2 => {  
  console.log(res2) // 2秒后 输出 4  
})  
})
```

或者我现在又有一个需求：先请求完 **接口1**，再拿 **接口1** 返回的数据，去当做 **接口2** 的请求参数，那我们也可以这么做

```
request(5).then(res1 => {  
  console.log(res1) // 1秒后 输出 10  
  
  request(res1).then(res2 => {  
    console.log(res2) // 2秒后 输出 20  
  })  
})
```

js 复制代码

其实这么做是没问题的，但是如果嵌套的多了，不免有点不雅观，这个时候就可以用 **async/await** 来解决了

```
async function fn () {  
  const res1 = await request(5)  
  const res2 = await request(res1)  
  console.log(res2) // 2秒后输出 20  
}  
fn()
```

js 复制代码

## 是怎么用？



1k+



78



收藏

还是用刚刚的例子

需求一：

```
async function fn () {  
  await request(1)  
  await request(2)  
  // 2秒后执行完  
}  
fn()
```

js 复制代码

需求二：

```
async function fn () {  
  const res1 = await request(5)  
  const res2 = await request(res1)  
  console.log(res2) // 2秒后输出 20  
}  
fn()
```

js 复制代码



```
async function fn () {  
  const res1 = await request(5)  ← 等待1秒  
  const res2 = await request(res1) ← 等待1秒  
  console.log(res2) // 2秒后输出 20  
}
```

@稀土掘金技术社区

其实就类似于生活中的 **排队**，咱们生活中排队买东西，肯定是要上一个人买完，才轮到下一个人。而上面也一样，在 `async` 函数中，`await` 规定了异步操作只能一个一个排队执行，从而达到**用同步方式，执行异步操作**的效果，这里注意了：**`await`只能在`async`函数中使用，不然会报错哦**

刚刚上面的例子 `await` 后面都是跟着异步操作 `Promise`，那如果不接 `Promise` 会怎么样呢？

```
function request(num) { // 去掉Promise  
  setTimeout(() => {  
    console.log(num * 2)  
  }, 1000)  
}  
  
async function fn() {  
  await request(1) // 2
```

js 复制代码

👍 1k+

💬 78

★ 收藏

```
}  
fn()
```

可以看出，如果 `await` 后面接的不是 `Promise` 的话，有可能其实是达不到 `排队` 的效果的

说完 `await`，咱们聊聊 `async` 吧，`async` 是一个位于function之前的前缀，只有 `async函数` 中，才能使用 `await`。那 `async` 执行完是返回一个什么东西呢？

```
async function fn () {}  
console.log(fn) // [AsyncFunction: fn]  
console.log(fn()) // Promise {<fulfilled>: undefined}
```

js 复制代码

可以看出，`async函数` 执行完会自动返回一个状态为 `fulfilled` 的 `Promise`，也就是成功状态，但是值却是 `undefined`，那要怎么才能使值不是 `undefined`呢？很简单，函数有 `return` 返回值就行了

```
async function fn (num) {  
  return num  
}  
console.log(fn) // [AsyncFunction: fn]  
console.log(fn(10)) // Promise {<fulfilled>: 10}  
fn(10).then(res => console.log(res)) // 10
```

js 复制代码

可以看出，此时就有值了，并且还能使用 `then方法` 进行输出

✂

👍 1k+

💬 78

★ 收藏

总结一下 `async/await` 的知识点

- `await`只能在`async`函数中使用，不然会报错
- `async`函数返回的是一个`Promise`对象，有无值看有无`return`值
- `await`后面最好是接`Promise`，虽然接其他值也能达到排队效果
- `async/await`作用是**用同步方式，执行异步操作**

## 什么是语法糖？

前面说了，`async/await` 是一种 **语法糖**，诶！好多同学就会问，啥是 **语法糖** 呢？我个人理解就是，**语法糖** 就是一个东西，这个东西你就算不用他，你用其他手段也能达到这个东西同样的效果，但是可能就没有这个东西这么方便了。

- 举个生活中的例子吧：你走路也能走到北京，但是你坐飞机会更快到北京。
- 举个代码中的例子吧：ES6的 `class` 也是语法糖，因为其实用普通 `function` 也能实现同样效果

回归正题，`async/await` 是一种 **语法糖**，那就说明用其他方式其实也可以实现他的效果，我们今天就是讲一讲怎么去实现 `async/await`，用到的是ES6里的 **迭代函数—generator函数**

## generator函数

### 基本用法



generator函数 跟普通函数在写法上的区别就是，多了一个星号 `*`，并且只有在 generator函数 中才能使用 `yield`，什么是 `yield` 呢，他相当于 generator函数 执行的 中途暂停点，比如下方有3个暂停点。而怎么才能暂停后继续走呢？那就得使用到 `next方法`，`next方法` 执行后会返回一个对象，对象中有 `value` 和 `done` 两个属性

- `value`：暂停点后面接的值，也就是`yield`后面接的值
- `done`：是否generator函数已走完，没走完为`false`，走完为`true`

