

19 | 异步编程（二）：V8是如何实现async/await的？

2020-04-28 李兵

《图解 Google V8》

课程介绍 >



讲述：李兵

时长 15:04 大小 13.81M



你好，我是李兵。

上一节我们介绍了 JavaScript 是基于单线程设计的，最终造成了 JavaScript 中出现大量回调的场景。当 JavaScript 中有大量的异步操作时，会降低代码的可读性，其中最容易造成的就是回调地狱的问题。

JavaScript 社区探索并推出了一系列的方案，从“Promise 加 then”到“generator 加 co”方案，再到最近推出“终极”的 async/await 方案，完美地解决了回调地狱所造成的问题。

今天我们就来分析下回调地狱问题是如何被一步步解决的，在这个过程中，你也就理解了 V8 实现 async/await 的机制。



什么是回调地狱？


我们先来看什么是回调地狱。

假设你们老板给了你一个小需求，要求你从网络获取某个用户的用户名，获取用户名称的步骤是先通过一个 id_url 来获取用户 ID，然后再使用获取到的用户 ID 作为另外一个 name_url 的参数，以获取用户名。

我做了两个 DEMO URL，如下所示：

 复制代码

```
1 const id_url = 'https://raw.githubusercontent.com/binaryacademy/geektime-v8/maste
```

 复制代码

```
1 const name_url = 'https://raw.githubusercontent.com/binaryacademy/geektime-v8/mas
```

那么你会怎么实现这个小小的需求呢？

其中最容易想到的方案是使用 XMLHttpRequest，并按照前后顺序异步请求这两个 URL。具体地讲，你可以先定义一个 GetUrlContent 函数，这个函数负责封装 XMLHttpRequest 来下载 URL 文件内容，由于下载过程是异步执行的，所以需要通过回调函数来触发返回结果。那么我们需要给 GetUrlContent 传递一个回调函数 result_callback，来触发异步下载的结果。

最终实现的业务代码如下所示：

 复制代码

```
1 //result_callback: 下载结果的回调函数
2 //url: 需要获取URL的内容
3 function GetUrlContent(result_callback,url) {
4     let request = new XMLHttpRequest()
5
6     request.open('GET', url)
7
8     request.responseType = 'text'
9
10    request.onload = function () {
11
12        result_callback(request.response)
13
14    }
15
```



```

16 request.send()
17
18 }
19
20 function IDCallback(id) {
21
22     console.log(id)
23
24     let new_name_url = name_url + "?id="+id
25
26     GetUrlContent(NameCallback,new_name_url)
27
28 }
29
30 function NameCallback(name) {
31
32     console.log(name)
33
34 }
35
36 GetUrlContent(IDCallback,id_url)

```

在这段代码中：

- 我们先使用 GetUrlContent 函数来异步下载用户 ID，之后再通过 IDCallback 回调函数来获取到请求的 ID；
- 有了 ID 之后，我们再在 IDCallback 函数内部，使用获取到的 ID 和 name_url 合并成新的获取用户名称的 URL 地址；
- 然后，再次使用 GetUrlContent 来获取用户名称，返回的用户名称会触发 NameCallback 回调函数，我们可以在 NameCallback 函数内部处理最终的返回结果。

可以看到，我们每次请求网络内容，都需要设置一个回调函数，用来返回异步请求的结果，这些穿插在代码之间的回调函数打乱了代码原有的顺序，比如正常的代码顺序是先获取 ID，再获取用户名。但是由于使用了异步回调函数，获取用户名代码的位置，反而在获取用户 ID 的代码之上了，这就直接导致了我们的代码逻辑的不连贯、非线性，非常不符合人的直觉。

因此，异步回调模式影响到我们的编码方式，如果在代码中过多地使用异步回调函数，会将你的整个代码逻辑打乱，从而让代码变得难以理解，这也就是我们经常所说的**回调地狱**问题。



使用 Promise 解决回调地狱问题

为了解决回调地狱的问题，JavaScript 做了大量探索，最开始引入了 Promise 来解决部分回调地狱的问题，比如最新的 fetch 就使用 Promise 的技术，我们可以使用 fetch 来改造上面这段代码，改造后的代码如下所示：

 复制代码

```
1  fetch(id_url)
2
3  .then((response) => {
4
5    return response.text()
6
7  })
8
9  .then((response) => {
10
11    let new_name_url = name_url + "?id=" + response
12
13    return fetch(new_name_url)
14
15  }).then((response) => {
16
17    return response.text()
18
19  }).then((response) => {
20
21    console.log(response) // 输出最终的结果
22
23  })
```

