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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {

result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

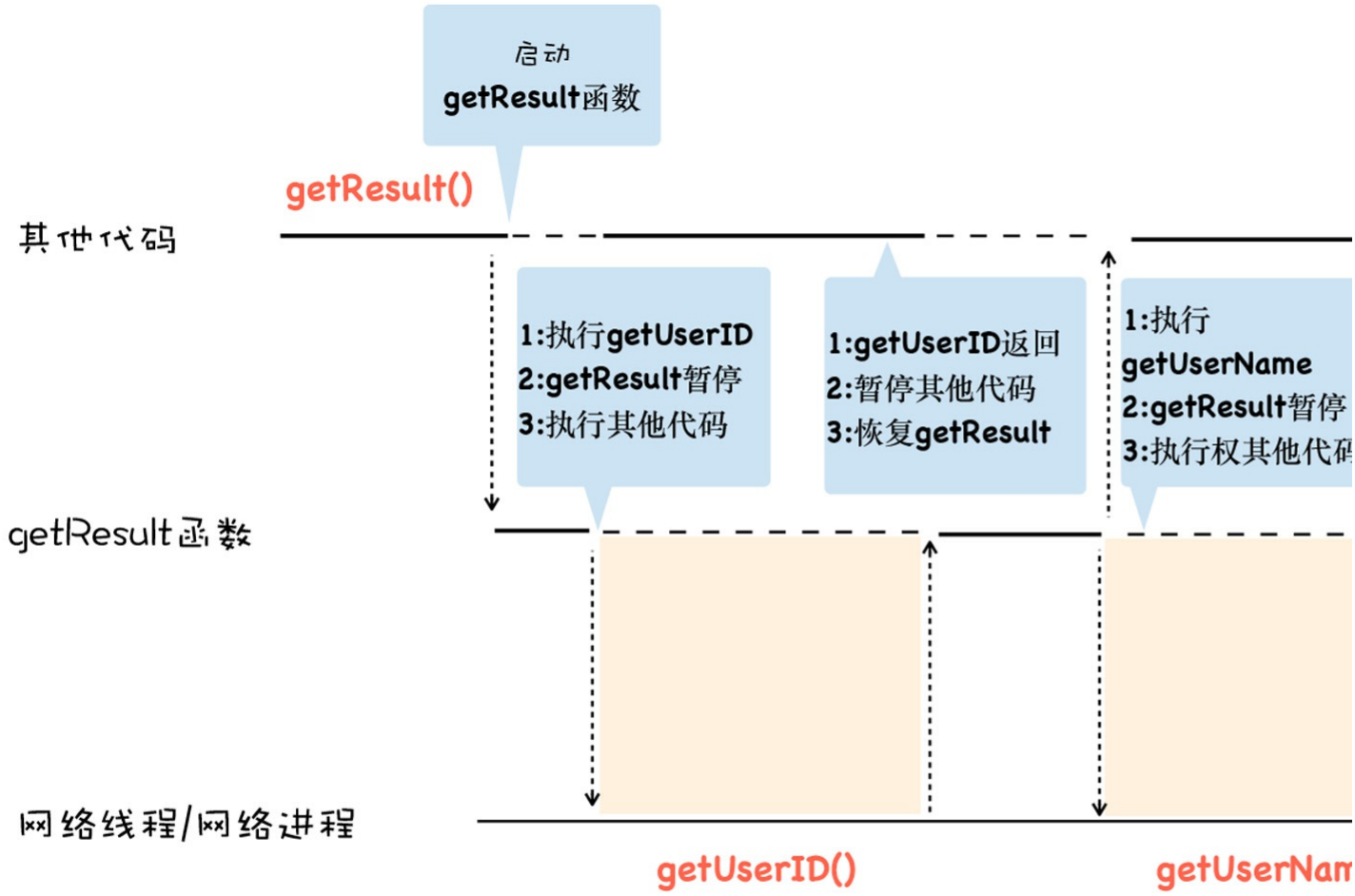
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
 function getResult(){
   let id = getUserID()
   let name = getUserName(id)
   return name
 }
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

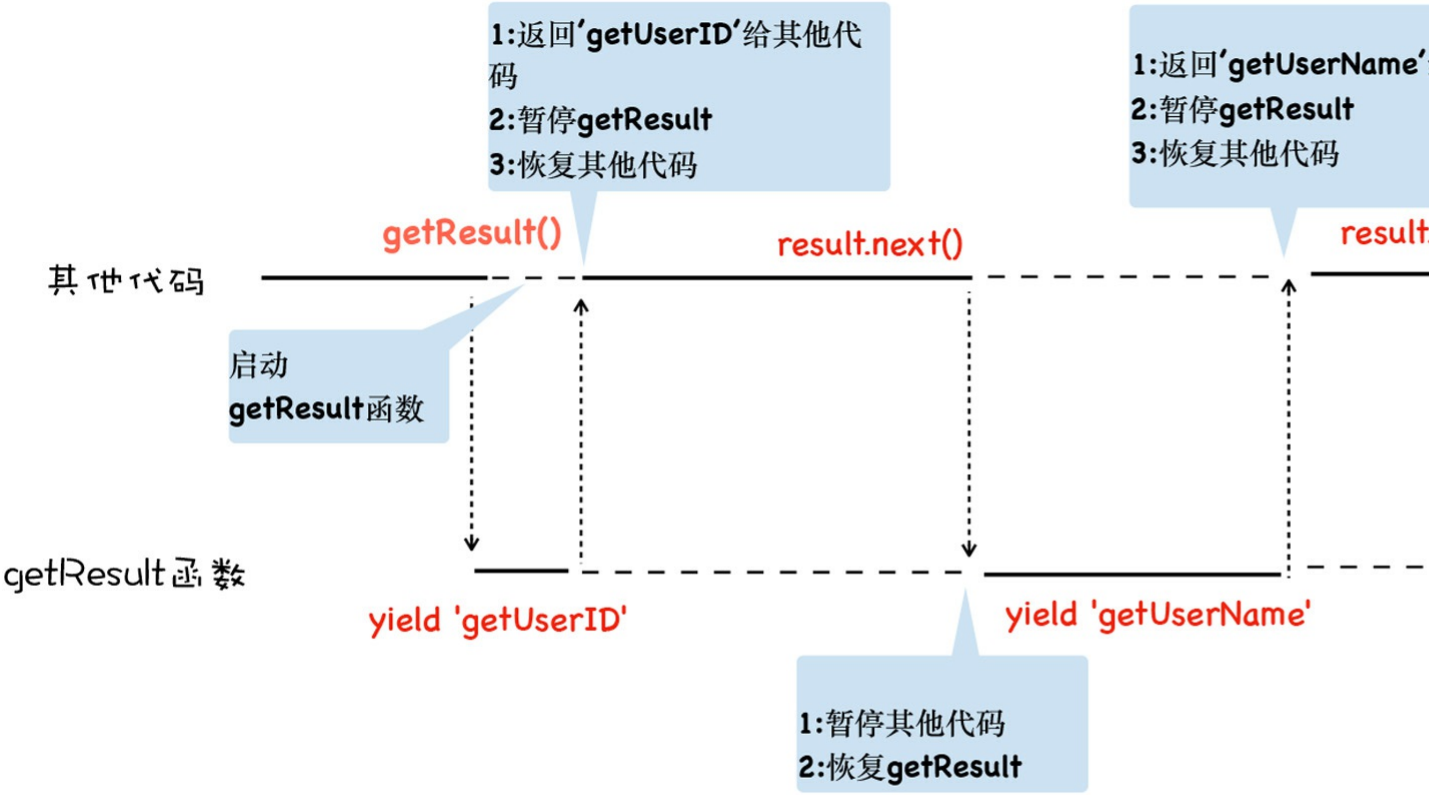
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

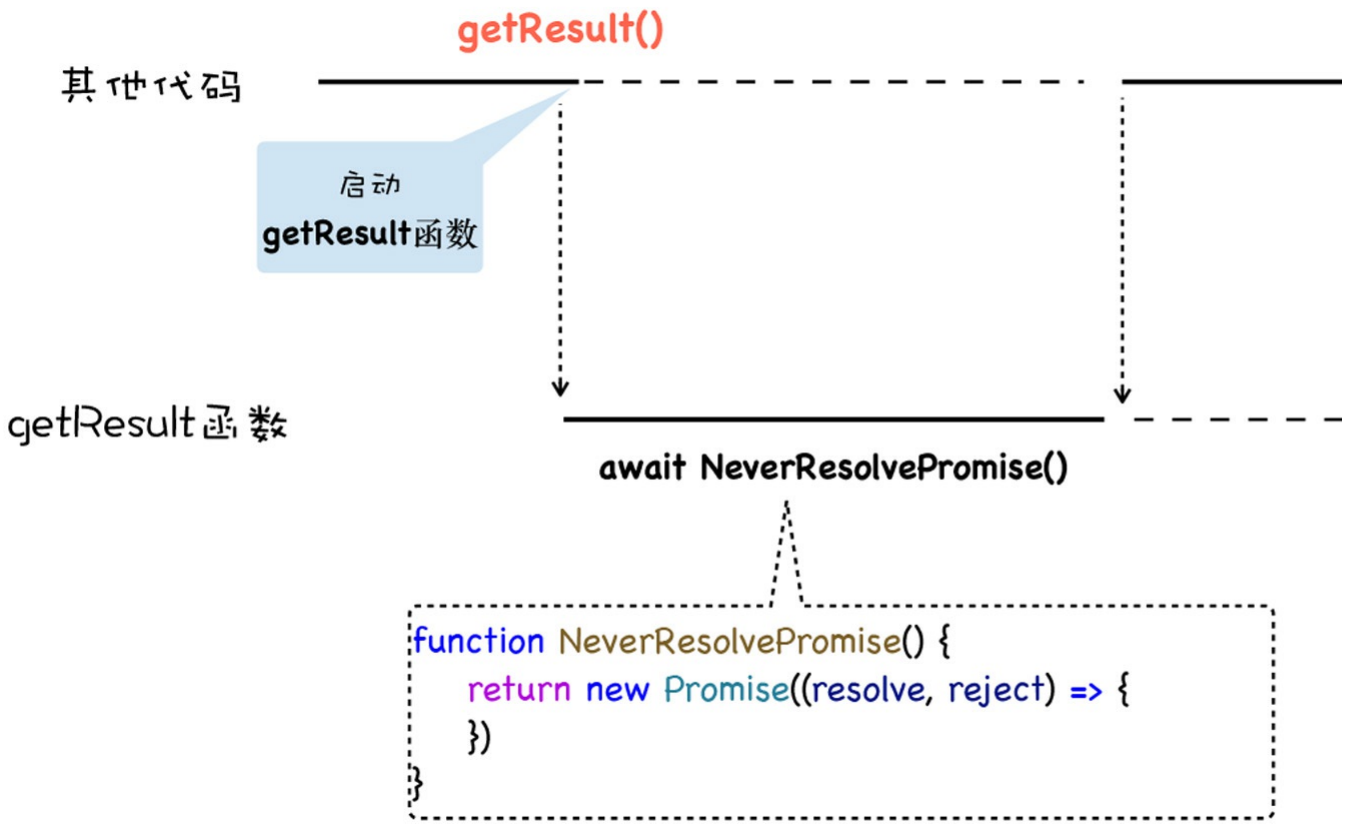
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

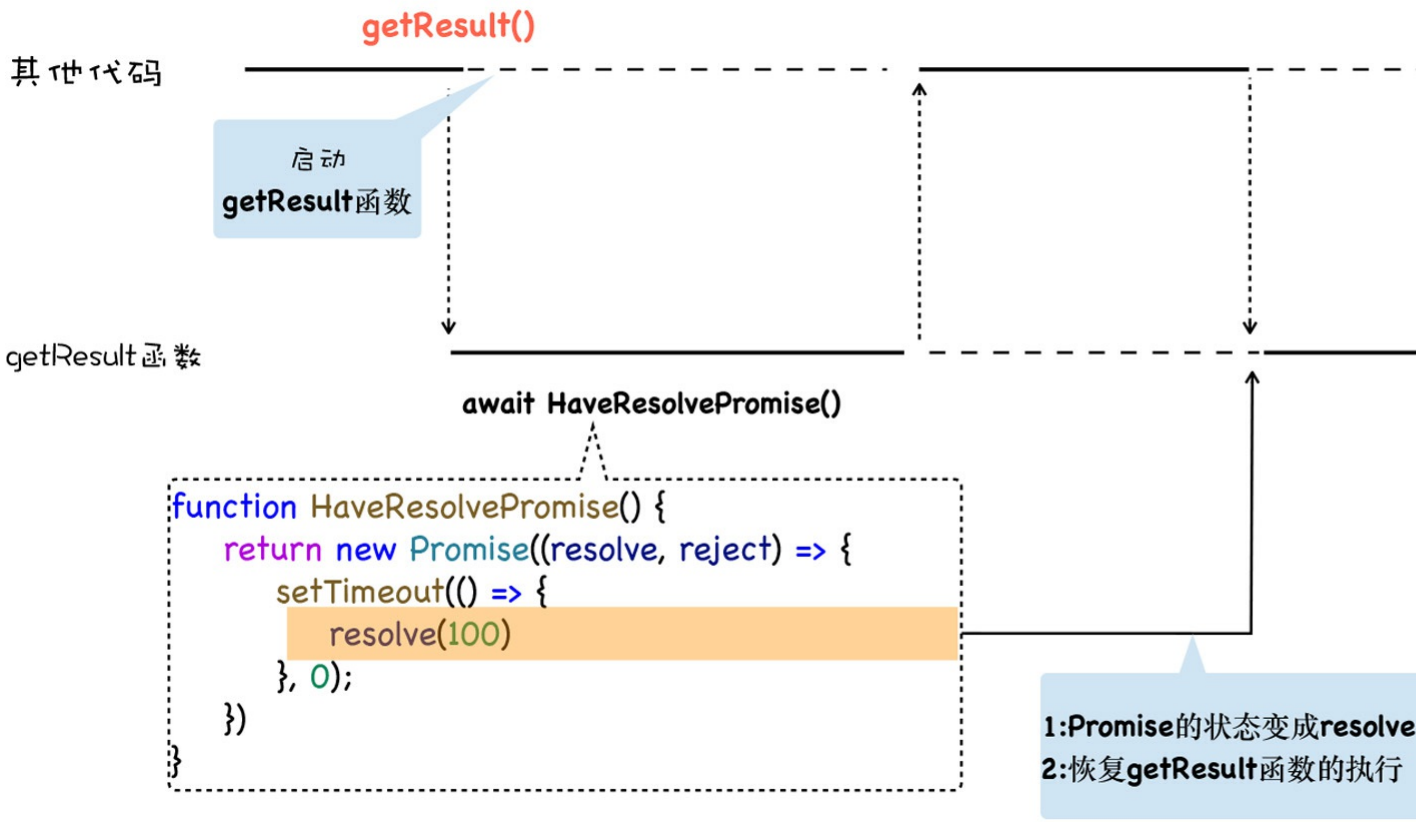


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

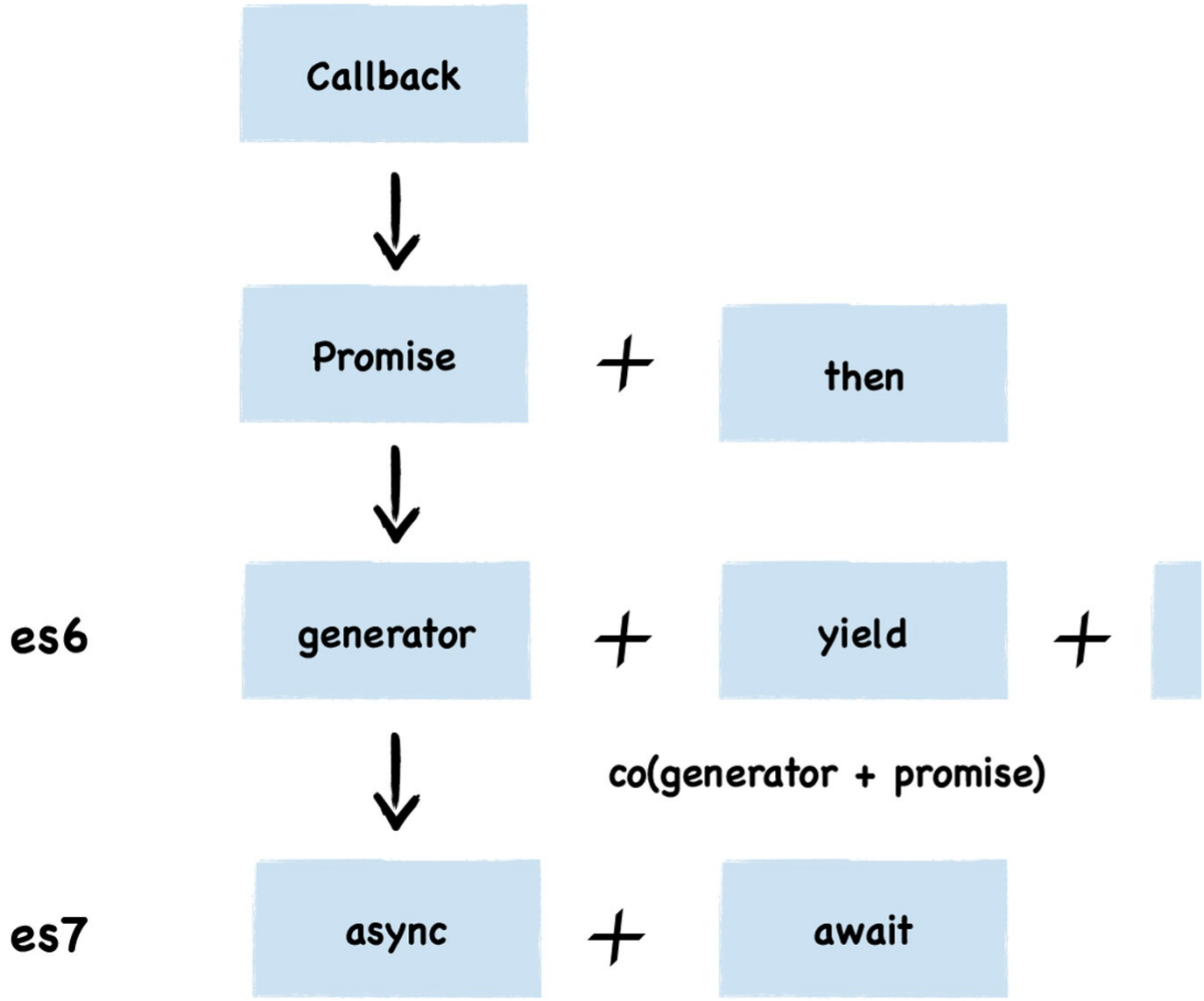
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