```
function* gen() {  
  yield 1  
  yield 2  
  yield 3  
}  
  
const g = gen()  
  
console.log(g.next()) // { value: 1, done: false }  
console.log(g.next()) // { value: 2, done: false }  
console.log(g.next()) // { value: 3, done: false }  
console.log(g.next()) // { value: undefined, done: true }
```

js 复制代码

可以看到最后一个是`undefined`，这取决于你generator函数有无返回值


```
function* gen() {  
  yield 1  
  yield 2  
  yield 3  
  return 4  
}  
  
const g = gen()
```

js 复制代码





```
console.log(g.next()) // { value: 2, done: false }
console.log(g.next()) // { value: 3, done: false }
console.log(g.next()) // { value: 4, done: true }
```

 截屏2021-09-11 下午9.46.17.png

## yield后面接函数

yield后面接函数的话，到了对应暂停点yield，会马上执行此函数，并且该函数的执行返回值，会被当做此暂停点对象的 **value**

```
function fn(num) {
  console.log(num)
  return num
}
function* gen() {
  yield fn(1)
  yield fn(2)
  return 3
}
const g = gen()
console.log(g.next())
// 1
// { value: 1, done: false }
console.log(g.next())
// 2
// { value: 2, done: false }
```

js 复制代码



## yield后面接Promise

前面说了，函数执行返回值会当做暂停点对象的value值，那么下面例子就可以理解了，前两个的value都是pending状态的Promise对象

```
function fn(num) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(num)  
    }, 1000)  
  })  
}  
  
function* gen() {  
  yield fn(1)  
  yield fn(2)  
  return 3  
}  
  
const g = gen()  
console.log(g.next()) // { value: Promise { <pending> }, done: false }  
console.log(g.next()) // { value: Promise { <pending> }, done: false }  
console.log(g.next()) // { value: 3, done: true }
```

js 复制代码



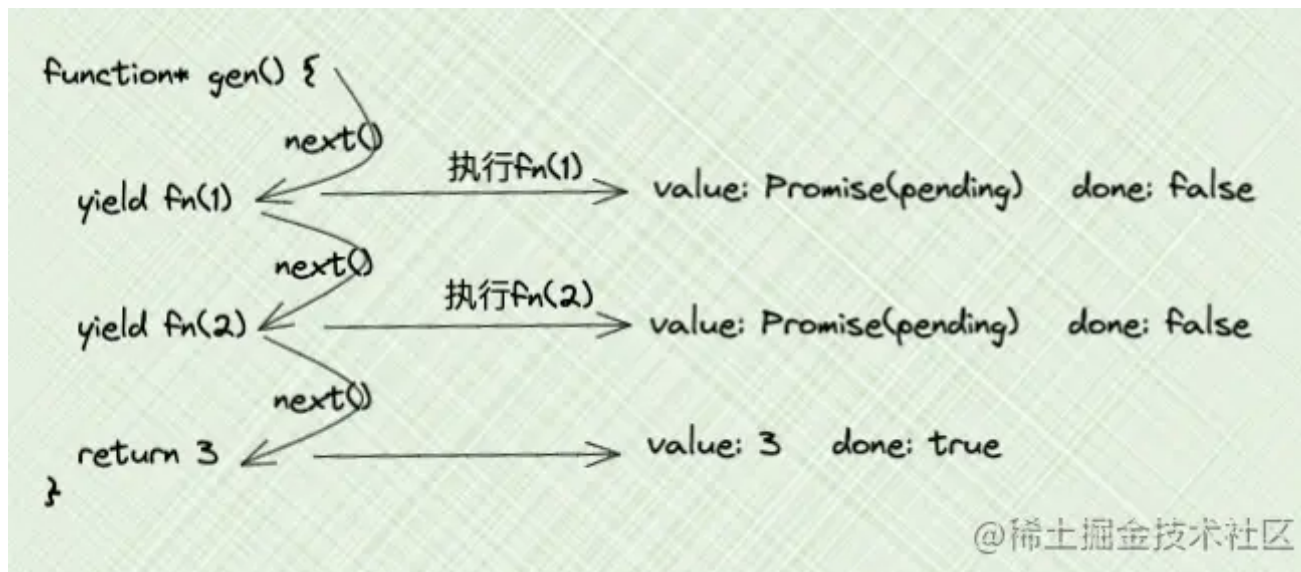
1k+



78



收藏



其实我们想要的结果是，两个Promise的结果 **1 和 2**，那怎么做呢？很简单，使用Promise的then方法就行了