我们可以看到，改造后的代码是先获取用户 ID，等到返回了结果之后，再利用用户 ID 生成新的获取用户名称的 URL，然后再获取用户名，最终返回用户名。使用 Promise，我们就可以按照线性的思路来编写代码，非常符合人的直觉。所以说，使用 Promise 可以解决回调地狱中编码非线性性的问题。

使用 Generator 函数实现更加线性化逻辑

虽然使用 Promise 可以解决回调地狱中编码非线性性的问题，但这种方式充满了 Promise 的 then() 方法，如果处理流程比较复杂的话，那么整段代码将充斥着大量的 then，异步逻辑之间依然被 then 方法打断了，因此这种方式的语义化不明显，代码不能很好地表示执行流程。



那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```

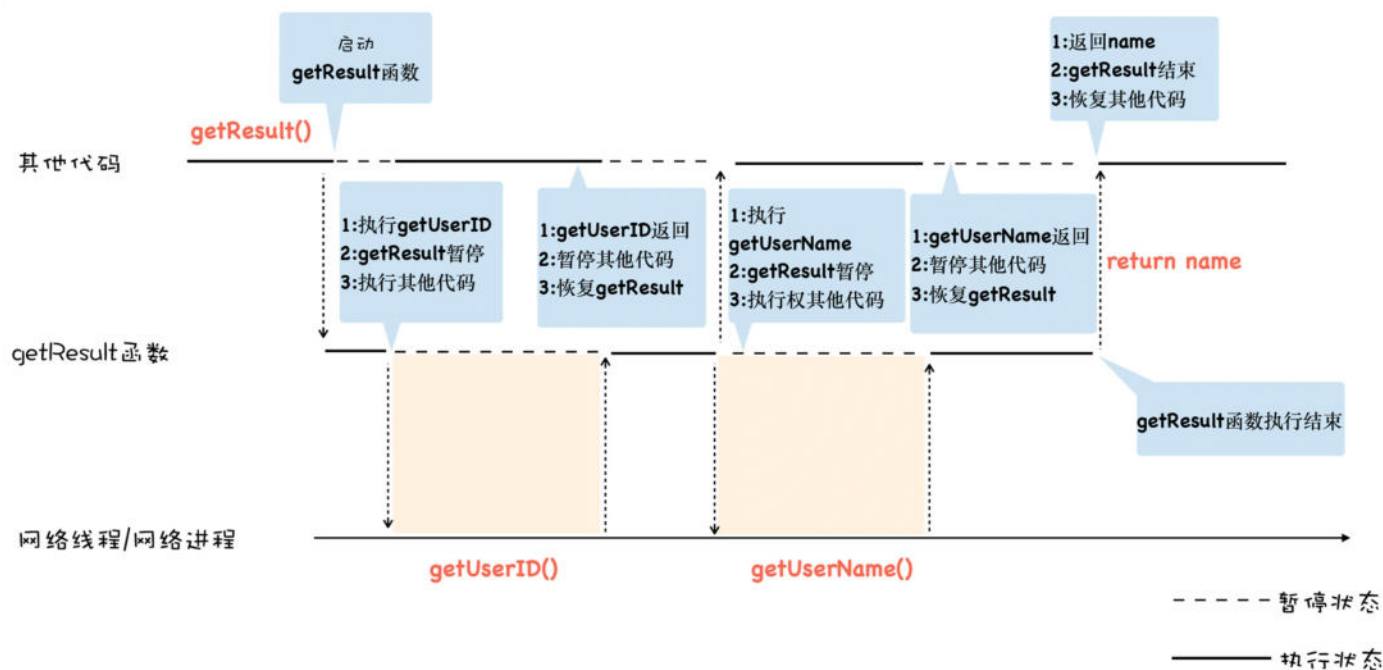
1  function getResult(){
2      let id = getUserID()
3      let name = getUserName(id)
4      return name
5  }

```

由于 `getUserID()` 和 `getUserName()` 都是异步请求，如果要实现这种线性的编码方式，那么一个可行的方案就是**执行到异步请求的时候，暂停当前函数，等异步请求返回了结果，再恢复该函数。**

具体地讲，执行到 `getUserID()` 时暂停 `getResult` 函数，然后浏览器在后台处理实际的请求过程，待 ID 数据返回时，再来恢复 `getResult` 函数。接下来再执行 `getUserName` 来获取到用户名，由于 `getUserName()` 也是一个异步请求，所以在使用 `getUserName()` 的同时，依然需要暂停 `getResult` 函数的执行，等到 `getUserName()` 返回了用户名数据，再恢复 `getResult` 函数的执行，最终 `getUserName()` 函数返回了 `name` 信息。


这个思维模型大致如下所示：



我们可以看出，这个模型的关键就是实现**函数暂停执行**和**函数恢复执行**，而生成器就是为了实现暂停函数和恢复函数而设计的。



生成器函数是一个带星号函数，配合 `yield` 就可以实现函数的暂停和恢复，我们看看生成器的具体使用方式：

 复制代码