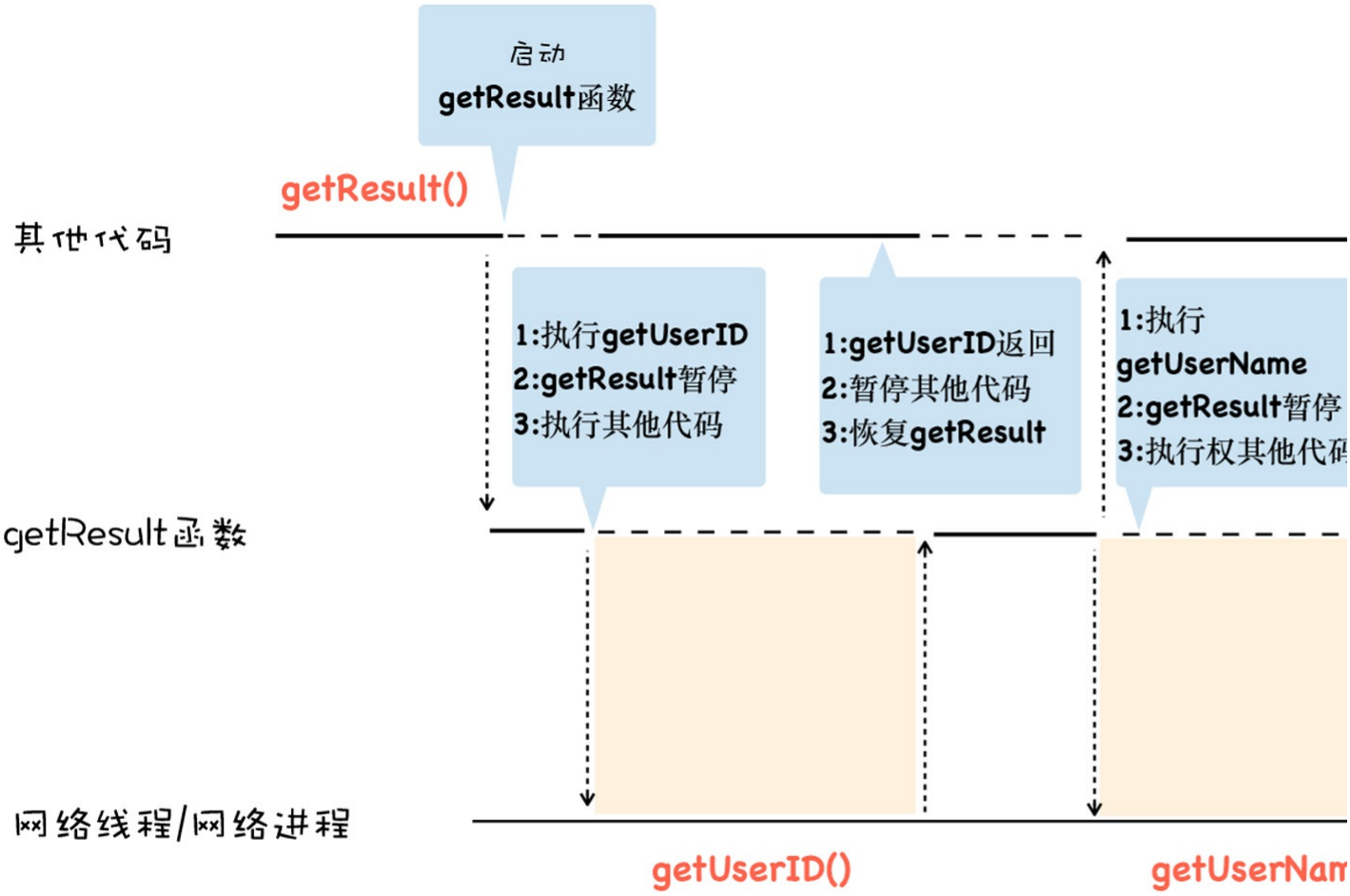
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

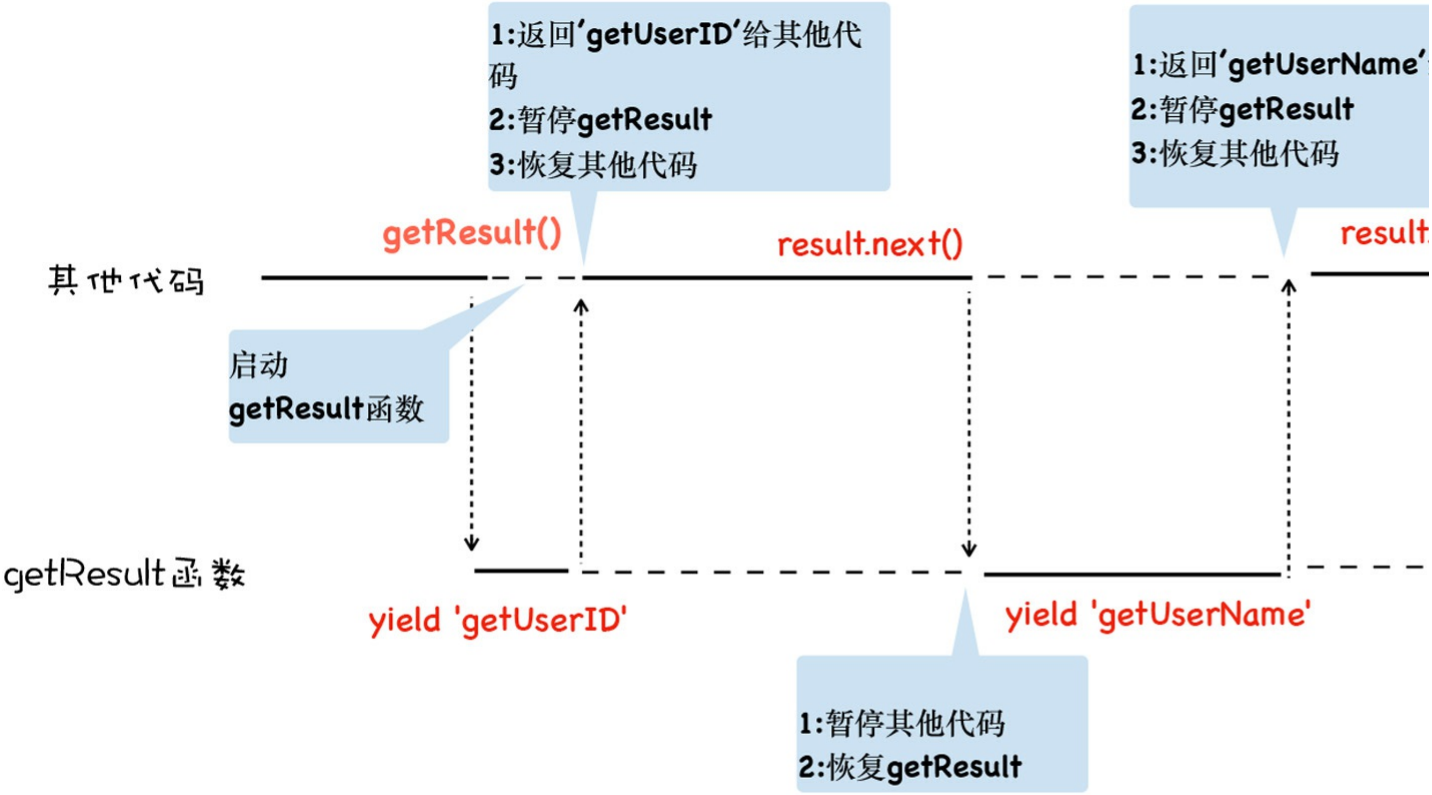
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

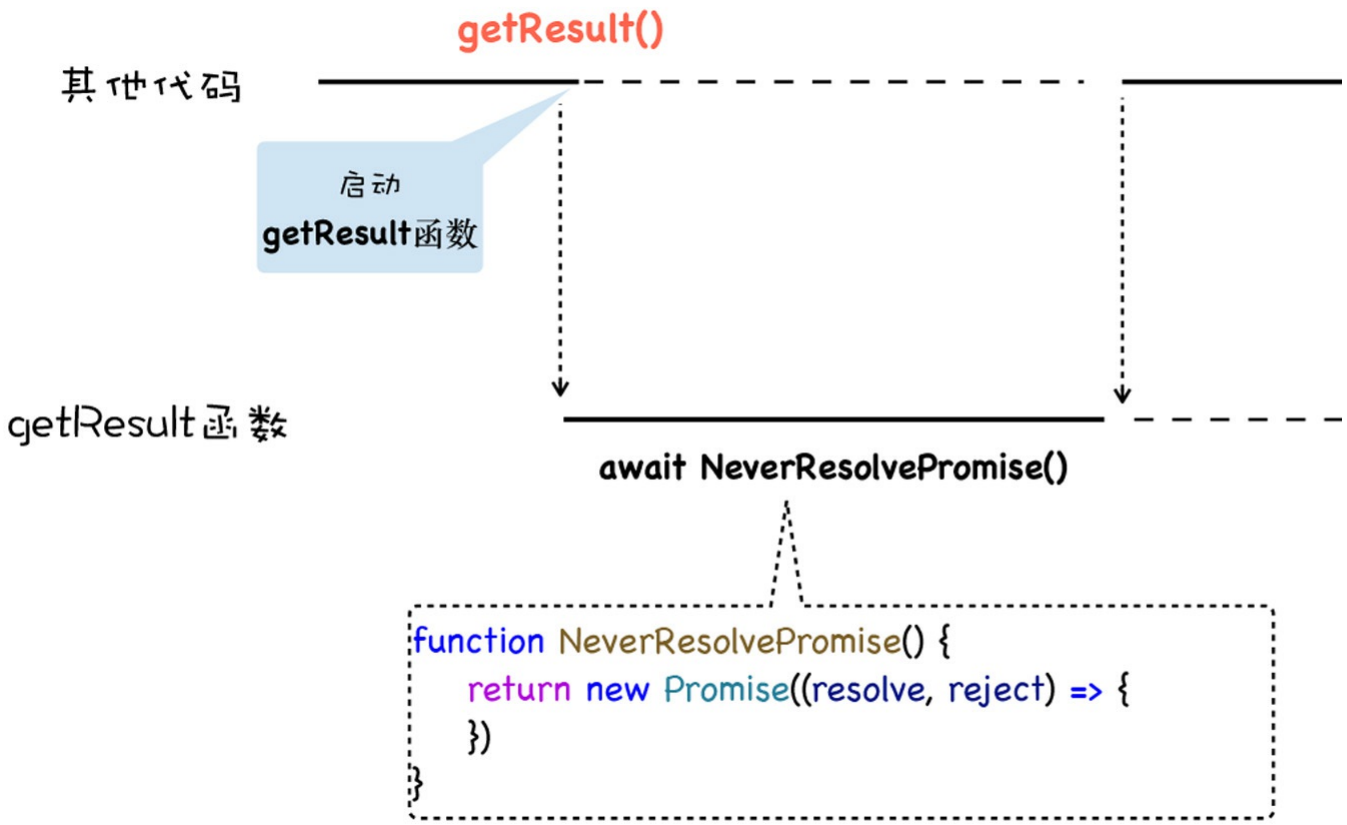
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

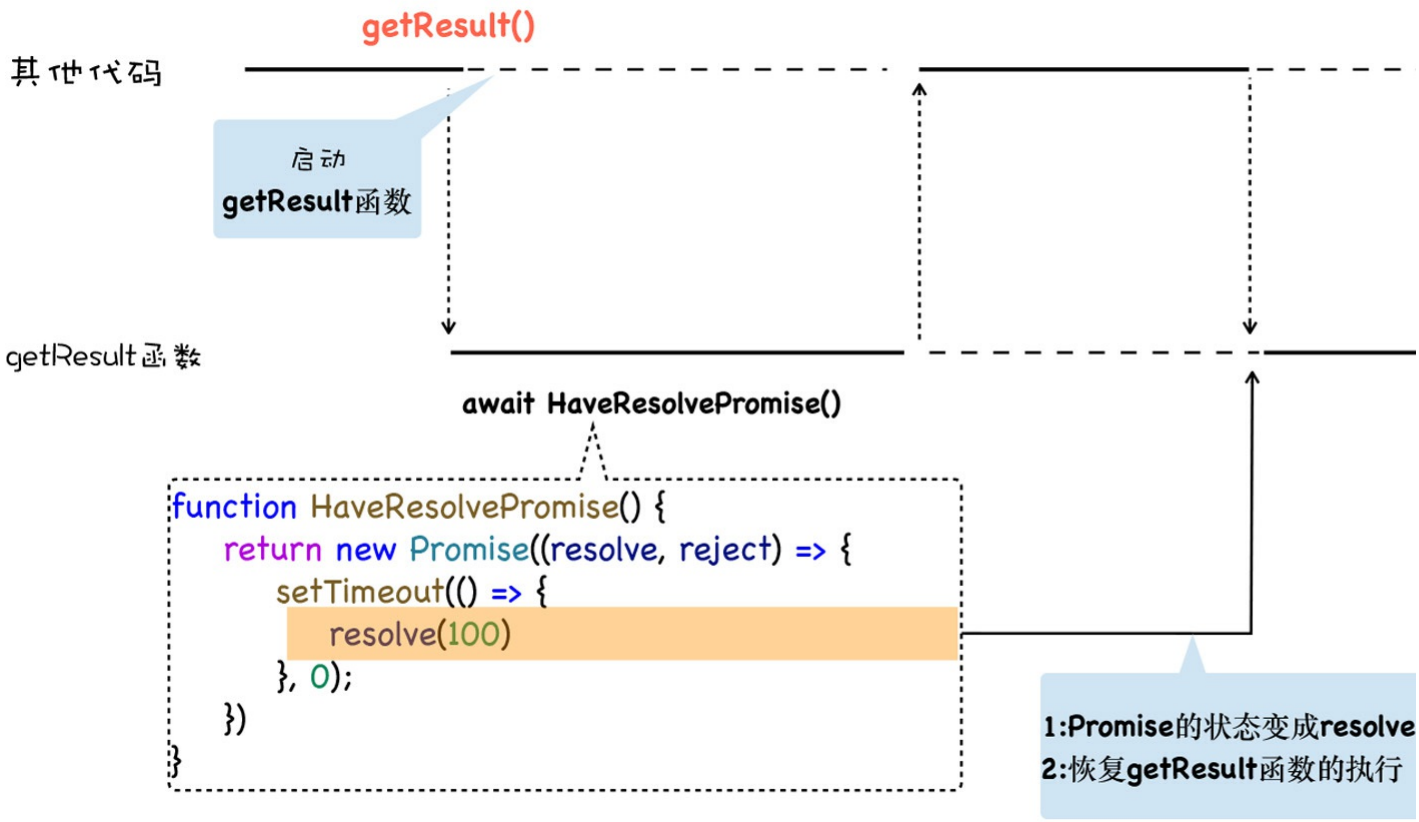


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

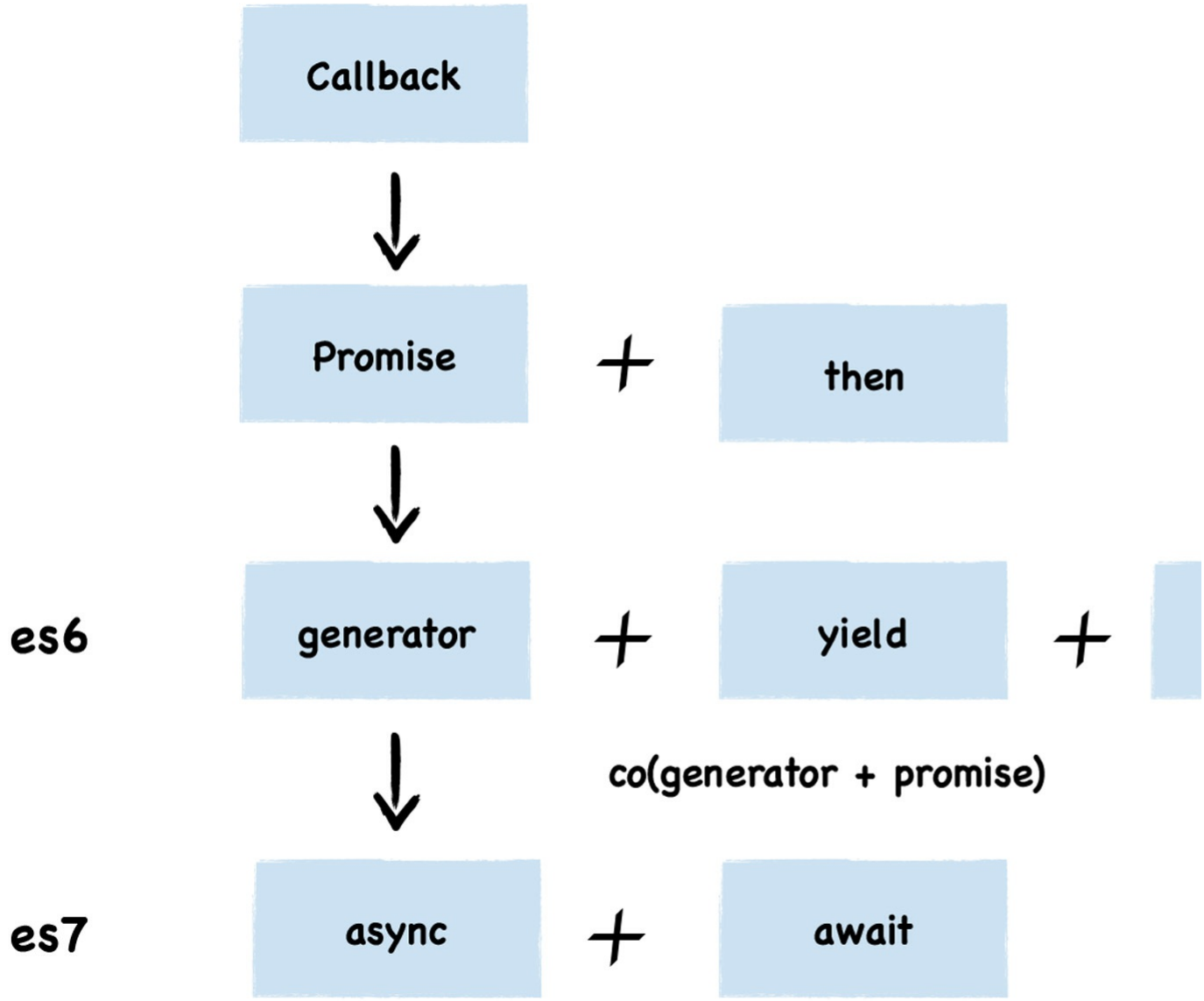
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