```
const g = gen()
const next1 = g.next()
next1.value.then(res1 => {
  console.log(next1) // 1秒后输出 { value: Promise { 1 }, done: false }
  console.log(res1) // 1秒后输出 1

  const next2 = g.next()
  next2.value.then(res2 => {
    console.log(next2) // 2秒后输出 { value: Promise { 2 }, done: false }
    console.log(res2) // 2秒后输出 2
    console.log(g.next()) // 2秒后输出 { value: 3, done: true }
  })
})
```

js 复制代码



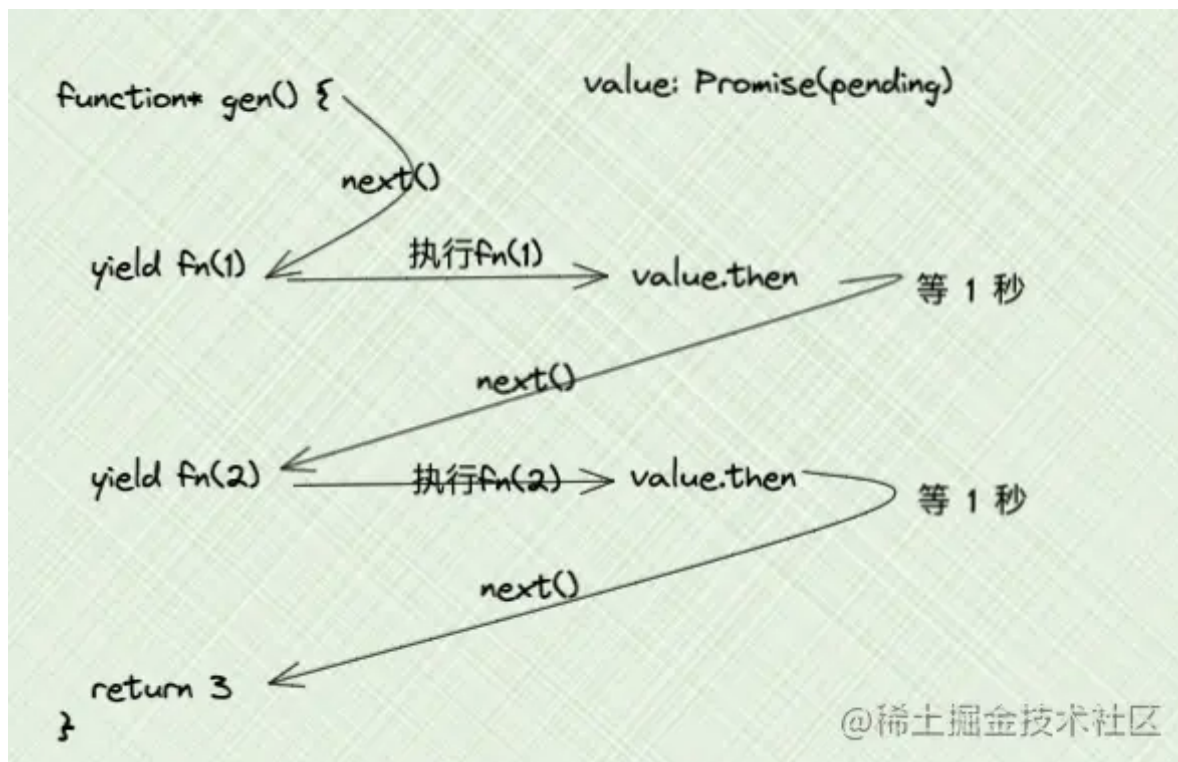
1k+



78



收藏



## next函数传参

generator函数可以用 **next方法** 来传参，并且可以通过 **yield** 来接收这个参数，注意两点

- 第一次next传参是没用的，只有从第二次开始next传参才有用
- next传值时，要记住顺序是，先右边yield，后左边接收参数



1k+



78



收藏

```
console.log(num1)
const num2 = yield 2
console.log(num2)
return 3
}
const g = gen()
console.log(g.next()) // { value: 1, done: false }
console.log(g.next(11111))
// 11111
// { value: 2, done: false }
console.log(g.next(22222))
// 22222
// { value: 3, done: true }
```

 截屏2021-09-11 下午10.53.02.png

## Promise+next传参

前面讲了

- yield后面接Promise
- next函数传参

那这两个组合起来会是什么样呢？

js 复制代码



```
    setTimeout(() => {
      resolve(nums * 2)
    }, 1000)
  })
}

function* gen() {
  const num1 = yield fn(1)
  const num2 = yield fn(num1)
  const num3 = yield fn(num2)
  return num3
}

const g = gen()
const next1 = g.next()
next1.value.then(res1 => {
  console.log(next1) // 1秒后同时输出 { value: Promise { 2 }, done: false }
  console.log(res1) // 1秒后同时输出 2

  const next2 = g.next(res1) // 传入上次的res1
  next2.value.then(res2 => {
    console.log(next2) // 2秒后同时输出 { value: Promise { 4 }, done: false }
    console.log(res2) // 2秒后同时输出 4

    const next3 = g.next(res2) // 传入上次的res2
    next3.value.then(res3 => {
      console.log(next3) // 3秒后同时输出 { value: Promise { 8 }, done: false }
      console.log(res3) // 3秒后同时输出 8