```
1 function* getResult() {  
2  
3   yield 'getUserID'  
4  
5   yield 'getUserName'  
6  
7   return 'name'  
8  
9 }  
10  
11 let result = getResult()  
12  
13 console.log(result.next().value)  
14  
15 console.log(result.next().value)  
16 console.log(result.next().value)
```

执行上面这段代码，观察输出结果，你会发现函数 `getResult` 并不是一次执行完的，而是全局代码和 `getResult` 函数交替执行。

其实这就是生成器函数的特性，在生成器内部，如果遇到 `yield` 关键字，那么 V8 将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用 `result.next` 方法。

那么，V8 是怎么实现生成器函数的暂停执行和恢复执行的呢？

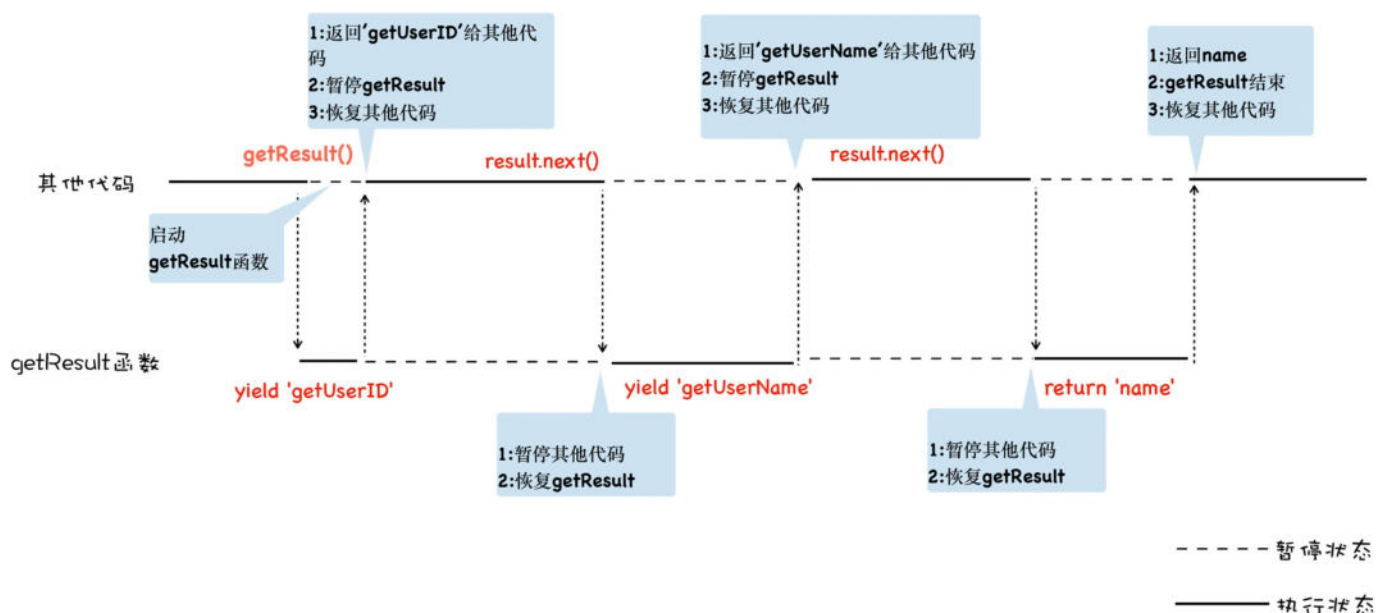
这背后的魔法就是**协程**，**协程是一种比线程更加轻量级的存在**。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程。比如，当前执行的是 A 协程，要启动 B 协程，那么 A 协程就需要将主线程的控制权交给 B 协程，这就体现在 A 协程暂停执行，B 协程恢复执行；同样，也可以从 B 协程中启动 A 协程。通常，**如果从 A 协程启动 B 协程，我们就把 A 协程称为 B 协程的父协程**。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。每一时刻，该线程只能执行其中某一个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制



(也就是在用户态执行)。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

为了让你更好地理解协程是怎么执行的，我结合上面那段代码的执行过程，画出了下面的“协程执行流程图”，你可以对照着代码来分析：



到这里，相信你已经弄清楚协程是怎么工作的了，其实在 JavaScript 中，生成器就是协程的一种实现方式，这样，你也就理解什么是生成器了。

因为生成器可以暂停函数的执行，所以，我们将所有异步调用的方式，写成同步调用的方式，比如我们使用生成器来实现上面的需求，代码如下所示：

复制代码