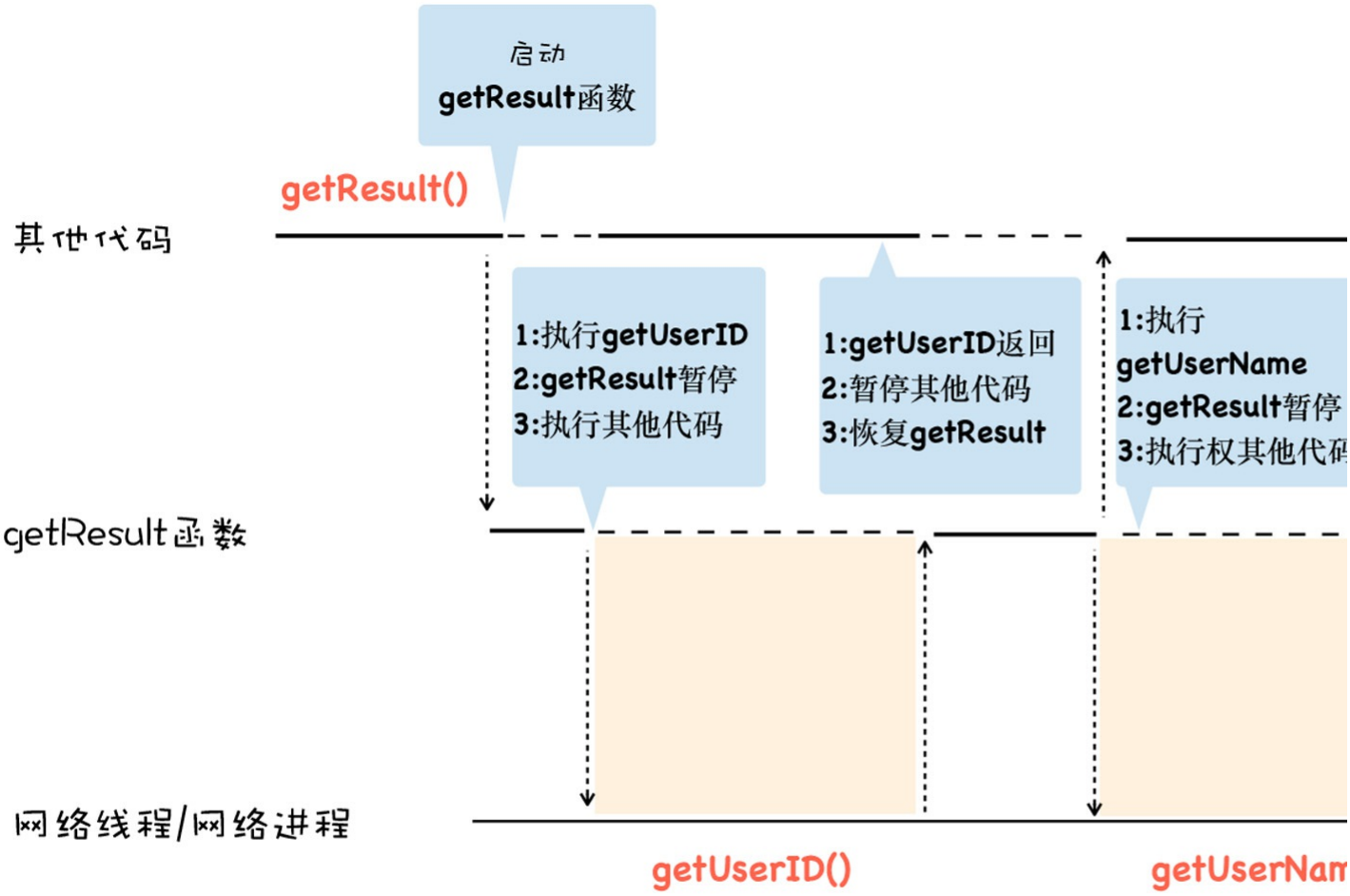
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

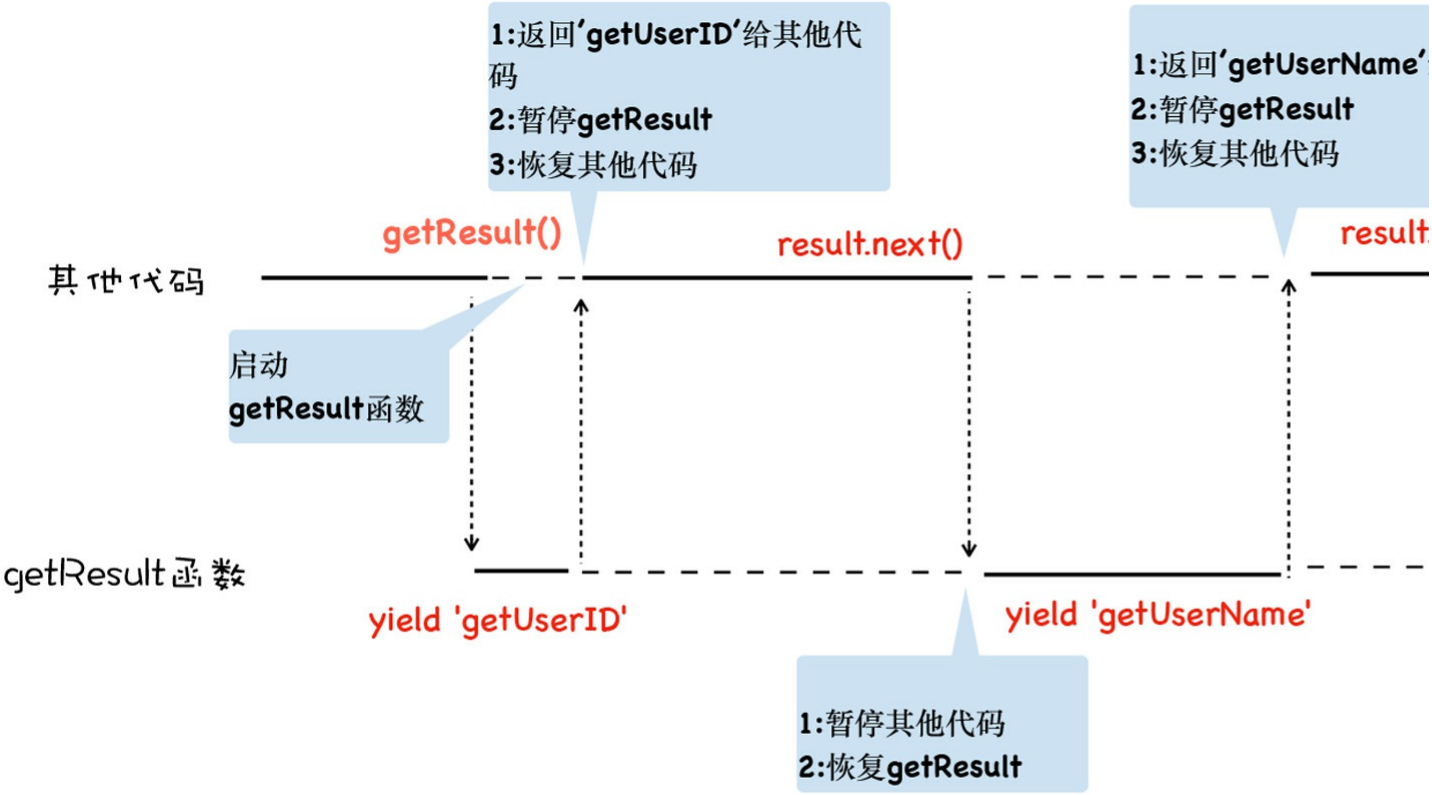
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

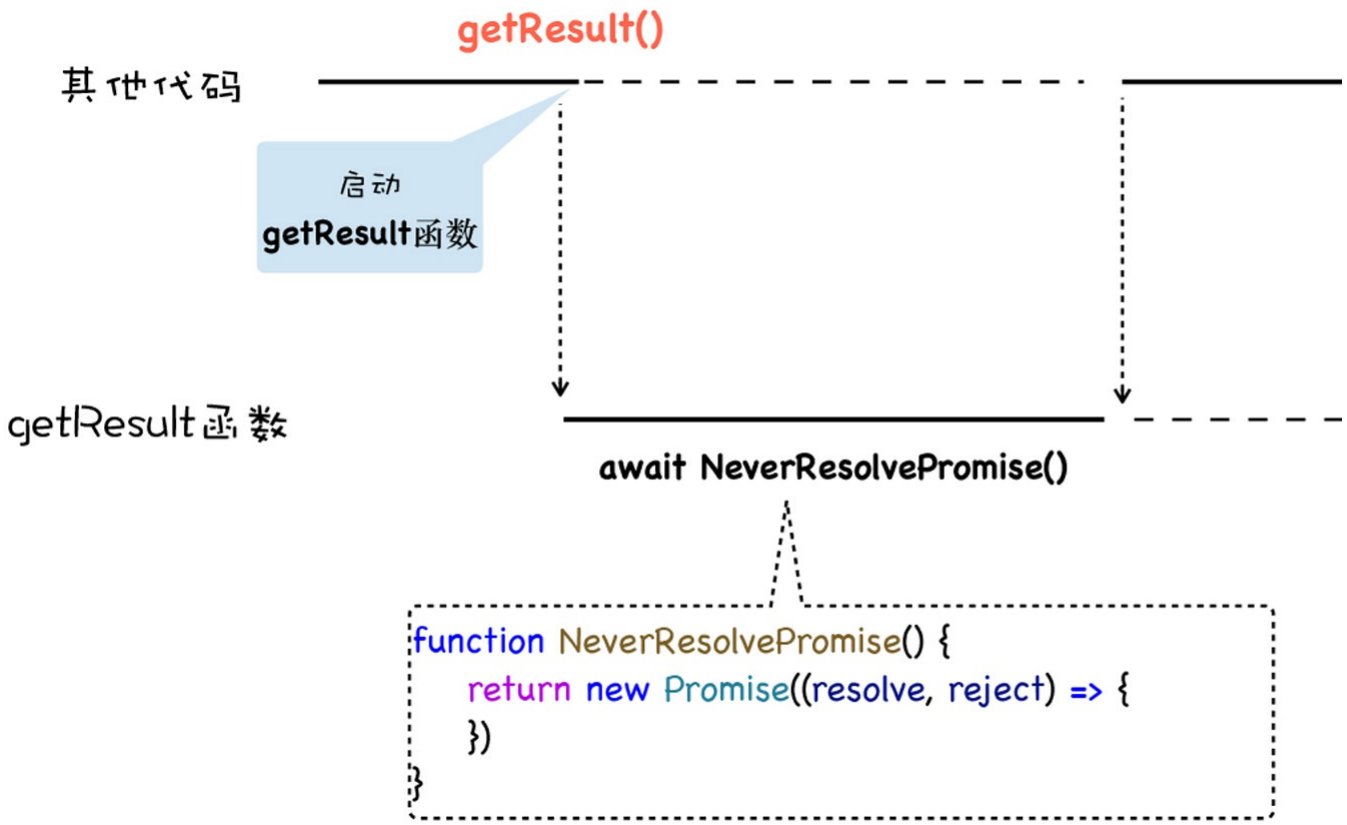
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

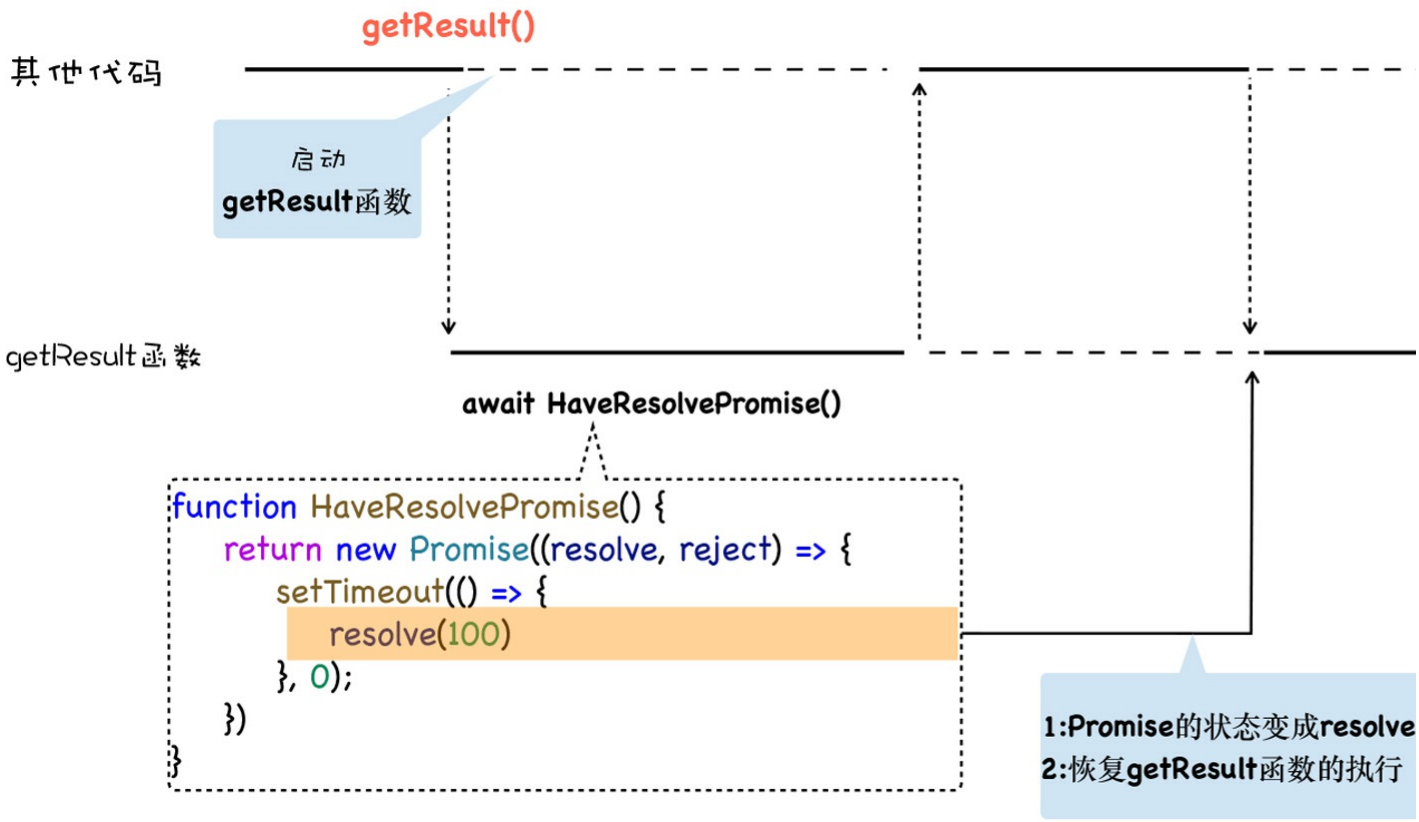


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

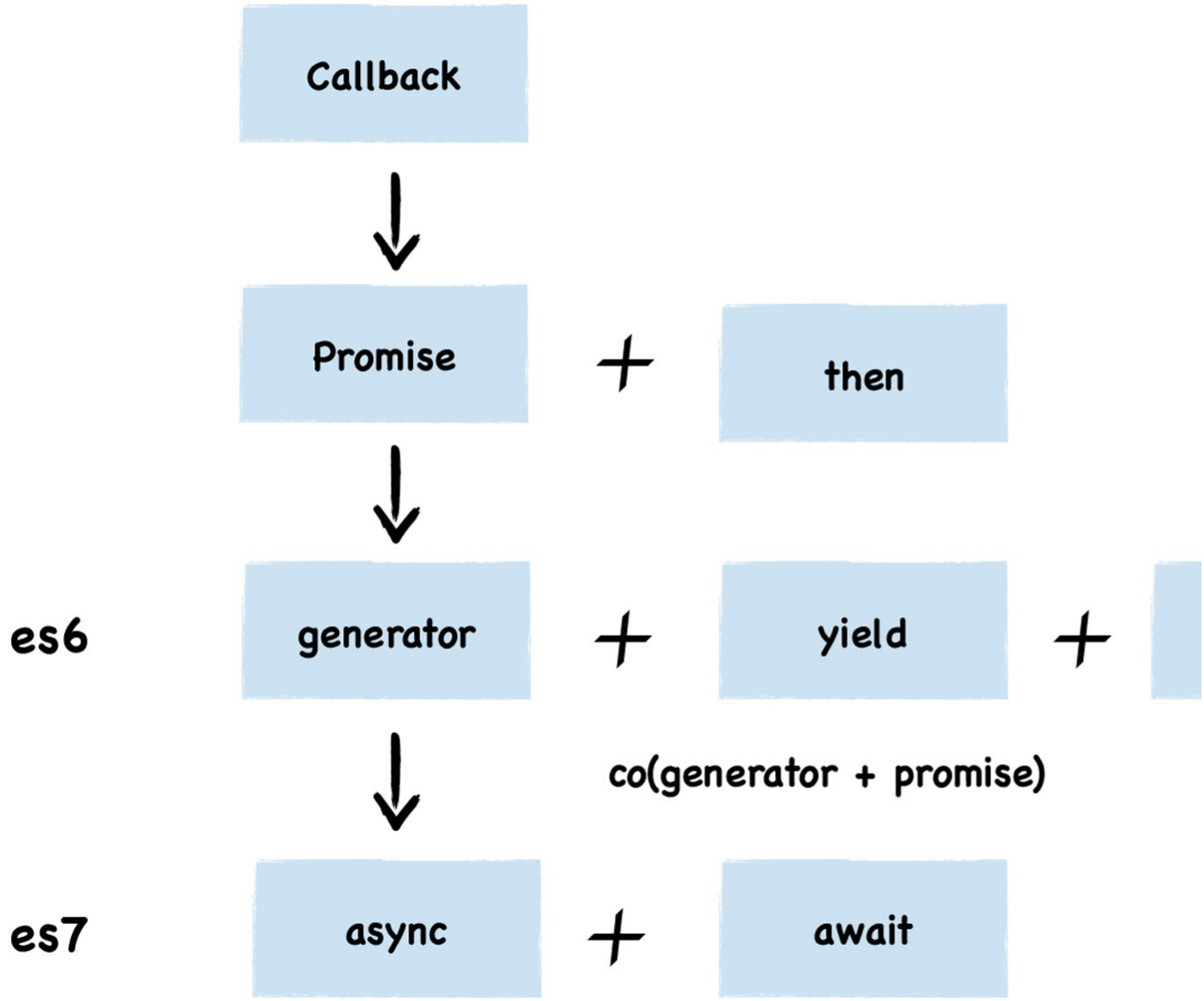
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