      // 传入上次的res3
      console.log(g.next(res3)) // 3秒后同时输出 { value: 8, done: true }
    })
  })
})
```



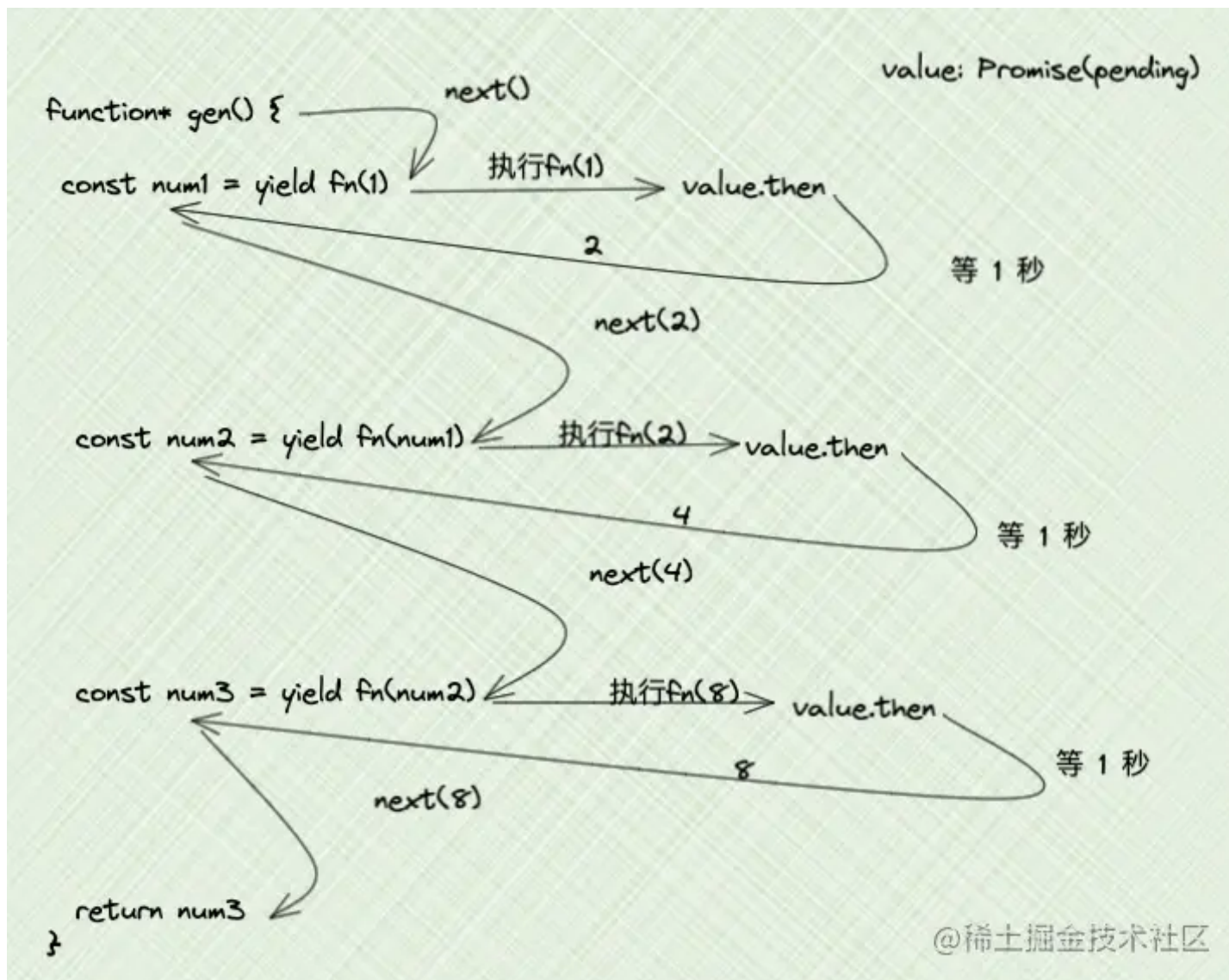
1k+



78



收藏



1k+



78



收藏

其实上方的 `generator函数` 的 `Promise+next传参`，就很像 `async/await` 了，区别在于

- `gen`函数执行返回值不是`Promise`，`asyncFn`执行返回值是`Promise`
- `gen`函数需要执行相应的操作，才能等同于`asyncFn`的排队效果
- `gen`函数执行的操作是不完善的，因为并不确定有几个`yield`，不确定会嵌套几次



1k+



78



收藏



```
function* gen() {
  const num1 = yield fn(1)
  const num2 = yield fn(num1)
  const num3 = yield fn(num2)
  return num3
}
```