```
1 function* getResult() {
2   let id_res = yield fetch(id_url);
3   console.log(id_res)
4   let id_text = yield id_res.text();
5   console.log(id_text)
6
7
8   let new_name_url = name_url + "?id=" + id_text
9   console.log(new_name_url)
10
11
12   let name_res = yield fetch(new_name_url)
13   console.log(name_res)
14   let name_text = yield name_res.text()
15   console.log(name_text)
```



```

16 }
17
18
19 let result = getResult()
20 result.next().value.then((response) => {
21     return result.next(response).value
22 }).then((response) => {
23     return result.next(response).value
24 }).then((response) => {
25     return result.next(response).value
26 }).then((response) => {
27     return result.next(response).value

```

这样，我们可以将同步、异步逻辑全部写进生成器函数 `getResult` 的内部，然后，我们在外面依次使用一段代码来控制生成器的暂停和恢复执行。以上，就是协程和 `Promise` 相互配合执行的大致流程。

通常，我们把执行生成器的代码封装成一个函数，这个函数驱动了 `getResult` 函数继续往下执行，我们把这个执行生成器代码的函数称为**执行器**（可参考著名的 `co` 框架），如下面这种方式：

 复制代码

```

1 function* getResult() {
2     let id_res = yield fetch(id_url);
3     console.log(id_res)
4     let id_text = yield id_res.text();
5     console.log(id_text)
6
7
8     let new_name_url = name_url + "?id=" + id_text
9     console.log(new_name_url)
10
11
12     let name_res = yield fetch(new_name_url)
13     console.log(name_res)
14     let name_text = yield name_res.text()
15     console.log(name_text)
16 }
17 co(getResult())

```

async/await：异步编程的“终极”方案

由于生成器函数可以暂停，因此我们可以在生成器内部编写完整的异步逻辑代码，不过生成器依然需要使用额外的 `co` 函数来驱动生成器函数的执行，这一点非常不友好。



基于这个原因，ES7 引入了 `async/await`，这是 JavaScript 异步编程的一个重大改进，它改进了生成器的缺点，提供了在不阻塞主线程的情况下使用同步代码实现异步访问资源的能力。你可以参考下面这段使用 `async/await` 改造后的代码：

 复制代码