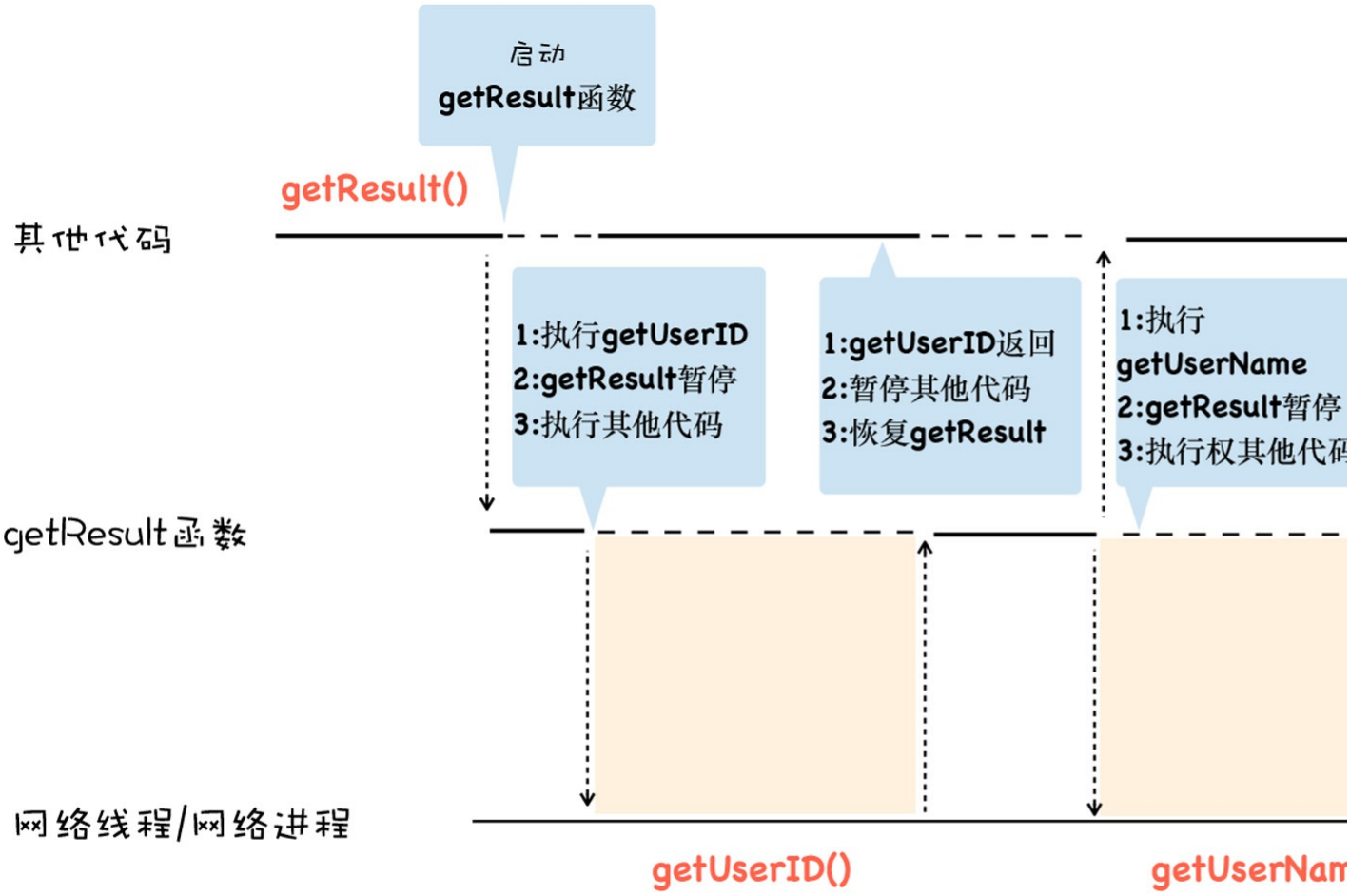
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

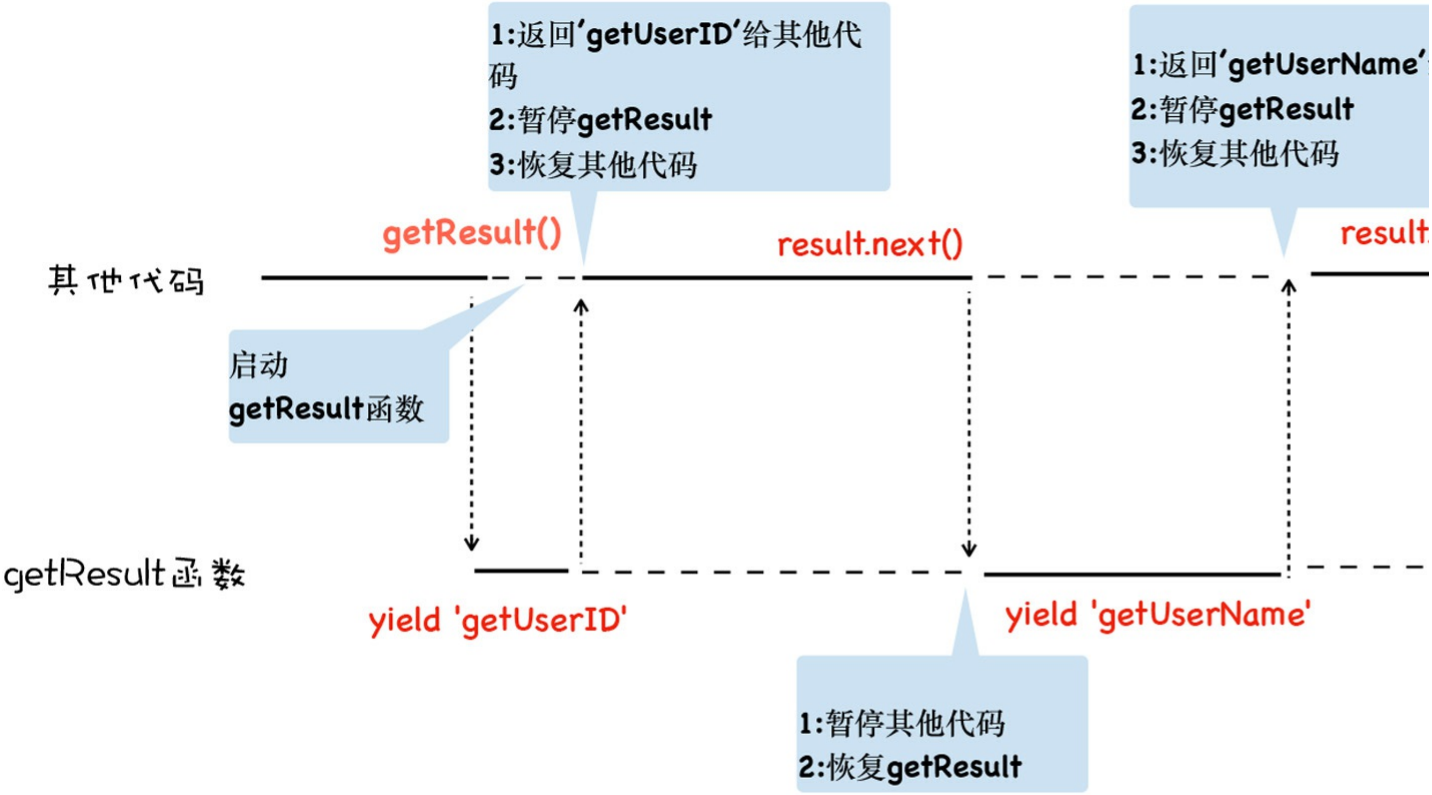
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

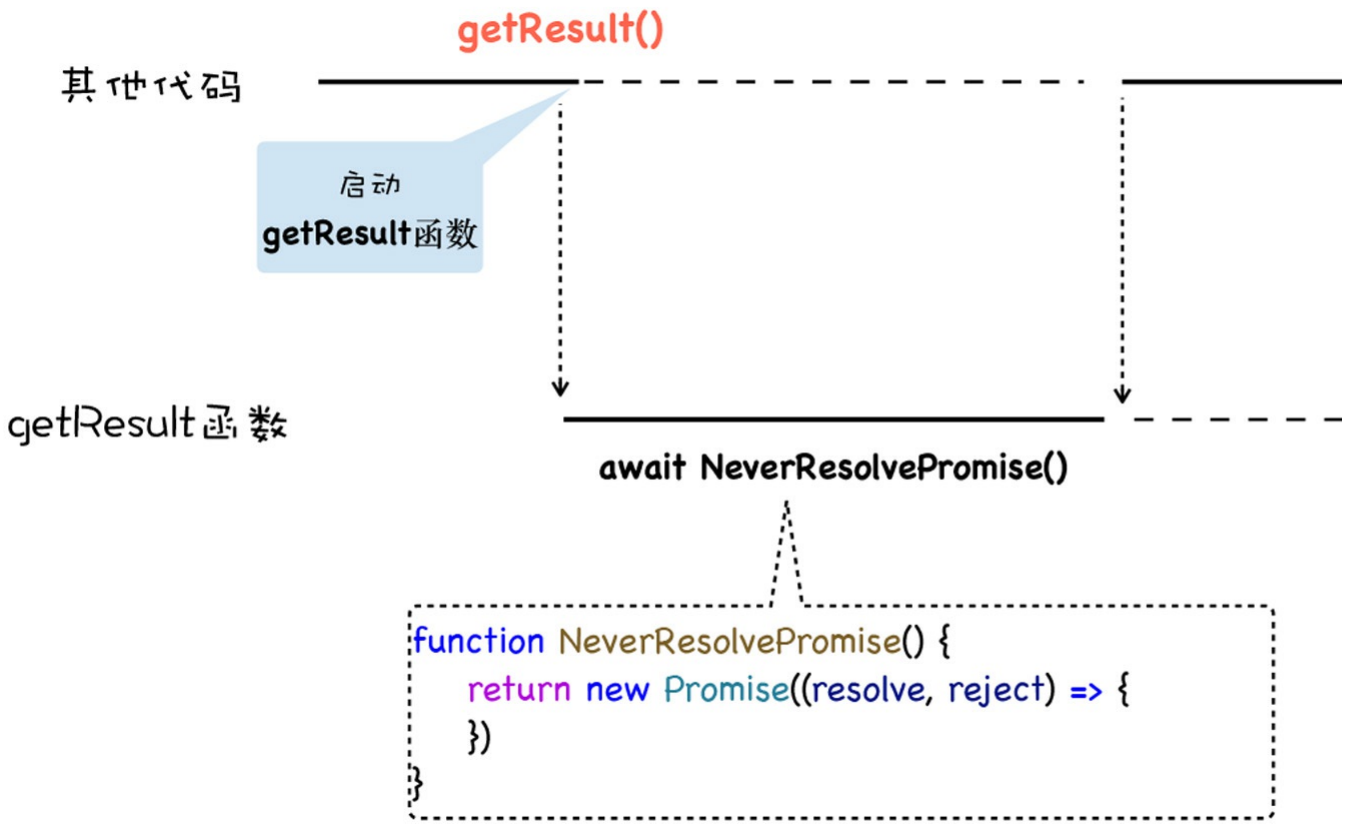
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

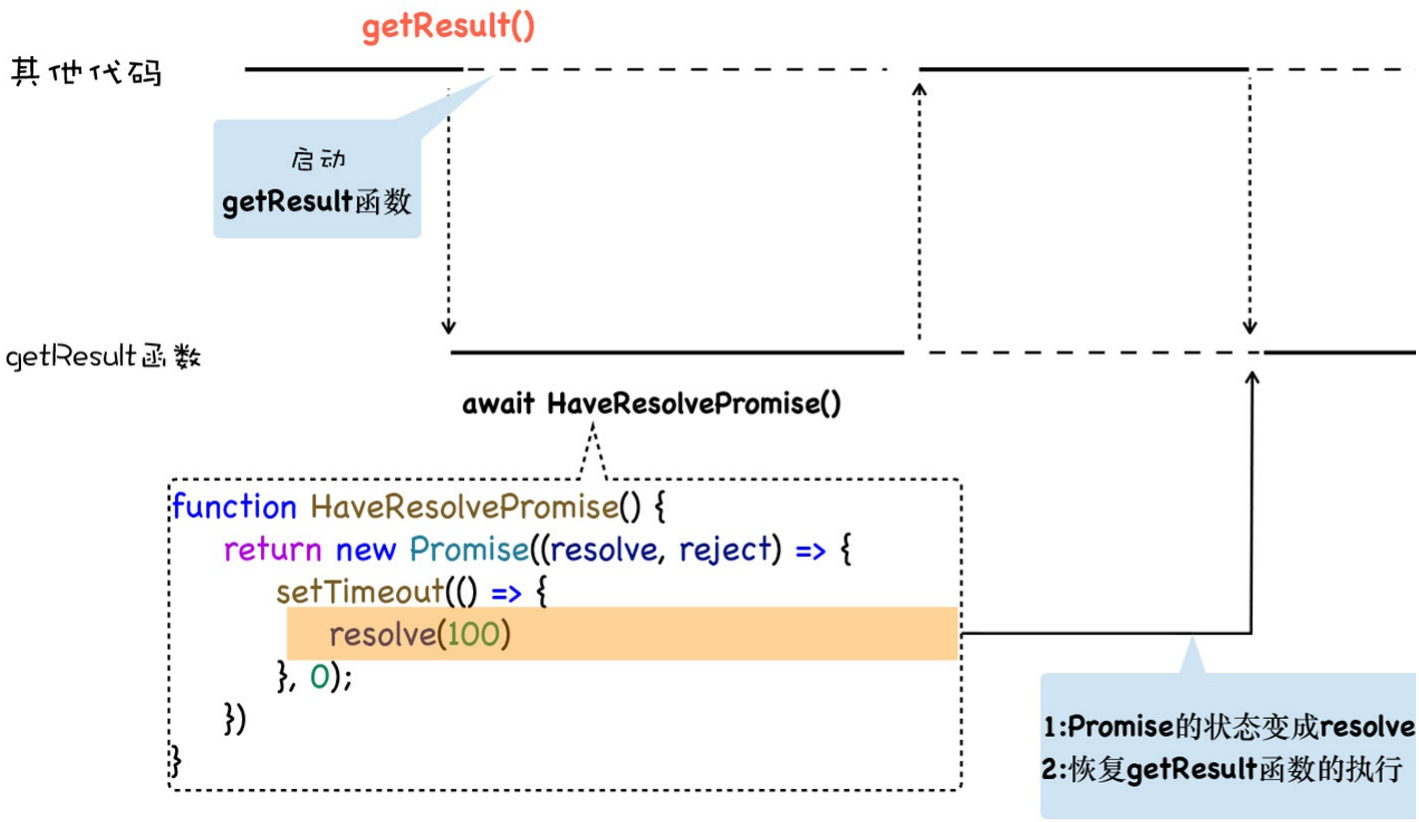


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

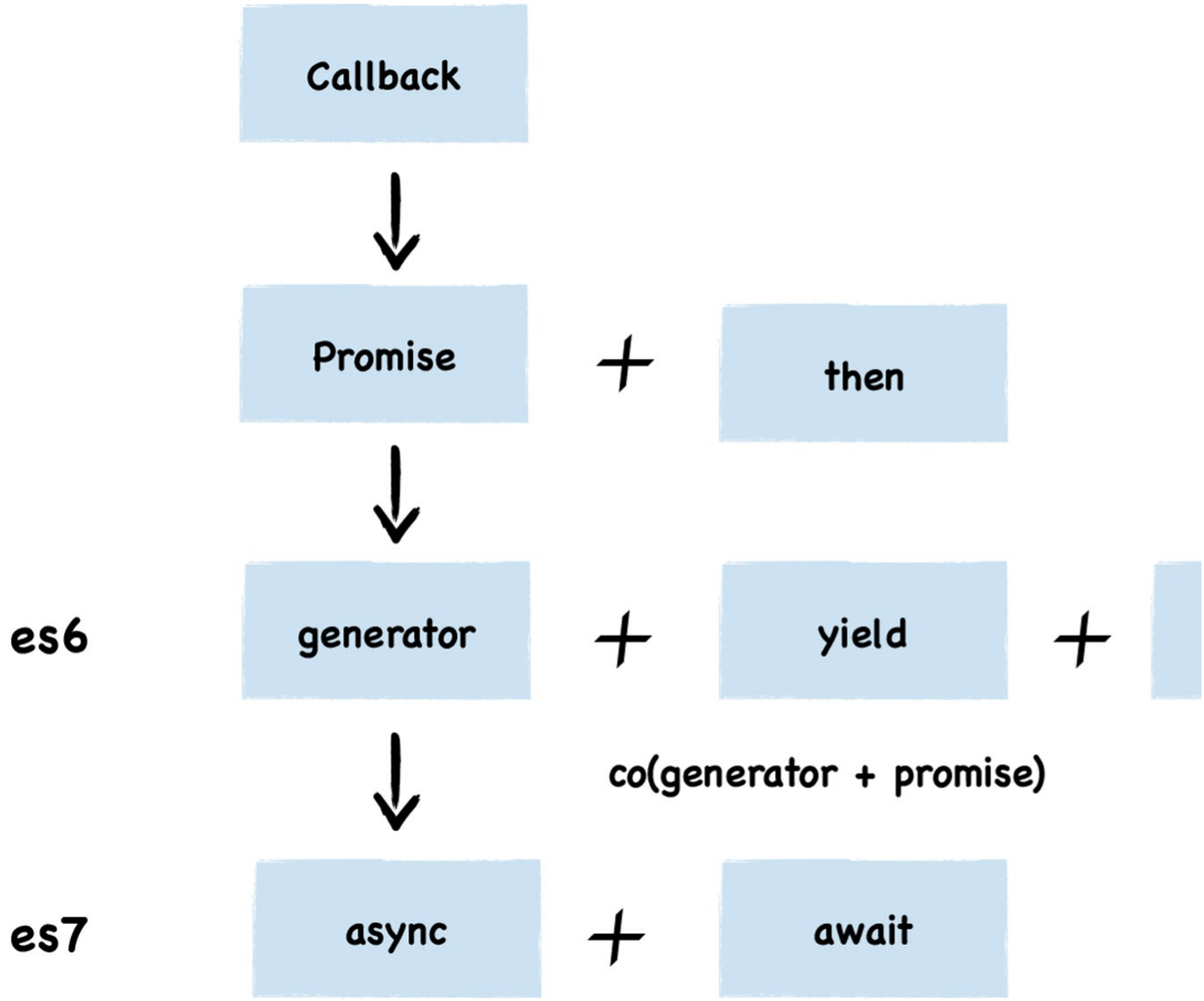
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