+

```
const g = gen()
const next1 = g.next()
next1.value.then(res1 => {
```

=

```
async function asyncFn() {
  const num1 = await fn(1)
  const num2 = await fn(num1)
  const num3 = await fn(num2)
  return num3
}
```

```
const next2 = g.next(res1) // 传入上次的res1
next2.value.then(res2 => {
```

```
const next3 = g.next(res2) // 传入上次的res2
next3.value.then(res3 => {
```

```
  // 传入上次的res3
  console.log(g.next(res3))
```

```
})
```

```
})
```

```
})
```

@稀土掘金技术社区

那我们怎么办呢？我们可以封装一个高阶函数。什么是 **高阶函数** 呢？**高阶函数** 的特点是：**参数是函数，返回值也可以是函数**。下方的 `highorderFn` 就是一个 **高阶函数**



1k+



78



收藏

[js](#) [复制代码](#)

```
function highorderFn(函数) {  
  // 一系列处理  
  
  return 函数  
}
```

我们可以封装一个高阶函数，接收一个generator函数，并经过一系列处理，返回一个具有async函数功能的函数

[js](#) [复制代码](#)

```
function generatorToAsync(generatorFn) {  
  // 经过一系列处理  
  
  return 具有async函数功能的函数  
}
```

## 返回值Promise

之前我们说到，async函数的执行返回值是一个Promise，那我们要怎么实现相同的结果呢

[js](#) [复制代码](#)

```
function* gen() {  
  
}  
  
const asyncFn = generatorToAsync(gen)
```



其实很简单，`generatorToAsync`函数里做一下处理就行了

```
function* gen() {  
  
}  
  
function generatorToAsync (generatorFn) {  
  return function () {  
    return new Promise((resolve, reject) => {  
  
    })  
  }  
}  
  
const asyncFn = generatorToAsync(gen)  
  
console.log(asyncFn()) // Promise
```

js 复制代码

## 加入一系列操作

咱们把之前的处理代码，加入 `generatorToAsync`函数 中

```
function fn(nums) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(nums * 2)  
    }, 1000)  
  })  
}
```

js 复制代码



1k+



78



收藏

```
}  
  
function* gen() {  
  const num1 = yield fn(1)  
  const num2 = yield fn(num1)  
  const num3 = yield fn(num2)  
  return num3  
}  
  
function generatorToAsync(generatorFn) {  
  return function () {  
    return new Promise((resolve, reject) => {  
      const g = generatorFn()  
      const next1 = g.next()  
      next1.value.then(res1 => {  
  
        const next2 = g.next(res1) // 传入上次的res1  
        next2.value.then(res2 => {  
  
          const next3 = g.next(res2) // 传入上次的res2  
          next3.value.then(res3 => {  
  
            // 传入上次的res3  
            resolve(g.next(res3).value)  
          })  
        })  
      })  
    })  
  })  
}  
  
const asyncFn = generatorToAsync(gen)
```



可以发现，咱们其实已经实现了以下的 `async/await` 的结果了

```
async function asyncFn() {
  const num1 = await fn(1)
  const num2 = await fn(num1)
  const num3 = await fn(num2)
  return num3
}
asyncFn().then(res => console.log(res)) // 3秒后输出 8
```

js 复制代码

## 完善代码

上面的代码其实都是死代码，因为一个`async`函数中可能有2个`await`，3个`await`，5个`await`，其实`await`的个数是不确定的。同样类比，`generator`函数中，也可能有2个`yield`，3个`yield`，5个`yield`，所以咱们得把代码写成活的才行

```
function generatorToAsync(generatorFn) {
  return function() {
    const gen = generatorFn.apply(this, arguments) // gen有可能传参

    // 返回一个Promise
    return new Promise((resolve, reject) => {

      function go(key, arg) {
        let res
```

js 复制代码



```

    } catch (error) {
      return reject(error) // 报错的话会走catch, 直接reject
    }

    // 解构获得value和done
    const { value, done } = res
    if (done) {
      // 如果done为true, 说明走完了, 进行resolve(value)
      return resolve(value)
    } else {
      // 如果done为false, 说明没走完, 还得继续走

      // value有可能是: 常量, Promise, Promise有可能是成功或者失败
      return Promise.resolve(value).then(val => go('next', val), err => go('throw', err))
    }
  }

  go("next") // 第一次执行
})
}

const asyncFn = generatorToAsync(gen)

asyncFn().then(res => console.log(res))

```

这样的话, 无论是多少个yield都会排队执行了, 咱们把代码写成活的了



1k+



78



收藏

## async/await 版本

```
async function asyncFn() {
  const num1 = await fn(1)
  console.log(num1) // 2
  const num2 = await fn(num1)
  console.log(num2) // 4
  const num3 = await fn(num2)
  console.log(num3) // 8
  return num3
}

const asyncRes = asyncFn()
console.log(asyncRes) // Promise
asyncRes.then(res => console.log(res)) // 8
```

js 复制代码

## 使用 generatorToAsync函数 的版本