```
1  async function getResult() {
2      try {
3          let id_res = await fetch(id_url)
4          let id_text = await id_res.text()
5          console.log(id_text)
6
7          let new_name_url = name_url+"?id="+id_text
8          console.log(new_name_url)
9
10
11         let name_res = await fetch(new_name_url)
12         let name_text = await name_res.text()
13         console.log(name_text)
14     } catch (err) {
15         console.error(err)
16     }
17 }
18 getResult()
```

观察上面这段代码，你会发现整个异步处理的逻辑都是使用同步代码的方式来实现的，而且还支持 `try catch` 来捕获异常，这就是完全在写同步代码，所以非常符合人的线性思维。

虽然这种方式看起来像是同步代码，但是实际上它又是异步执行的，也就是说，在执行到 `await fetch` 的时候，整个函数会暂停等待 `fetch` 的执行结果，等到函数返回时，再恢复该函数，然后继续往下执行。

其实 `async/await` 技术背后的秘密就是 `Promise` 和生成器应用，往底层说，就是微任务和协程应用。要搞清楚 `async` 和 `await` 的工作原理，我们就得对 `async` 和 `await` 分开分析。

我们先来看看 `async` 到底是什么。根据 MDN 定义，`async` 是一个通过**异步执行并隐式返回 `Promise`** 作为结果的函数。



这里需要重点关注异步执行这个词，简单地理解，如果在 `async` 函数里面使用了 `await`，那么此时 `async` 函数就会暂停执行，并等待合适的时机来恢复执行，所以说 `async` 是一个异步执行的函数。

那么暂停之后，什么时机恢复 async 函数的执行呢？

要解释这个问题，我们先来看看，V8 是如何处理 await 后面的内容的。

通常，await 可以等待两种类型的表达式：

- 可以是任何普通表达式；
- 也可以是一个 Promise 对象的表达式。

如果 await 等待的是一个 Promise 对象，它就会暂停执行生成器函数，直到 Promise 对象的状态变成 resolve，才会恢复执行，然后得到 resolve 的值，作为 await 表达式的运算结果。

我们看下面这样一段代码：

 复制代码

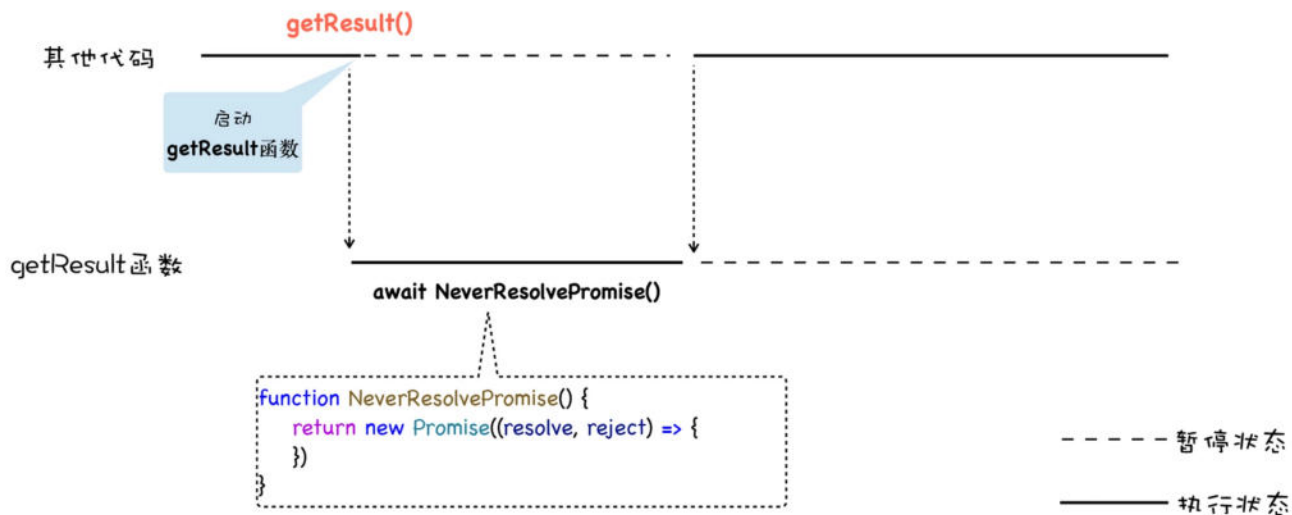
```
1 function NeverResolvePromise(){
2     return new Promise((resolve, reject) => {})
3 }
4 async function getResult() {
5     let a = await NeverResolvePromise()
6     console.log(a)
7 }
8 getResult()
9 console.log(0)
```

这一段代码，我们使用 await 等待一个没有 resolve 的 Promise，那么这也就意味着，getResult 函数会一直等待下去。

和生成器函数一样，使用了 async 声明的函数在执行时，也是一个单独的协程，我们可以使用 await 来暂停该协程，由于 await 等待的是一个 Promise 对象，我们可以 resolve 来恢复该协程。

下面是我从协程的视角，画的这段代码的执行流程图，你可以对照参考下：





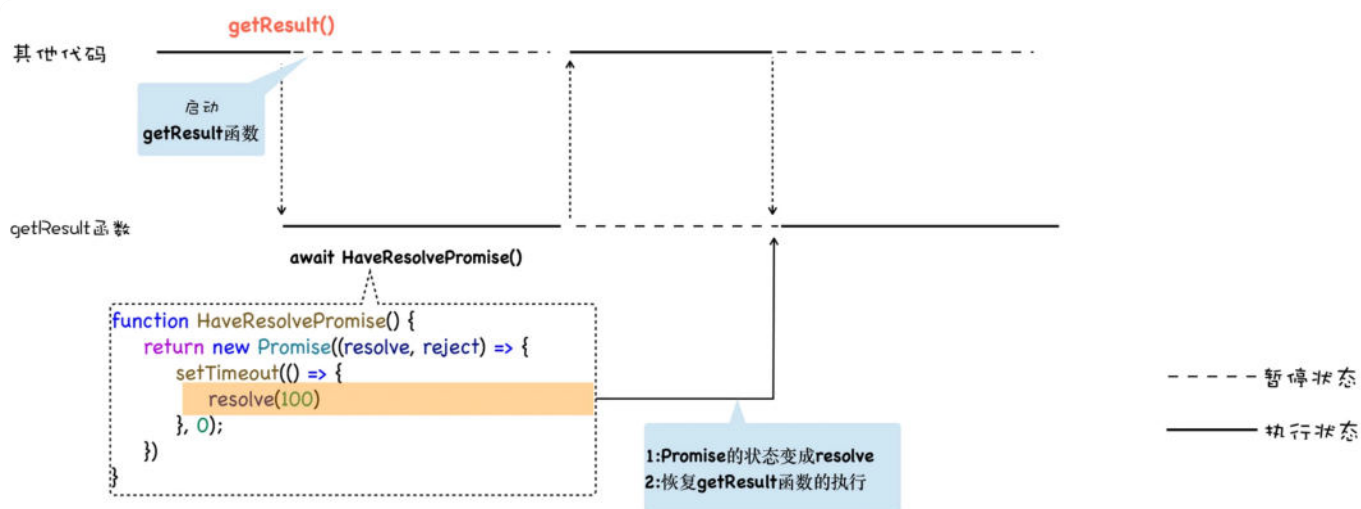
如果 await 等待的对象已经变成了 resolve 状态，那么 V8 就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

复制代码