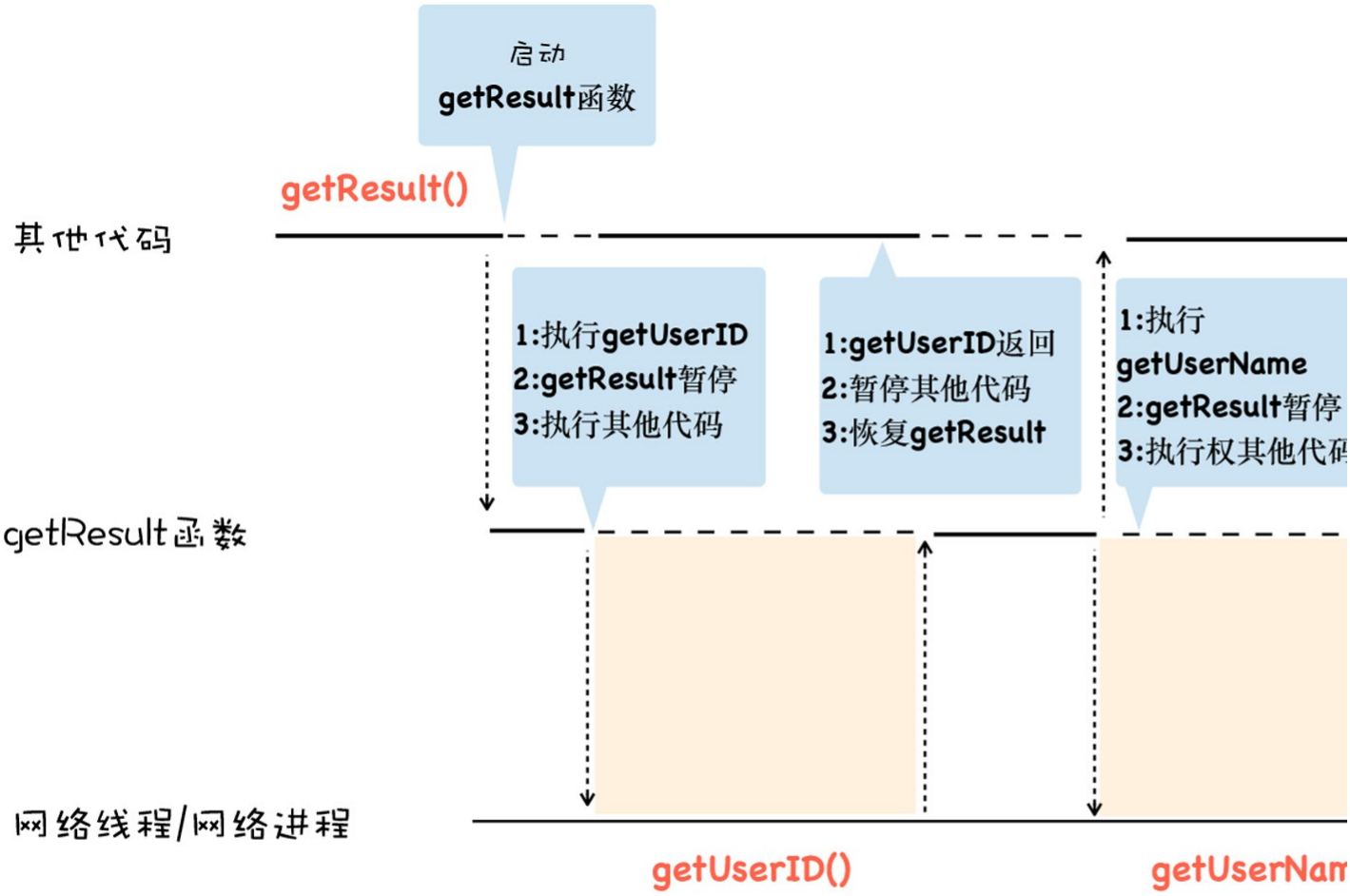
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

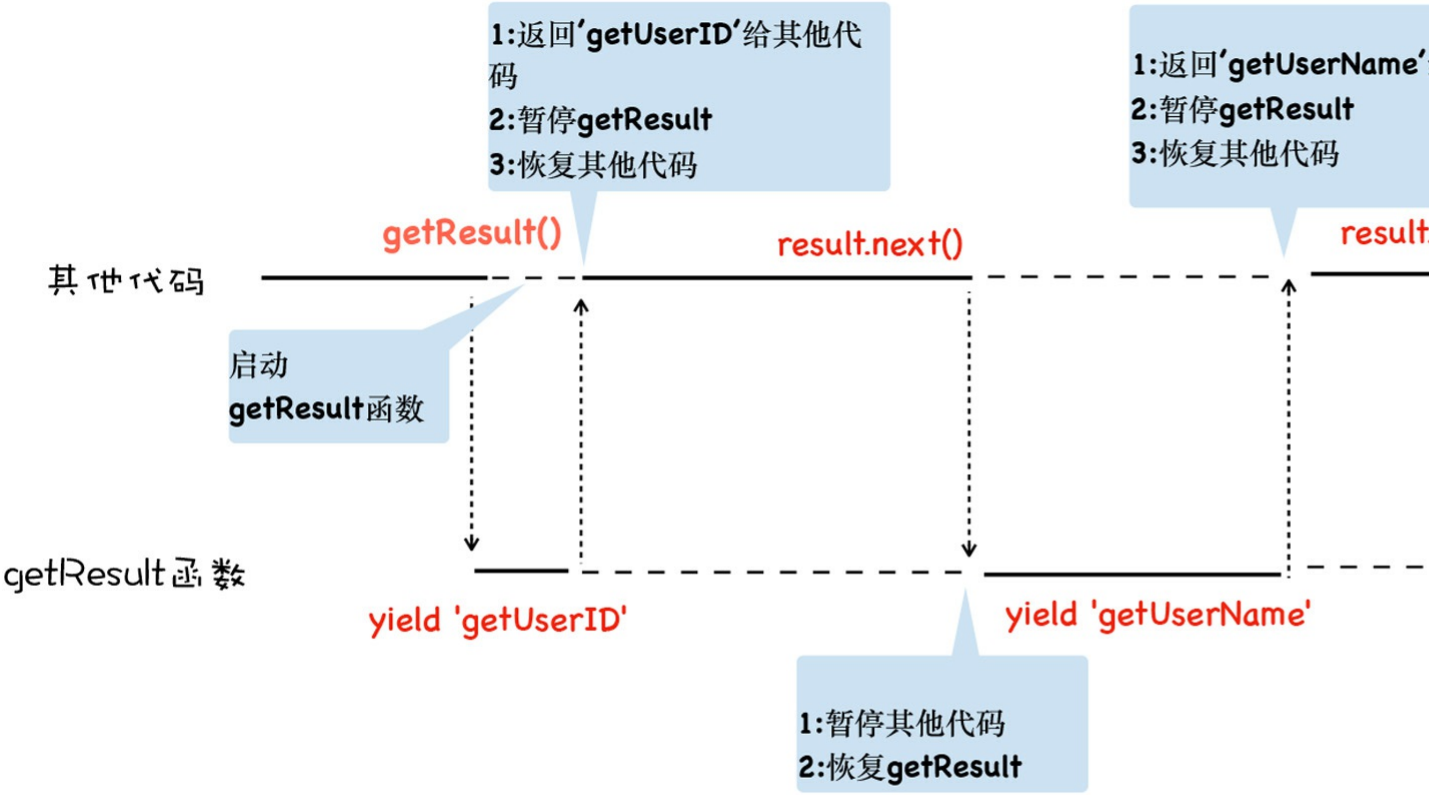
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

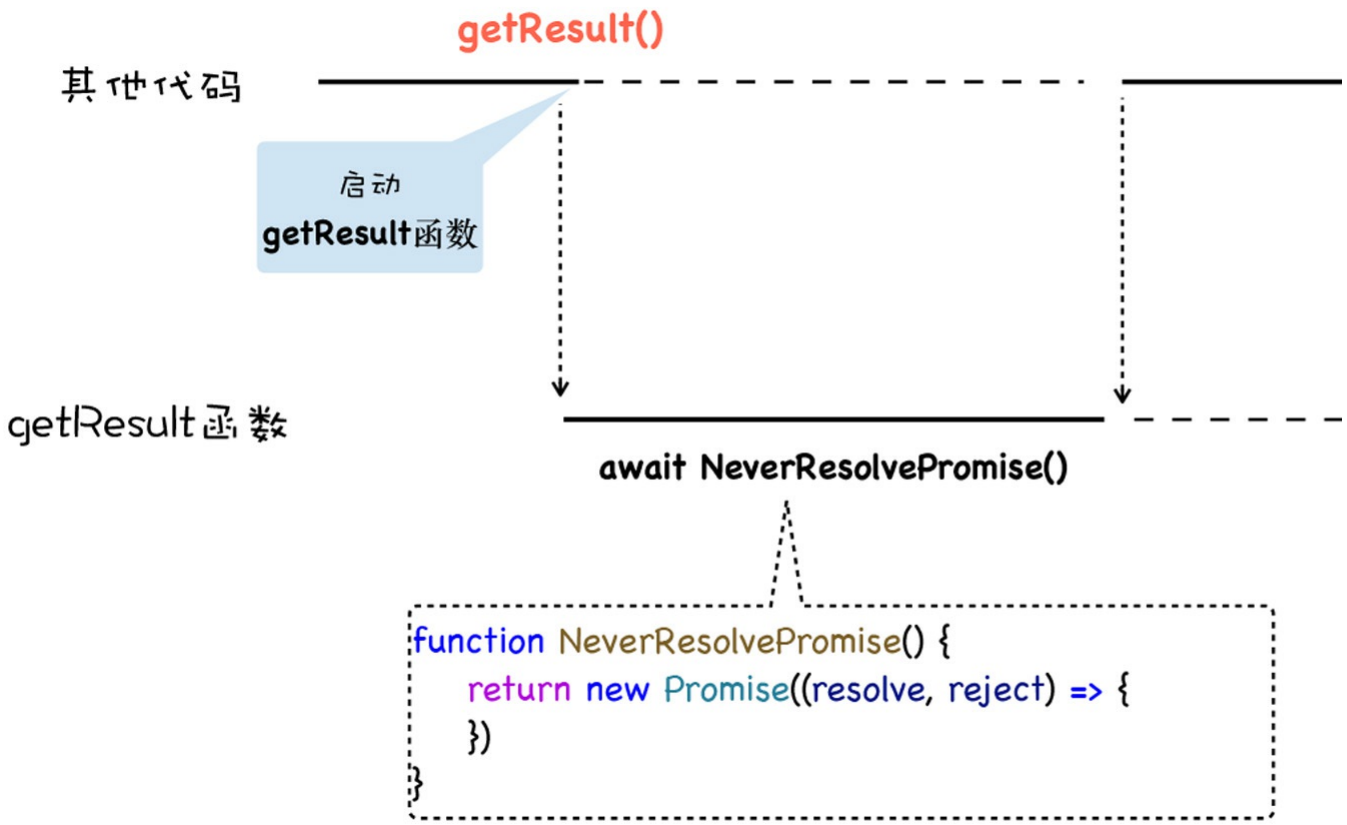
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

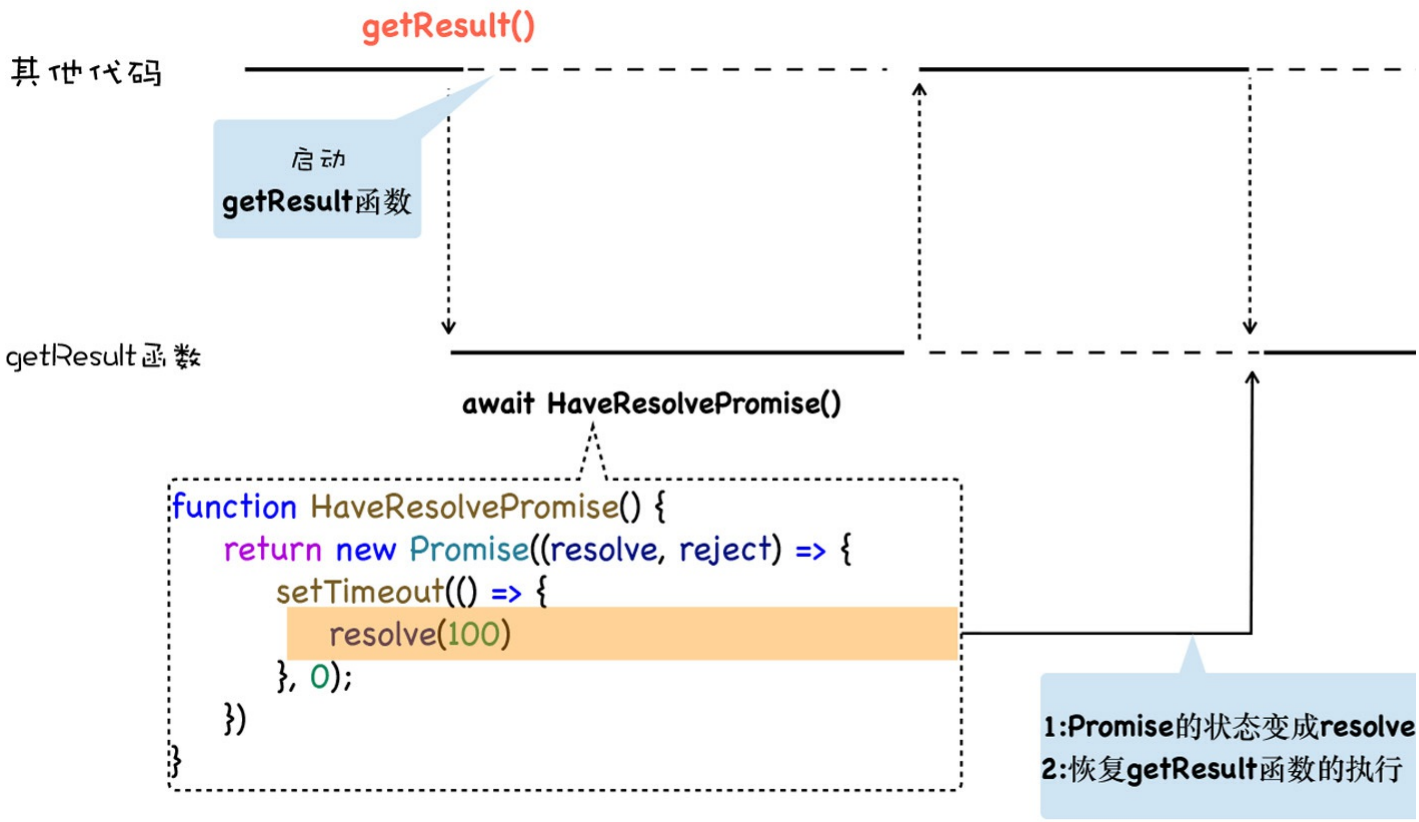


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

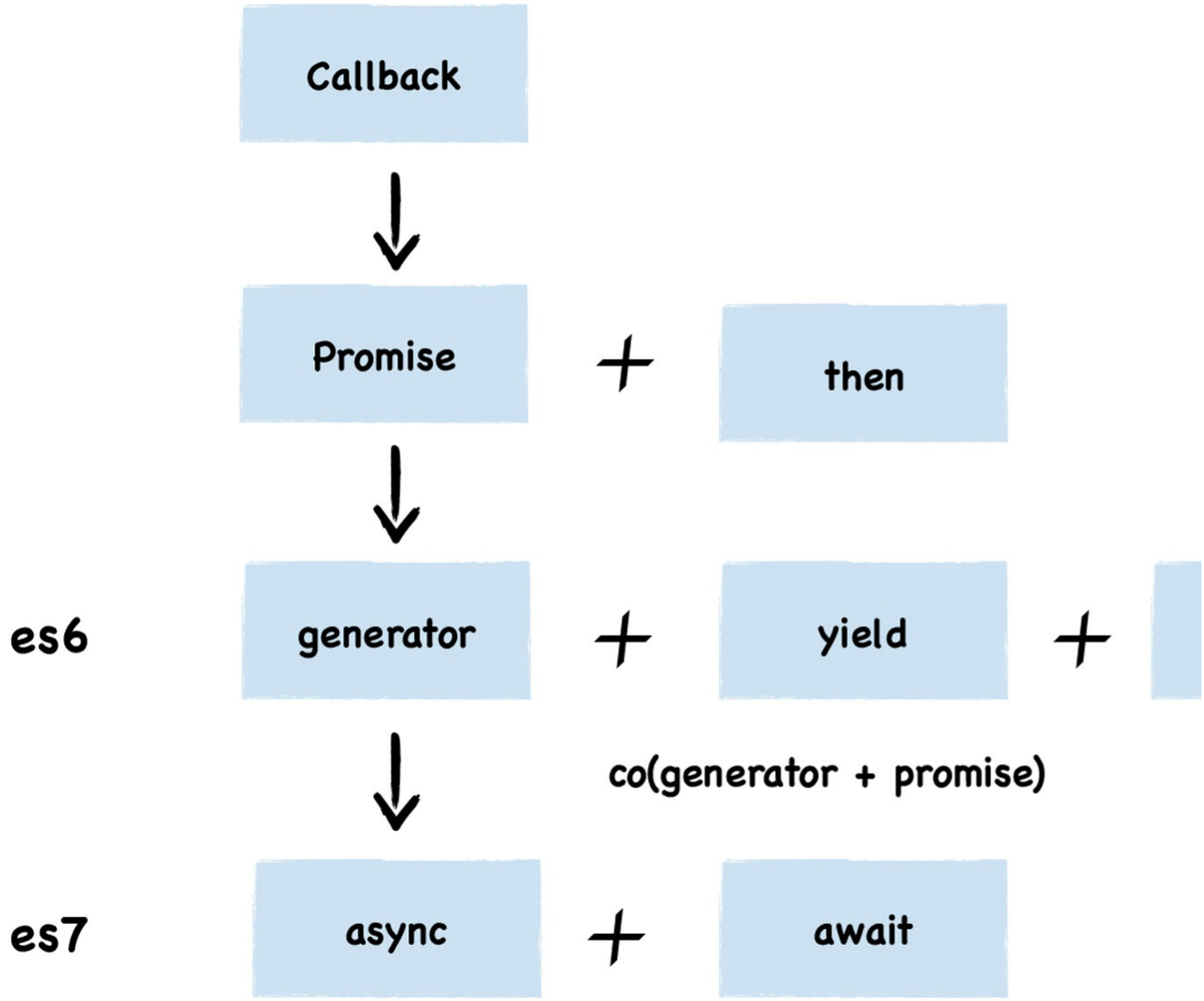
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