```
function* gen() {
  const num1 = yield fn(1)
  console.log(num1) // 2
  const num2 = yield fn(num1)
  console.log(num2) // 4
  const num3 = yield fn(num2)
  console.log(num3) // 8
  return num3
}
```

js 复制代码



1k+



78



收藏

```
console.log(asyncRes) // Promise
asyncRes.then(res => console.log(res)) // 8
```

## 结语

如果你觉得此文对你有一丁点帮助，点个赞，鼓励一下林三心哈哈。

如果你想一起学习前端或者摸鱼，那你可以加我，加入我的摸鱼学习群，[点击这里](#) ---> [摸鱼沸点](#)

如果你是有其他目的的，别加我，我不想跟你交朋友，我只想简简单单学习前端，不想搞一些有的没的!!!

分类： [前端](#) 标签： [前端](#) [JavaScript](#) [面试](#)

文章被收录于专栏：



面试——百毒不侵

面试

[关注专栏](#)

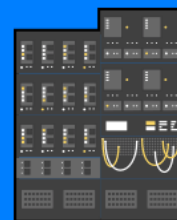
 1k+

 78

 收藏



找对——  
属于你的  
技术圈子



回复进群加入  
掘金  
微信交流群



## 评论



输入评论（Enter换行，Ctrl + Enter发送）

## 热门评论





作者一步一步讲的很清楚，最后实现的co模块（就是自动执行generator的库）👍📄

说下我的理解generator其实就是JS在语法层面对协程的支持，真正支持与否看运行时环境，比如高版本的node就是支持的。协程就是主程序和子协程直接控制权的切换，并伴随通信的过程，那么，从generator语法的角度来讲，yield，next就是通信接口，next是主协程向子协程通信，而yield就是子协程向主协程通信😁

👍 21    💬 3



**Sunshine\_Lin** Lv6 (作者)

6月前

牛，记下了

👍 点赞    💬 回复



**ChenyMartin**

6月前

老哥说的好，学习了👍

👍 点赞    💬 回复

查看更多回复 ▾



**越前君** Lv1 资深摸鱼工程师

6月前

对这句话不太认同：“await后面只有接了Promise才能实现排队效果”

await any可以理解成用 Promise.resolve(any) 处理的。那它本质上就变成了微任务，也就会有“排队”效果，即使 any 是一个原始值，待同步任务完成后再处理。

当然实际项目不会这么用，通常 await 都用于处理异步操作。

👍 8    💬 3

👍 1k+

💬 78

★ 收藏

有道理，谢谢指出

👍 点赞    💬 回复



ng\_kp Lv2    回复    Sunshine\_Lin

5月前

确实是，我觉得你可能想表达里面写了settimeout就没有排队效果，但这个实际原理是宏任务和微任务的执行问题，async/await本身还是会串行执行微任务的，就算不是微任务，也会转成微任务，这里改一下就更好啦

“有道理，谢谢指出”

👍 点赞    💬 回复

查看更多回复 ▾

## 全部评论 78

🔵 最新

🔥 最热



用户8025012919...

1天前

只有你讲人话,爱了

👍 点赞    💬 回复



用户7739847200...

4天前

for循环先执行获取里面的各个参数，然后再以for循环里面的参数为参数请求下一个方法，也能用async和await吗

👍 点赞    💬 回复

👍 1k+

💬 78

☆ 收藏

文章挺长时间了，但是我还是想交流一下我对于async/await的理解，async函数本身就是同步的，如果返回结果不是Promise类型，会将返回结果用Promise包装，如果是Promise对象不做处理。

在async函数执行的时候内部的generator手动执行一次yield，这样才会让Promise的状态变成fulfilled，Promise里面只要不是resolve/reject代码自身都会理解执行。...

[展开](#)

 点赞  回复

 东隅已逝 前端

11天前

“await只能在async函数中使用，否则会报错哦”