```
1 function HaveResolvePromise(){
2   return new Promise((resolve, reject) => {
3     setTimeout(() => {
4       resolve(100)
5     }, 0);
6   })
7 }
8 async function getResult() {
9   console.log(1)
10  let a = await HaveResolvePromise()
11  console.log(a)
12  console.log(2)
13 }
14 console.log(0)
15 getResult()
16 console.log(3)
```

现在，这段代码的执行流程就非常清晰了，具体执行流程你可以参看下图：





如果 await 等待的是一个非 Promise 对象，比如 await 100，那么 V8 会隐式地将 await 后面的 100 包装成一个已经 resolve 的对象，其效果等价于下面这段代码：

复制代码

```
1 function ResolvePromise(){  
2   return new Promise((resolve, reject) => {  
3     resolve(100)  
4   })  
5 }  
6 async function getResult() {  
7   let a = await ResolvePromise()  
8   console.log(a)  
9 }  
10 getResult()  
11 console.log(3)
```

总结

Callback 模式的异步编程模型需要实现大量的回调函数，大量的回调函数会打乱代码的正常逻辑，使得代码变得非线性、不易阅读，这就是我们所说的回调地狱问题。

使用 Promise 能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

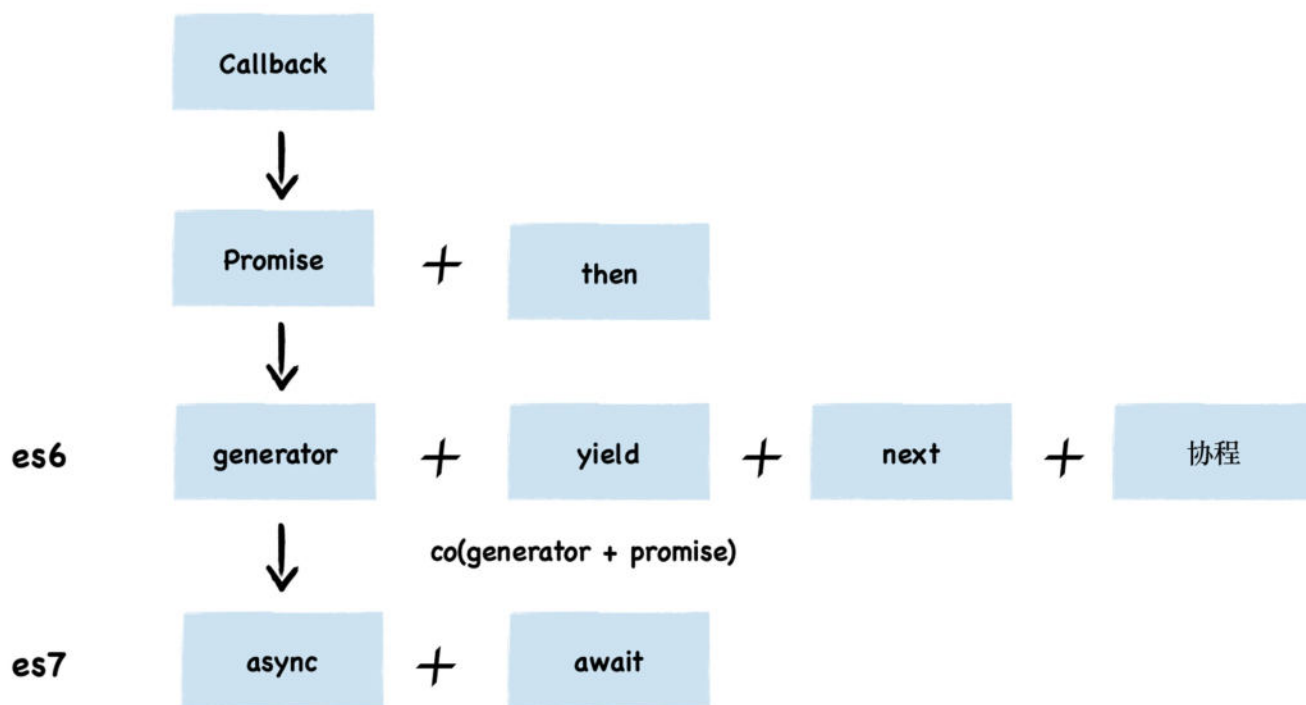
但是这种方式充满了 Promise 的 then() 方法，如果处理流程比较复杂的话，那么整段代码将充斥着大量的 then，语义化不明显，代码不能很好地表示执行流程。



我们想要通过线性的方式来编写异步代码，要实现这个理想，最关键的是要能实现函数暂停和恢复执行的功能。而生成器就可以实现函数暂停和恢复，我们可以在生成器中使用同步代码的逻辑来异步代码 (实现该逻辑的核心是协程)，但是在生成器之外，我们还需要一个触发器来驱动生成器的执行，因此这依然不是我们最终想要的方案。

我们的最终方案就是 `async/await`，`async` 是一个可以暂停和恢复执行的函数，我们会在 `async` 函数内部使用 `await` 来暂停 `async` 函数的执行，`await` 等待的是一个 `Promise` 对象，如果 `Promise` 的状态变成 `resolve` 或者 `reject`，那么 `async` 函数会恢复执行。因此，使用 `async/await` 可以实现以同步的方式编写异步代码这一目标。

你会发现，这节课我们讲的也是前端异步编程的方案史，我把这一过程也画了一张图供你参考：