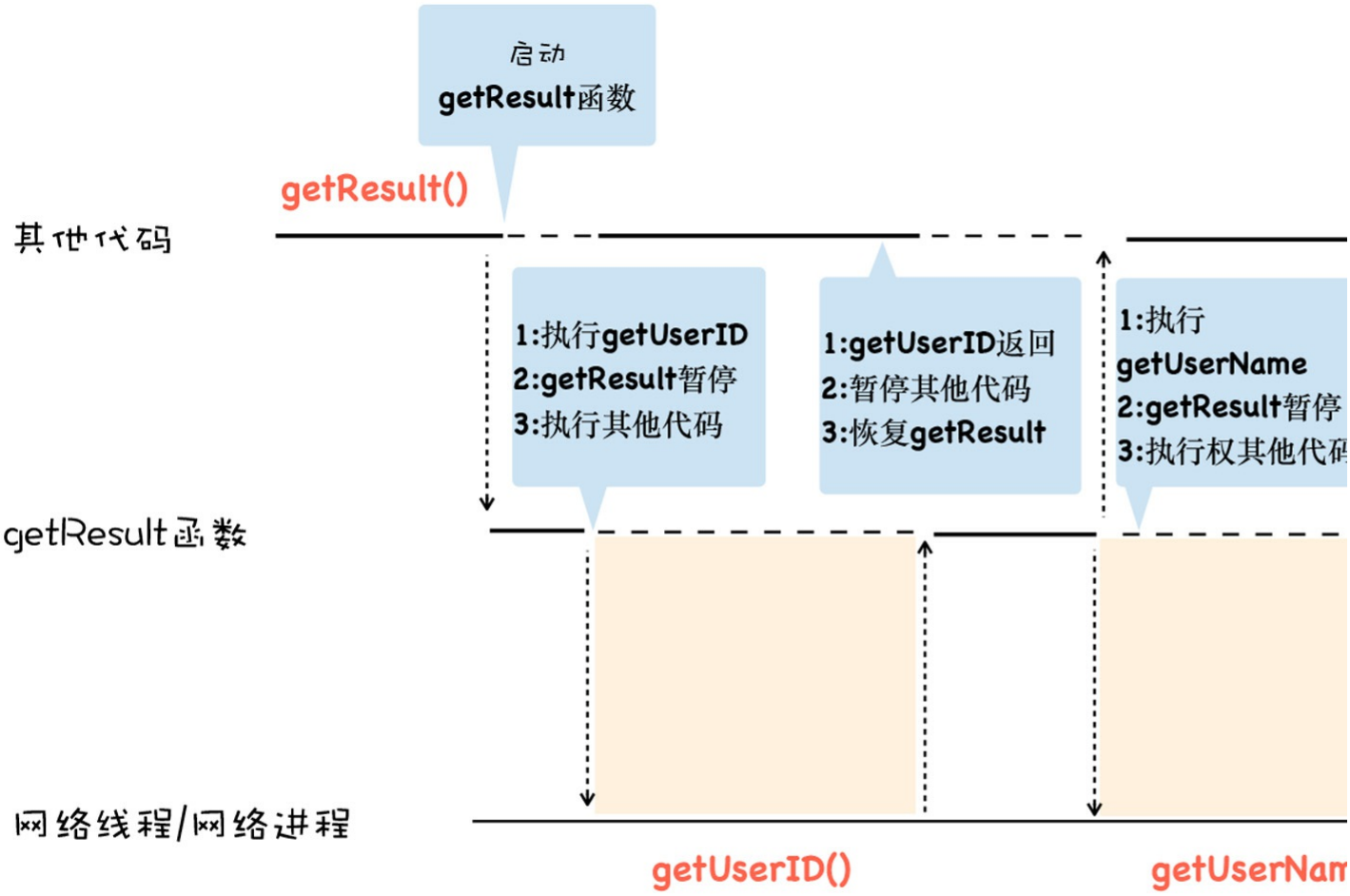
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

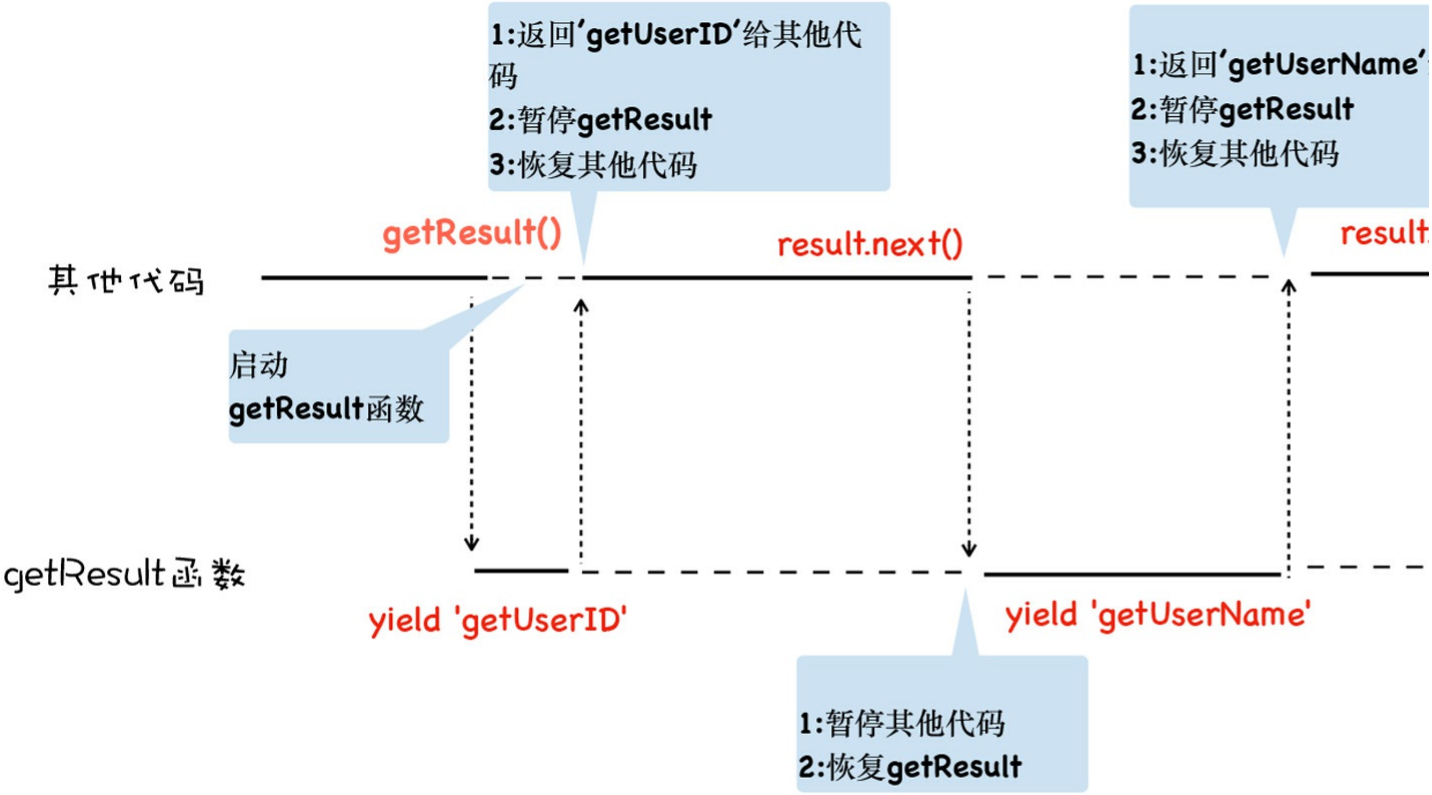
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

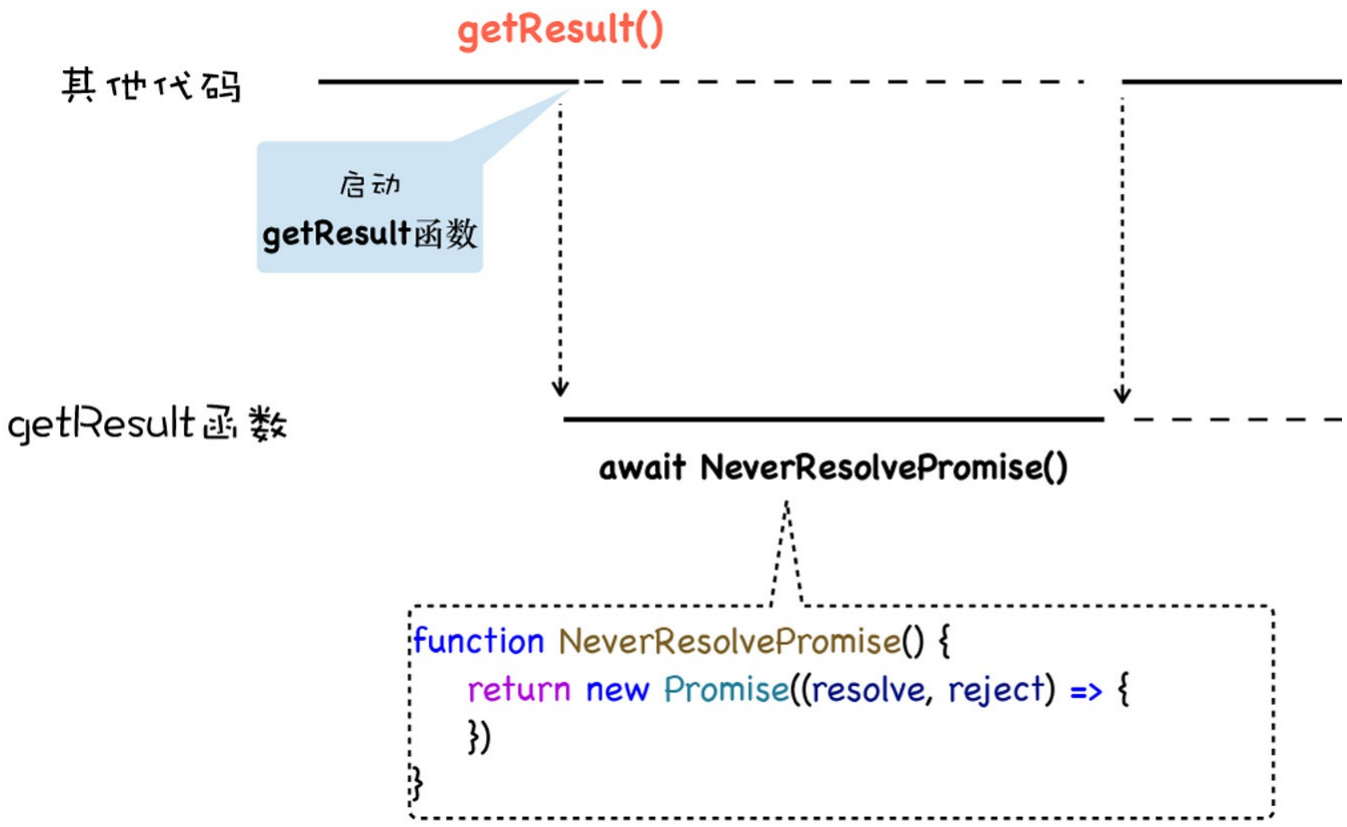
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

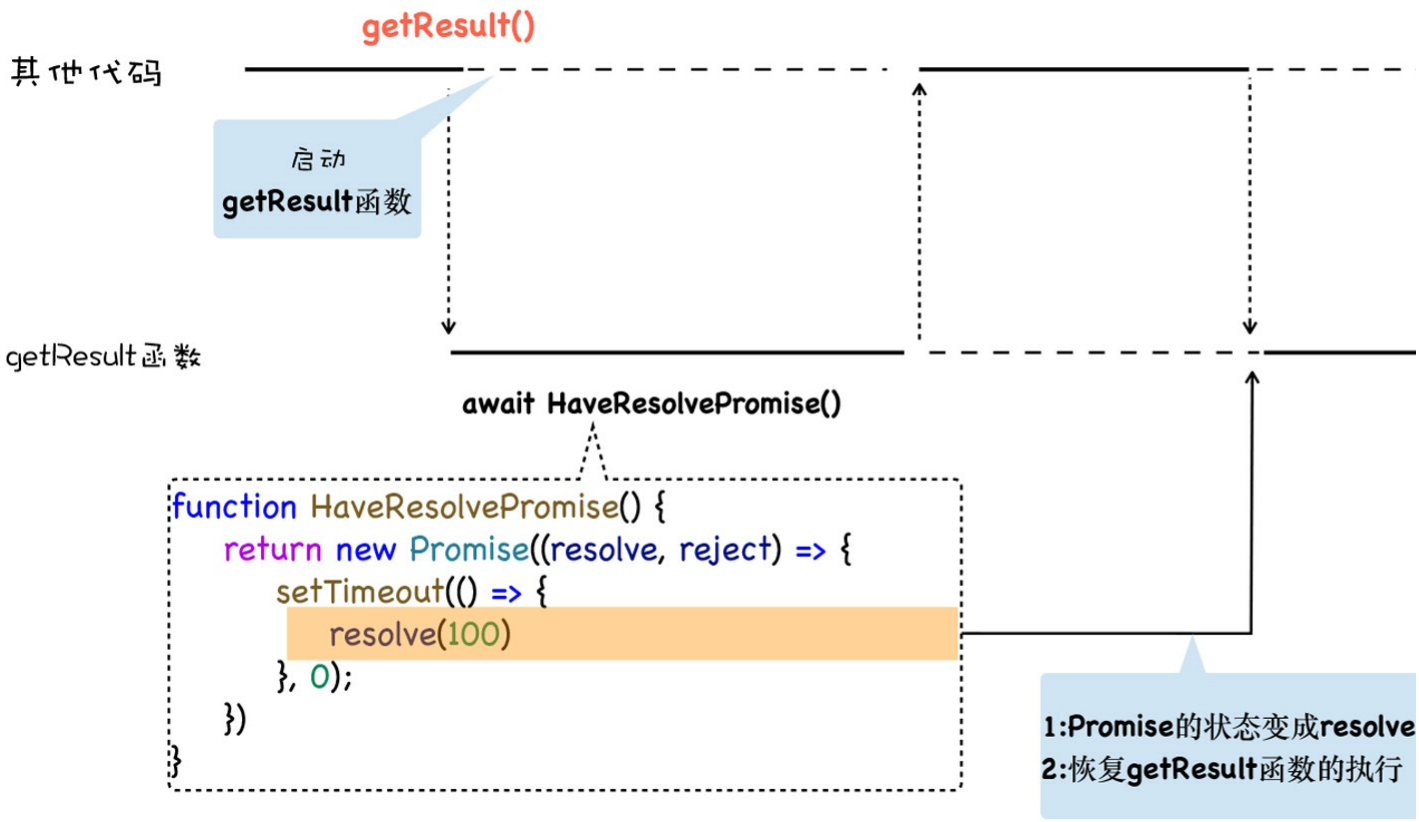


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

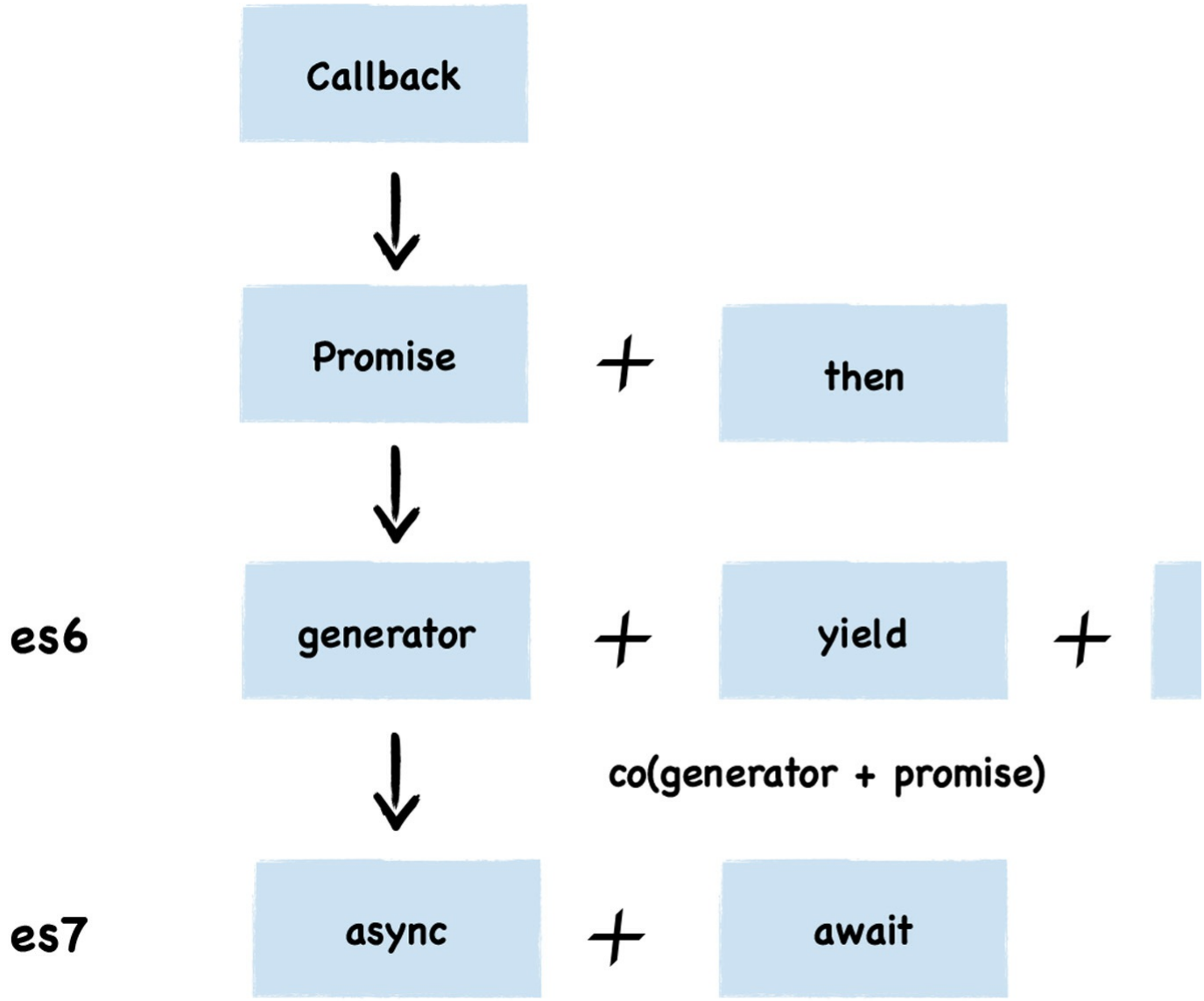
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