```
const a = await 1
```

```
console.log(a) //1
```

```
typeof(a) //'number'
```

所以await也可以不搭配await使用

 1  回复

 群演77 前端开发

2月前

又是看了心哥的一天，什么时候可以跟心哥一样优秀 🤔🤔🤔

 点赞  回复

 DongChuan\_12 Lv1 前端开发

2月前

 1k+

 78

 收藏

👍 点赞 💬 回复

前端小白菜\_ 前端

3月前

虽然我现在看不懂，但是终有一天我会看懂的

👍 点赞 💬 2

 Sunshine\_Lin Lv6 (作者)

3月前

看不懂就看十遍

👍 点赞 💬 回复

 前端小白菜\_ 回复 Sunshine\_Lin

3月前

好的，大佬

“看不懂就看十遍”


👍 点赞 💬 回复

 GreenTea Lv2 Web前端 @加菲猫

3月前

“await只能在aysnc函数中使用”这个可以更新一下，从Chrome 92开始已经支持 top-level await了，Webpack5也支持在模块顶层使用await

👍 1 💬 1

 Sunshine\_Lin Lv6 (作者)

3月前

这样啊

👍 1k+

💬 78

★ 收藏

雨越下越大 打工仔

4月前

图中的字体能不能换一个，看着头皮发麻

👍 点赞 💬 1

Sunshine\_Lin Lv6 (作者)

4月前

自带的

👍 点赞 💬 回复

\_Peach Lv1 前端

5月前

学到了

👍 点赞 💬 回复

pk\_啾 前端开发

5月前

总结里面的第二点：`async`函数返回的是一个状态为`fulfilled`的`Promise`对象`感觉不是很正确。

阮一峰的《ES6入门》中有谈及，`async`函数内部抛出错误，会导致返回的`Promise`对象变为`reject`状态。抛出的错误对象会被`catch`方法回调函数接收到。

应该改为`async`函数返回的是一个`Promise`对象`正确些。

👍 1 💬 1

Sunshine\_Lin Lv6 (作者)

5月前

👍 1k+

💬 78

☆ 收藏

👍 1    💬 回复

**yxne** Lv1 前端开发 @ 秘密

6月前

作者一步一步讲的很清楚，最后实现的co模块（就是自动执行generator的库）👍📄

说下我的理解generator其实就是JS在语法层面对协程的支持，真正支持与否看运行时环境，比如高版本的node就是支持的。协程就是主程序和子协程直接控制权的切换，并伴随通信的过程，那么，从generator语法的角度来讲，yield，next就是通信接口，next是主协程向子协程通信，而yield就是子协程向主协程通信 😊

👍 21    💬 3

**Sunshine\_Lin** Lv6 (作者)

6月前

牛，记下了

👍 点赞    💬 回复

**ChenyMartin**

6月前

老哥说的好，学习了 👍

👍 点赞    💬 回复

查看更多回复 ▾

**chasing**

6月前

刚开始的时候，都用promise了为什么还要嵌套呢

👍 2    💬 2

👍 1k+

💬 78

☆ 收藏



Hedgehog-03

4月前

因为不嵌套是同时出来的，试一下是这样的，但是我不理解

👍 点赞    💬 回复



Hedgehog-03    回复    Hedgehog-03

4月前

现在理解啦

“因为不嵌套是同时出来的，试一下是这样的，但是我不理解”

👍 点赞    💬 回复



Jeff在写代码 Lv1

6月前

大佬图用啥画的，真好看 ❤️

👍 点赞    💬 1



Sunshine\_Lin Lv6 (作者)

6月前

excalidraw

👍 1    💬 回复



芬达哥 Lv2    前端达人

6月前

很清晰的一次实现，不错哦

👍 1    💬 1

👍 1k+

💬 78

☆ 收藏





Sunshine\_Lin Lv6 (作者)

6月前

哈哈谢谢

点赞 回复



越前君 Lv1 资深摸鱼工程师

6月前

对这句话不太认同：“await后面只有接了Promise才能实现排队效果”

await any可以理解成用 Promise.resolve(any) 处理的。那它本质上就变成了微任务，也就会有“排队”效果，即使 any 是一个原始值，待同步任务完成后再处理。

当然实际项目不会这么用，通常 await 都用于处理异步操作。

8 3



Sunshine\_Lin Lv6 (作者)

6月前

有道理，谢谢指出

点赞 回复



ng\_kp Lv2 回复 Sunshine\_Lin

5月前

确实是，我觉得你可能想表达里面写了setTimeout就没有排队效果，但这个实际原理是宏任务和微任务的执行问题，async/await本身还是会串行执行微任务的，就算不是微任务，也会转成微任务，这里改一下就更好啦

“有道理，谢谢指出”

点赞 回复

查看更多回复

1k+

78

收藏

提伯斯 Lv1

6月前

async/await 是C#常用的一个语法糖,还有go里面的协程(go是自己调度这些微线程),这里拿火影忍者里的影分身举例,鸣人以极小的查克拉制作了个影分身,用来监听某个事件(例如一个非常耗时的网络请求),而自己的本尊则可以去做其它的事,从而提升资源的利用率

👍 1    💬 1

 **Sunshine\_Lin** Lv6 (作者)

6月前

对的，协程，上次听一个大佬讲过

👍 点赞    💬 回复

躺着吃肉都会胖 Lv2

6月前

我的心 是那光能使者变幻魔法～

👍 点赞    💬 回复