思考题

了解 `async/await` 的演化过程，对于理解 `async/await` 至关重要，在进化过程中，`co+generator` 是比较优秀的一个设计。今天留给你的思考题是，`co` 的运行原理是什么？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。



分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 9 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 异步编程（一）：V8是如何实现微任务的？

下一篇 20 | 垃圾回收（一）：V8的两个垃圾回收器是如何工作的？

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (22)

写留言



若川

置顶

2020-05-02

co源码实现原理：其实就是通过不断的调用generator函数的next()函数，来达到自动执行generator函数的效果（类似async、await函数的自动自行）。

具体代码分析，我之前写过一篇文章：

《学习 koa 源码的整体架构，浅析koa洋葱模型原理和co原理》
<https://juejin.im/entry/5e6a080af265da575b1bd160>

作者回复: 赞，高手

共 4 条评论 >

👍 29



HoSalt

2020-04-29

老师 async、await 是 generator promise 的语法糖吗，v8里面前者是借助后者实现的吗？async await 为什么能用try catch 捕获错误？

作者回复: async/await可以不是语法糖，而是从设计到开发都是一套完整的体系，只不过使用了协程和promise！

支持try catch也是引擎的底层来实现的



👍 9



潇潇雨歇

2020-04-28

co的原理是自动识别生成器代码的yield，做暂停执行和恢复执行的操作

作者回复: 没问题



👍 7



Geek_gaoqin

2020-06-11

哦，我知道了，async 修饰的函数会有自己的协程，那么它代码内部创建的宏任务，主线程有空了还是会拿消息队列中的宏任务来执行，如果await等待了一个never resolve，那么它后面的代码就再也不会执行！但是却不会影响消息队列中键盘鼠标事件等其它任务的执行！

作者回复: 是这样的



👍 6



蹦哒

2020-05-12

请教老师：为什么Generator方案不实现自动执行next的功能呢？我理解async/await相对于Generator方案主要是能够自动执行next吧，co方案也是这么做的



作者回复: 如果自动执行了, 那么就是await了, 之所以没有这样实现, 我想是因为技术在迭代发展吧, 完美的技术总是很难一步到位

共 5 条评论 >

👍 4



Aaaaaaaayou

2020-04-28

co 里面一般会定义一个方法, 比如nextStep, 执行该方法时会调用迭代器的next, 根据结果中的done的取值来决定是继续递归调用nextStep还是结束。



👍 3



华仔

2020-05-05