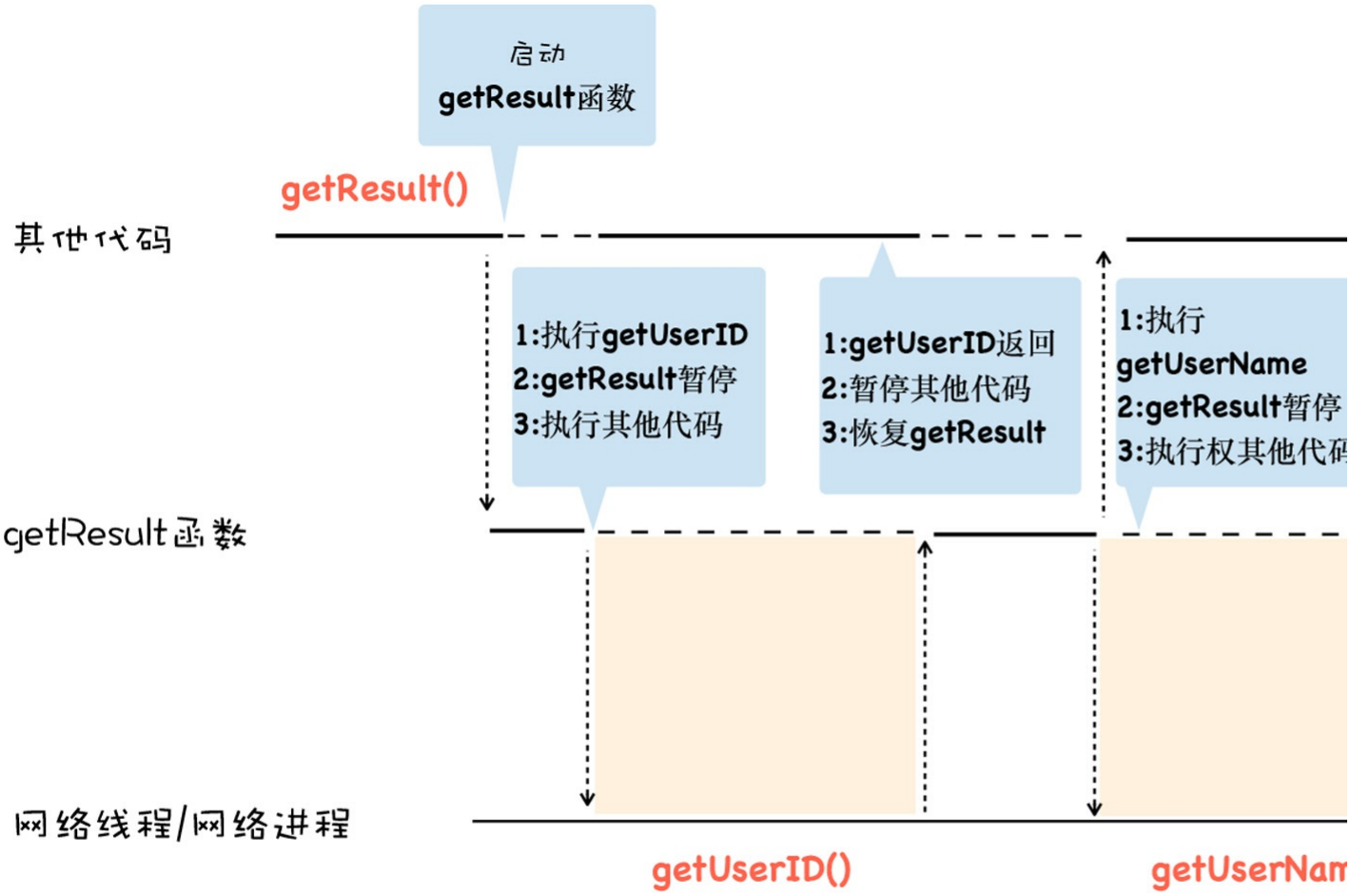
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

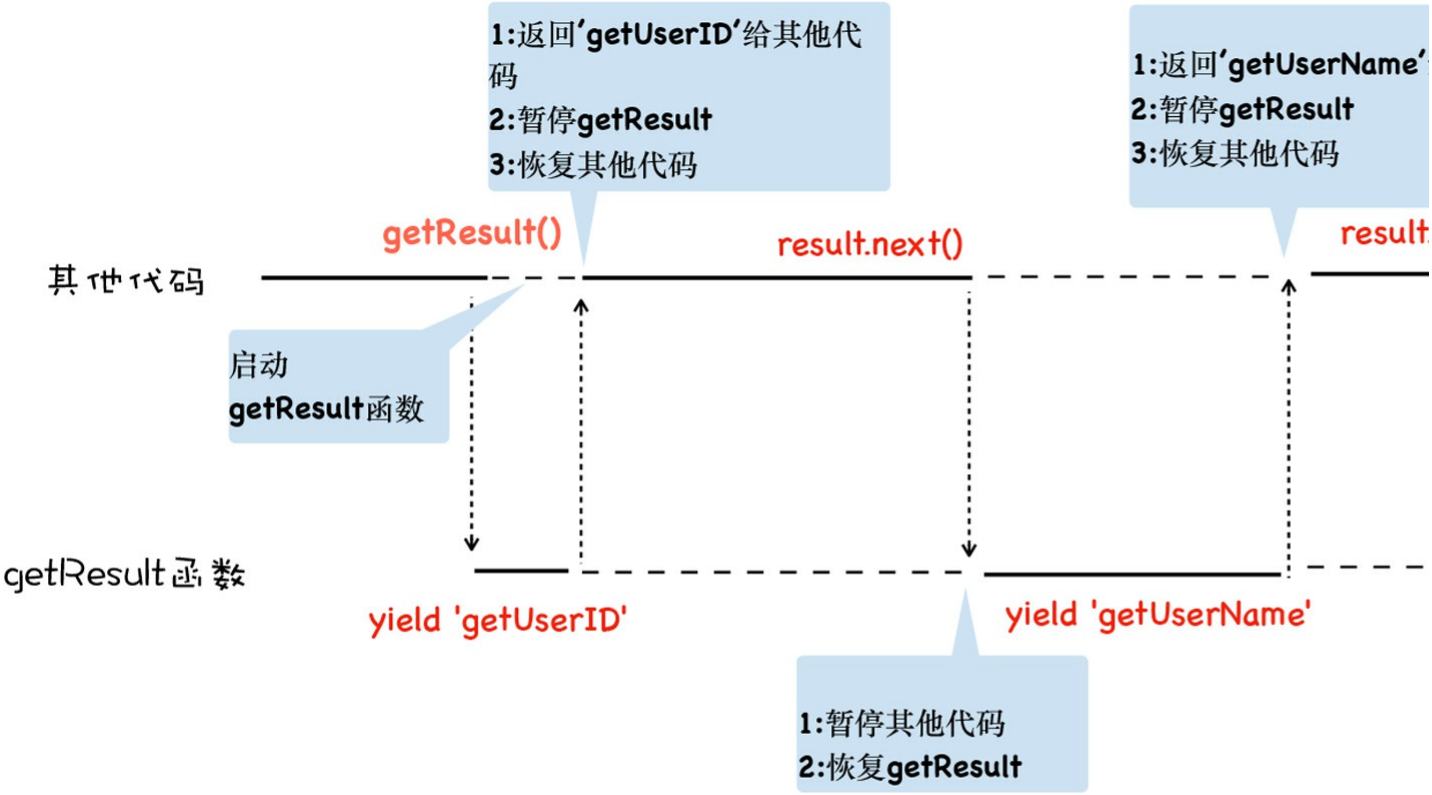
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

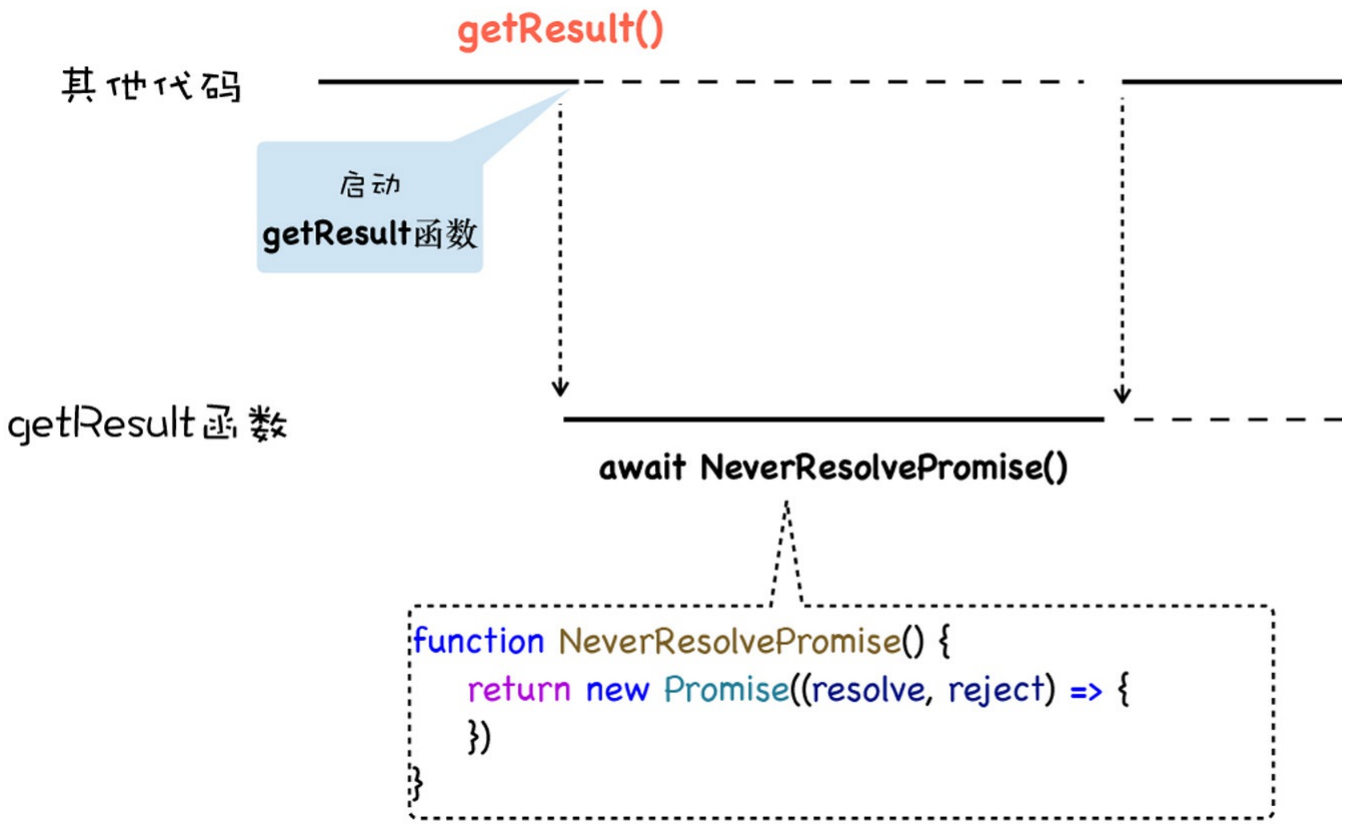
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

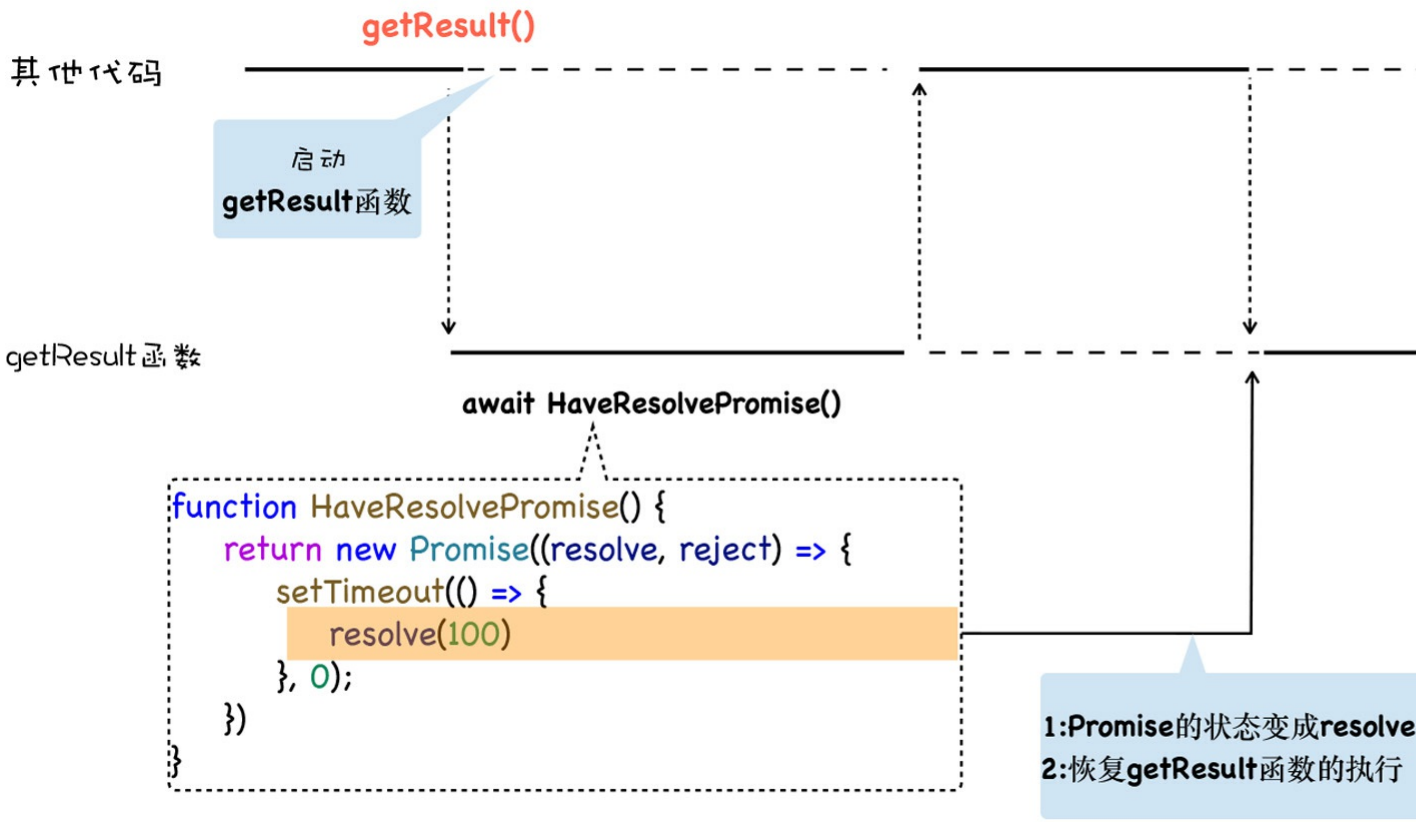


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

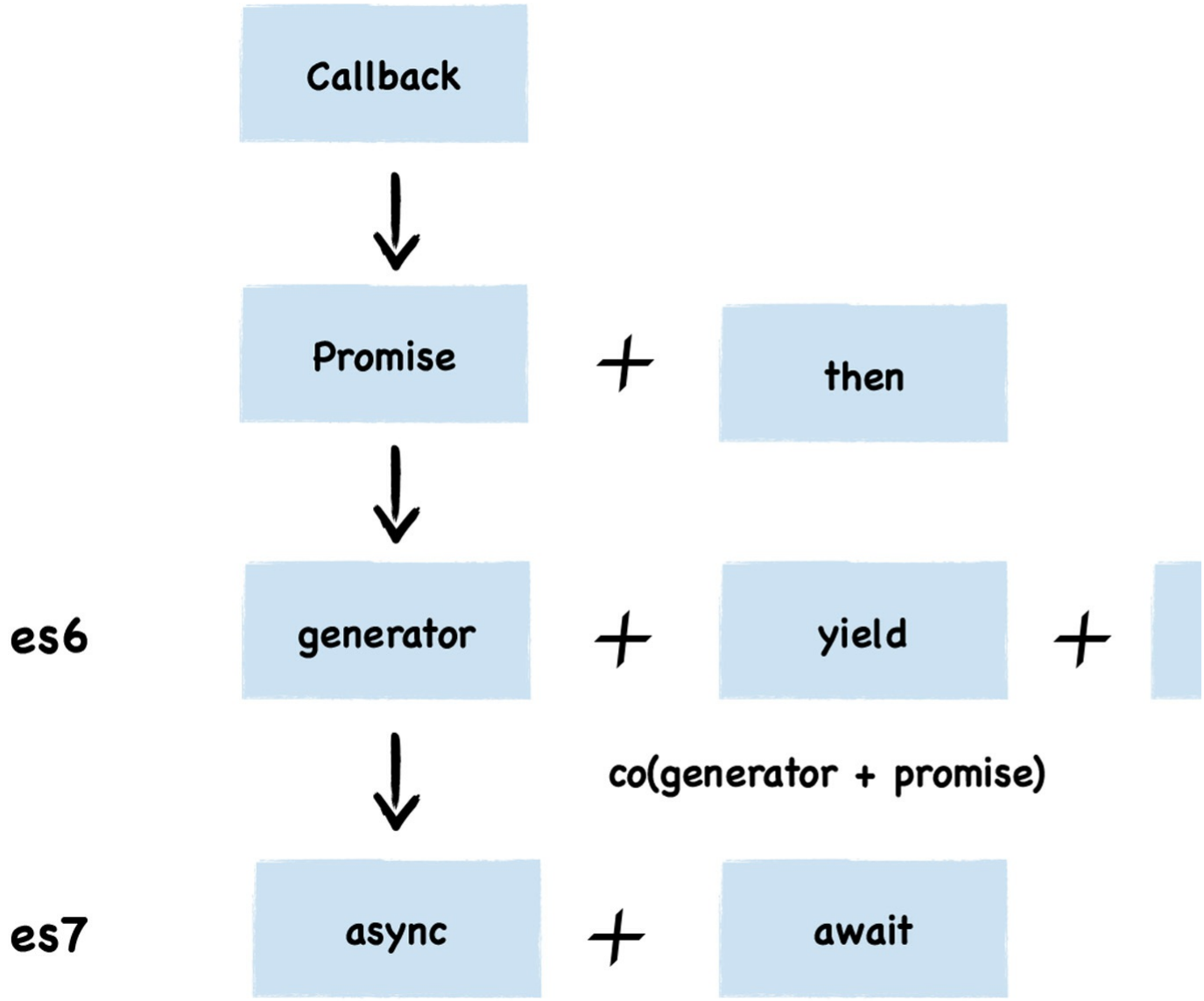
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