**Alone381** Lv3 前端 @ 广州某公司

6月前

小白路过。看完了，感觉文章只是描述了他们的执行顺序，没有说原理。

所以它们是怎样做到这样的异步请求的呢？

是使用了哪些数据结构来实现的异步等待和请求？

如果我们自己实现一遍类似async/await的函数，没有wait，没有promise，没有next等等，

我们要用怎样的算法来实现？

👍 点赞    💬 1

👍 1k+

💬 78

★ 收藏

根据这篇文章的说法，async/await 使用 generator 实现的。然而 generator 可以用 facebook 的 regenerator 实现，promise 可以自行实现。这个过程可以使用 babel / swc 在线 playground 看到，这可能是你关心的内容。

[github.com](#)


👍 1    💬 回复

来自拉夫德鲁的... Lv2 前端、全栈工程师 @ 99...

6月前

20分钟过去了，懂了但没完全懂 🐱

👍 3    💬 2

 **Sunshine\_Lin** Lv6 (作者)

6月前

多看一遍

👍 点赞    💬 回复

 **秋楓暮霞惋紅曲** Lv1

6月前

那你来实现原理，杠精

👍 点赞    💬 回复

查看全部 78 条回复 ▾

👍 1k+

💬 78

★ 收藏

快跑啊小卢\_ | 7天前 | 前端 · JavaScript · 面试

## 几个一看就会的实用JavaScript优雅小技巧✪

👁 1.6w    👍 252    💬 48

Sunshine\_Lin | 7月前 | 前端 · JavaScript · 面试

## 95%的人都回答不上来的问题：函数的length是多少？

👁 1.7w    👍 247    💬 131

Sunshine\_Lin | 1月前 | 前端 · JavaScript

## async/await 你是会用，但是你知道怎么处理错误吗？

👁 3.4w    👍 374    💬 140

前端胖头鱼 | 3月前 | 前端 · JavaScript · 面试

## 因为实现不了Promise.all，一场面试凉凉了

👁 4.4w    👍 603    💬 96

南玖 | 1月前 | JavaScript · 面试 · 前端

## 为什么大厂前端监控都在用GIF做埋点？

👁 3.8w    👍 544    💬 72

Sunshine\_Lin | 4月前 | 前端 · JavaScript · 面试

## 「干货」今年我写了55篇文章，面试了30个人，学习了385个知识点！

👍 1k+

💬 78

☆ 收藏

天行无忌 | 7月前 | Promise · JavaScript · 前端

## 从 async 和 await 函数返回值说原理

👁 1594    👍 7    💬 评论

天明夜尽 | 4月前 | 面试 · 前端 · JavaScript

## 4 年经验裸辞 2 个月，40 场面试、一路的心态变化及经验总结

👁 3.4w    👍 580    💬 86

Sunshine\_Lin | 5月前 | 前端 · JavaScript · 面试

## 史上最全！熬夜整理56个JavaScript高级的手写知识点！！专业扫盲！

👁 3.8w    👍 1157    💬 56

大Y | 3年前 | JavaScript · 前端 · Promise

## 一次性让你懂async/await，解决回调地狱

👁 3.6w    👍 803    💬 61

Sunshine\_Lin | 6月前 | 前端 · JavaScript · 面试

## 这可能是掘金讲「原型链」，讲的最好最通俗易懂的了，附练习题！

👁 1.4w    👍 357    💬 23

Sunshine\_Lin | 7月前 | 前端 · JavaScript · Vue.js

15张图，20分钟吃透ES6算法核心原理，我说的！

👍 1k+

💬 78

☆ 收藏

yeyan1996 | 2年前 | JavaScript · Webpack

## 嘿，不要给 async 函数写那么多 try/catch 了

👁 5.0w | 👍 1584 | 💬 151

zxcg\_神说要有光 | 6月前 | 前端 · JavaScript · Redux

## 为什么 redux-saga 不能用 async await 实现

👁 3122 | 👍 37 | 💬 4

浪里行舟 | 3年前 | JavaScript · 前端

## 九种跨域方式实现原理（完整版）

👁 13.6w | 👍 2613 | 💬 107

掘金安东尼 | 8月前 | JavaScript · 前端 · 面试

## 感谢 compose 函数，让我的代码屎山逐渐美丽了起来~

👁 5.1w | 👍 882 | 💬 167

Sunshine\_Lin | 4月前 | 前端 · JavaScript · 面试

## 看似简单的题，席卷几十个前端群，王红元老师都亲自出面解答

👁 3.3w | 👍 200 | 💬 92

doodlewind | 3月前 | 前端 · 年终总结 · JavaScript

这篇文章对你有帮助吗？

👍 1k+

💬 78

☆ 收藏

杰司 | 1年前 | JavaScript

## 当 `forEach` 遇上了 `async` 与 `await`

👁 4130    👍 33    💬 30

黄子毅 | 3年前 | React.js · 前端 · jQuery

## 精读《`async/await` 是把双刃剑》

👁 6050    👍 180    💬 17

👍 1k+

💬 78

☆ 收藏