```
setTimeout(() => {  
  console.log('in timeout');  
})  
new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve(3);  
    console.log('in promise-timeout')  
  })  
  console.log('in promise')  
}).then((res) => {  
  console.log('in then')  
})
```

老师, 想问下promise创建的then是微任务, 是宏任务中创建的队列保存的消息队列中维护的。那么我这里这样一个场景, 在promise中通过setTimeout(模拟宏任务http request)异步resolve的场景下, then也就会在下一个宏任务执行之后再执行了。这种当前宏任务中注册的微任务被拖到下一个宏任务执行, 是怎么实现的呢?

作者回复: 微任务是在resolve时生成的, 你创建一个promise并不会立马产生一个微任务, 而是要等到resolve或者reject时, 才会触发微任务, 在那个宏任务中触发了微任务, 微任务就在该宏任务快要执行结束之后执行, 无论你promise是在哪个宏任务中创建的!

共 4 条评论 >

👍 1



ruoyiran

2021-11-14

老师您好, 想请教一个问题, 在V8中是否有办法实现类似下面这样的代码, 或者有什么思路可以提供吗?



```
var signal = createSignal();

async function asyncFunction() {
  Promise.resolve().then(() => {
    console.info("resolved!!!");
    signal.notify();
  });
}

function run() {
  asyncFunction();
  console.info("waiting for Promise.resolve");
  signal.wait();
  console.info("continue to do sth.");
}

function main() {
  // code1....
  run();
  // code2...
  // 期望输出结果:
  // waiting for Promise.resolve
  // resolved!!!
  // continue to do sth.
}
```

学习了老师这节的异步编程，收获挺多的。最近在工作中因项目需要，想实现一个在非async函数（如上面的run函数）中，等待一个async函数执行完成。因为在某个地方看到别人的代码实现逻辑差不多就这样的，在项目中也希望能实现这样的逻辑，只是createSignal内部实现不知道是怎样的。想请教下。



Prof. Bramble

2021-08-28

我觉得有部分描述有点小问题，比如 generator 运行，其运行后返回就不是函数了，而且也不会直接执行函数体的内容



小云子

2021-03-28

还是想不通协程和消息队列、调用栈之间的执行流程是怎样的，有点难理解。



慢慢来的比较快

2021-03-17

协程在运行的时候，主线程可以从消息队列中获取事件执行吗？



吴少

2021-01-04

```
const id_url = 'https://raw.githubusercontent.com/binaryacademy/geektime-v8/master/id'
```

这两个链接如果不翻墙，访问不到啊，国内这网络环境.....



吴少

2021-01-03

v8里面竟然也有协程，我以为只有go才有



Geek_gaoqin

2020-09-13

老师，请问一下，`const a = await funA()`时，此时 `async funA`中，比如又有 `const b = await funB()`，`return`的数据如果不依赖于`funA`中`await`的结果时 它就会不等`funB`执行就`return`数据了，如果是`return b`(或者是返回结果对`b`有依赖)它就会等`funB`执行结束再返回。我感觉项目中这也是一个容易出bug的地方，为什么不是统一都按照`funB`执行结束后再`return`数据呢？直觉和主观意图应该是要等结果后再返回呀。

共 1 条评论 >



落风

2020-09-01

`co` 迭代了好多个版本，目前把支持的数据类型，都转换成 `Promise` 在流转；核心就是一个 `generator runner`，通过 `next(val)` 来不断执行 `generator`，理解 `next(val)` 用来获取和传参对于流程分析至关重要



Geek_gaoqin



2020-06-11

老师，很长很长一段代码中，业务逻辑很复杂，既有产生微任务，又有setTimeout产生宏任务，更有很多await的语句，那么这些结合上一章节讲的内容，它的执行顺序是怎样的呢？可以帮我分析下吗？

作者回复: 微任务先执行，setTimeout后执行，await可以跨越多个宏任务



Presbyter

2020-06-11

老师，我如果想在一個async內同時執行多個await操作，這個應該怎麼處理呢



断线人偶

2020-05-24

老師可以講一下為什麼在使用async...await...可以通過try...catch...來捕獲到異步函數中的異常嗎，v8是怎麼實現的

作者回复: 这个是V8實現的，比如觸發了reject的時候，v8就會跳轉到catch



地球外地人

2020-05-04

老師能讲讲 generate 和 await async中的閉包嗎？

作者回复: 閉包的實現原理都是一樣的，你可以列出具體問題，我來給你回答，最好新開一個回復，這樣我更容易看到



天然呆

2020-04-28

```
function HaveResolvePromise(){
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(100)
    }, 0);
  });
}
```



```
    })
  }
  async function getResult() {
    console.log(1)
    let a = await NoResolvePromise()
    console.log(a)
    console.log(2)
  }
  console.log(0)
  getResult()
  console.log(3)
```

是不是要改动? NoResolvePromise ==>> HaveResolvePromise

作者回复: 嗯 改过来了