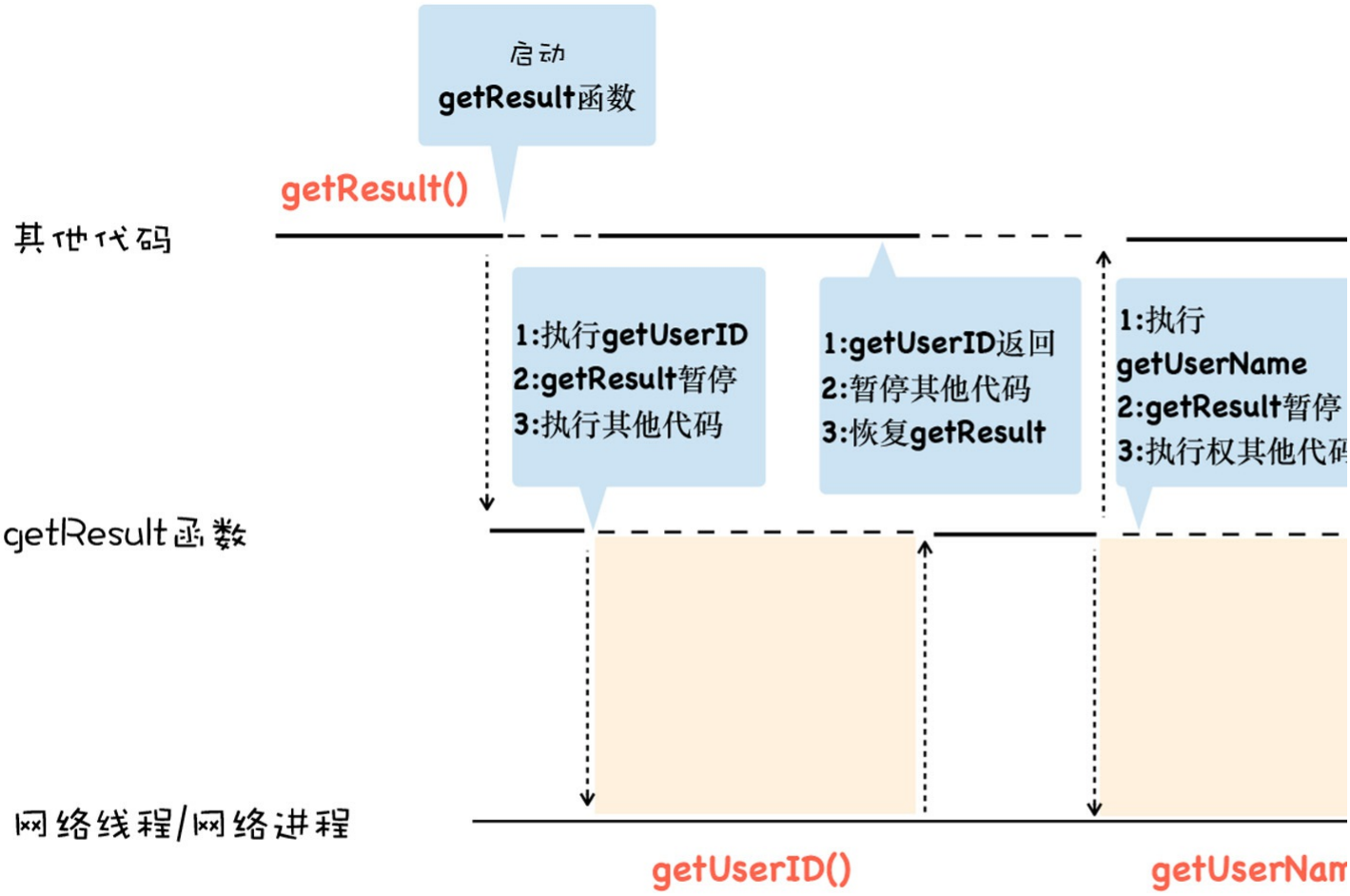
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

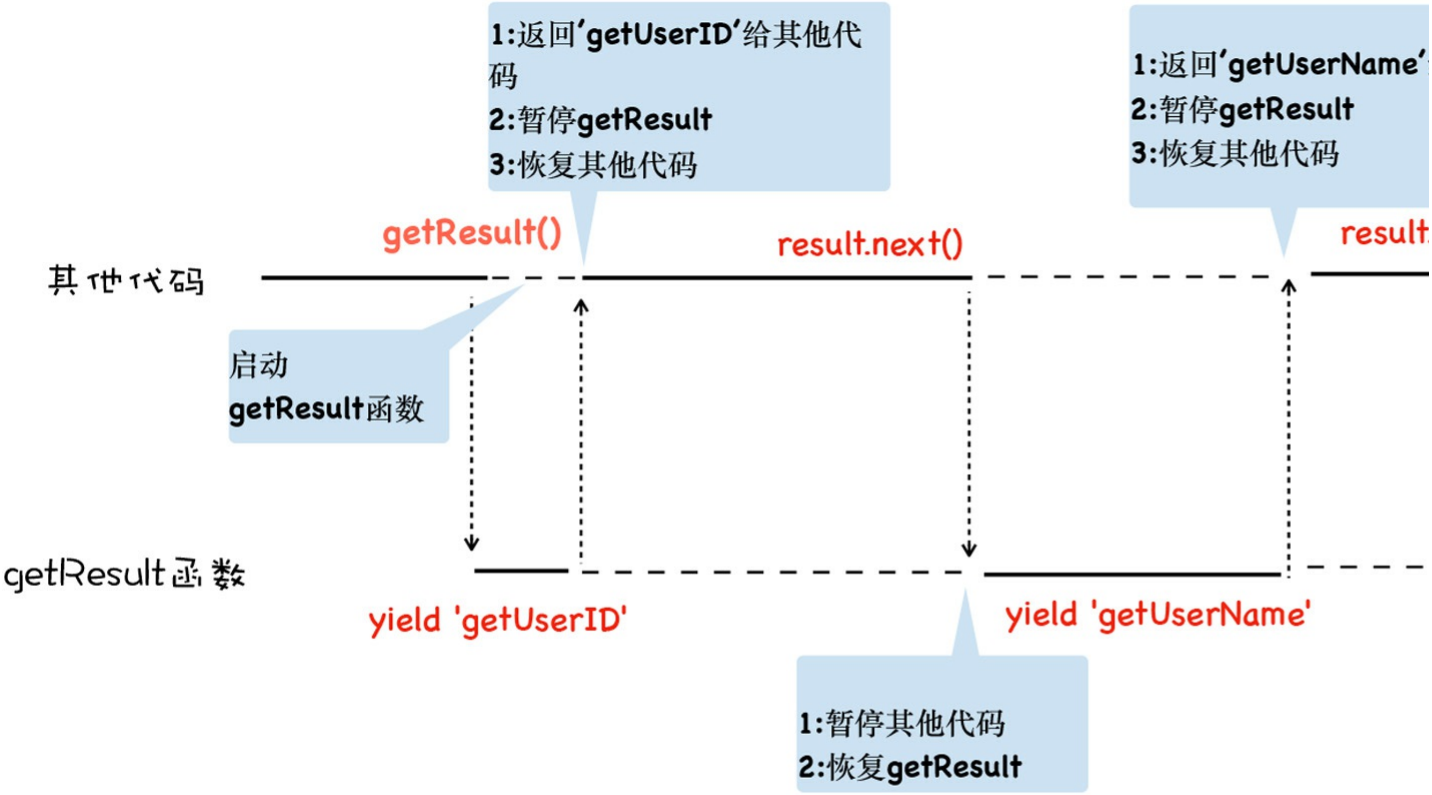
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

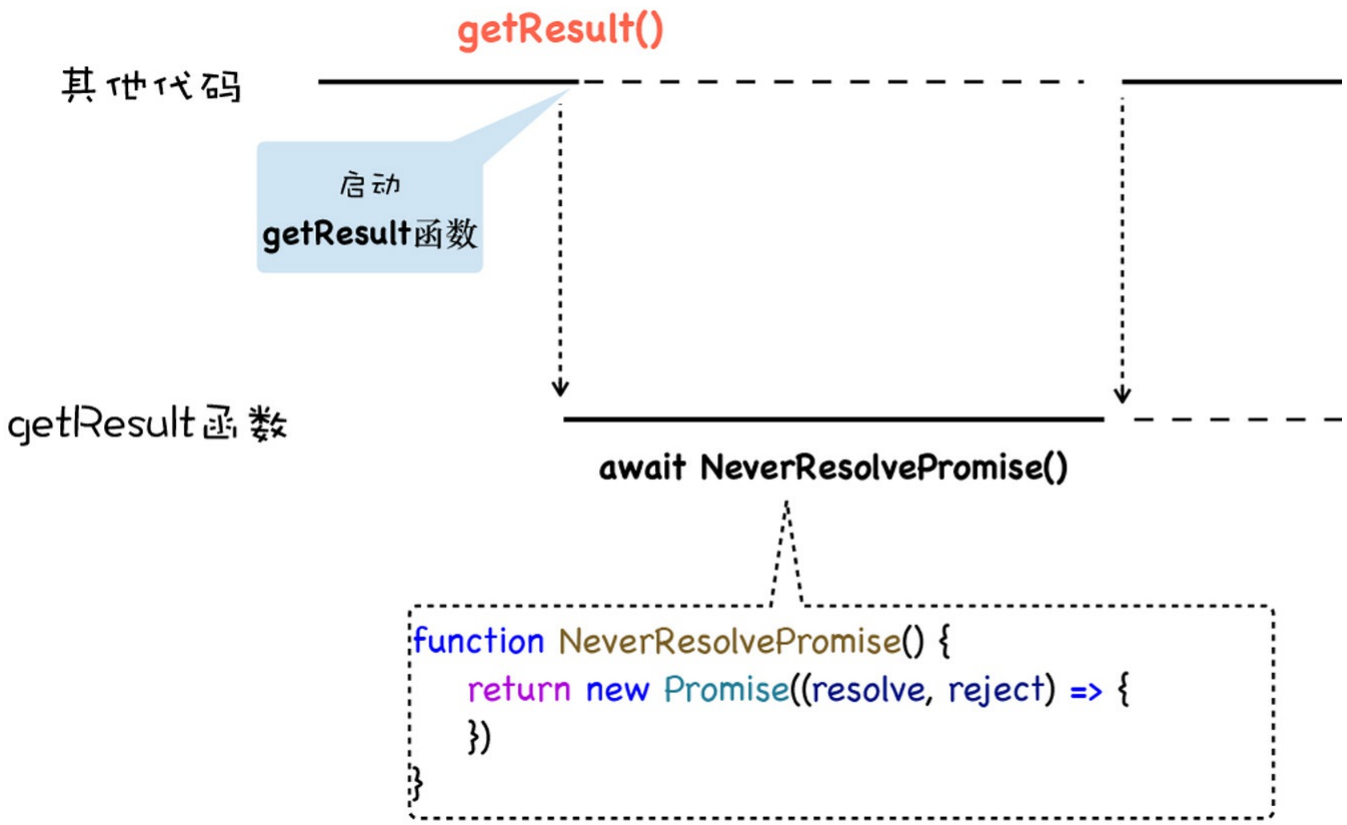
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

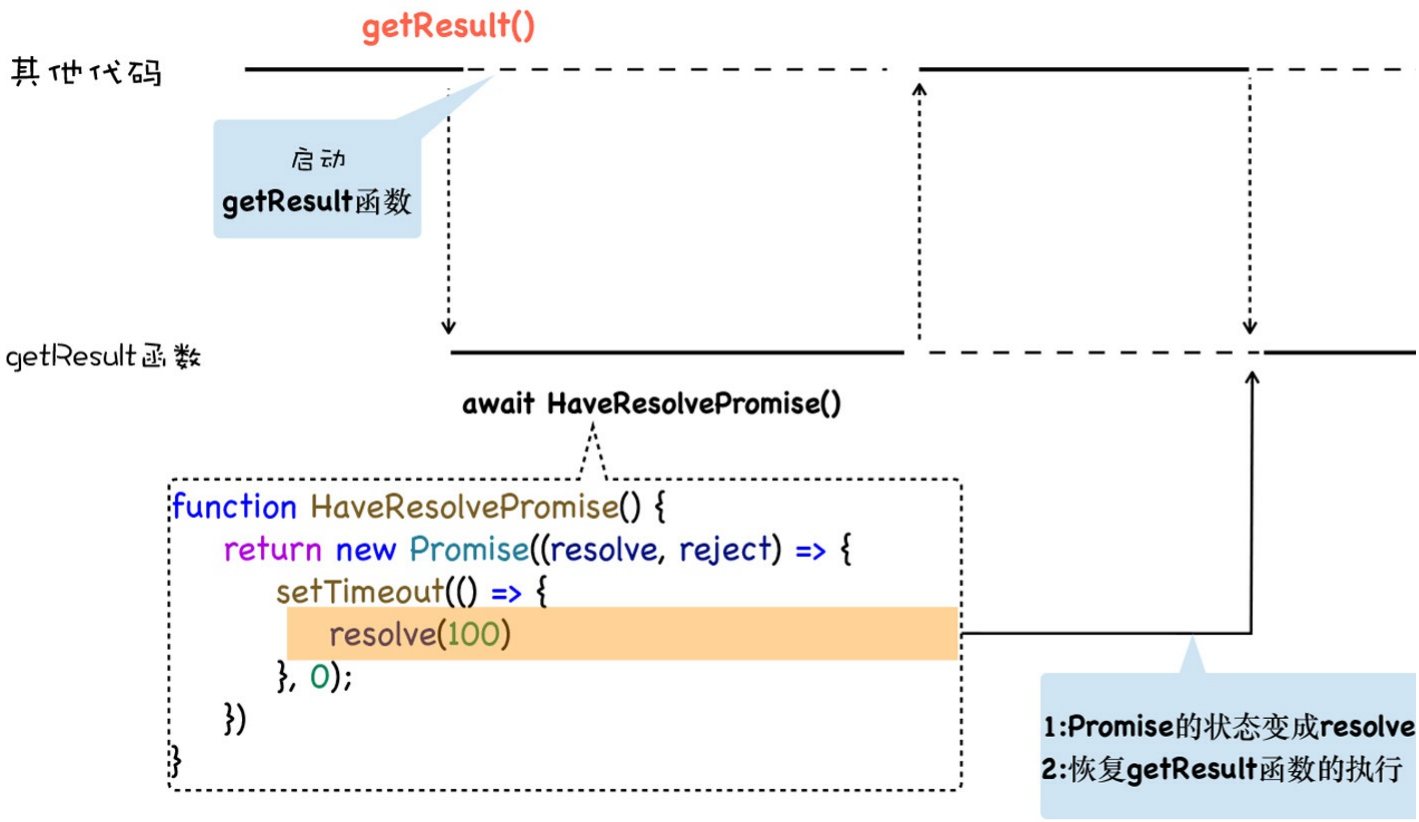


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

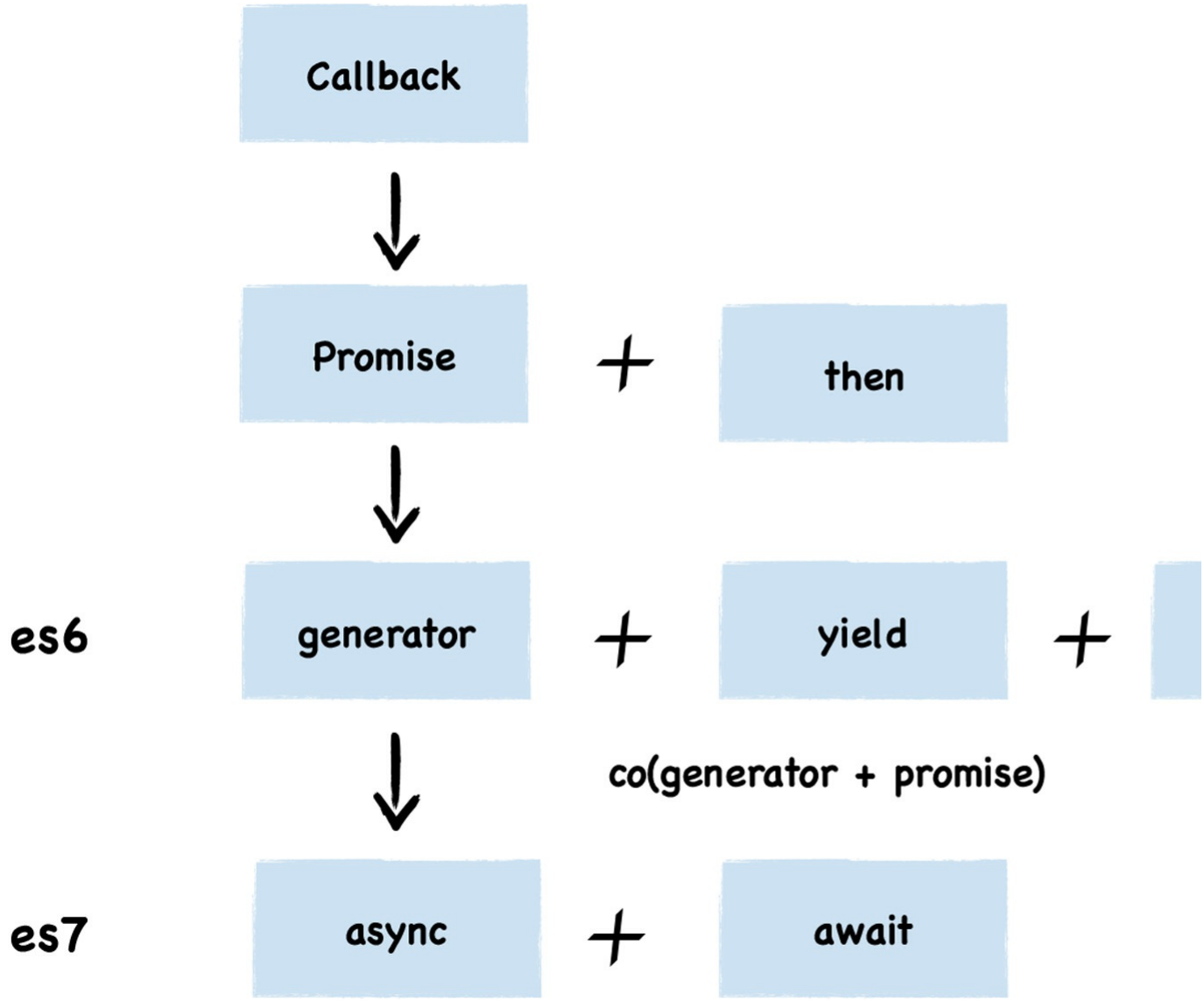
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

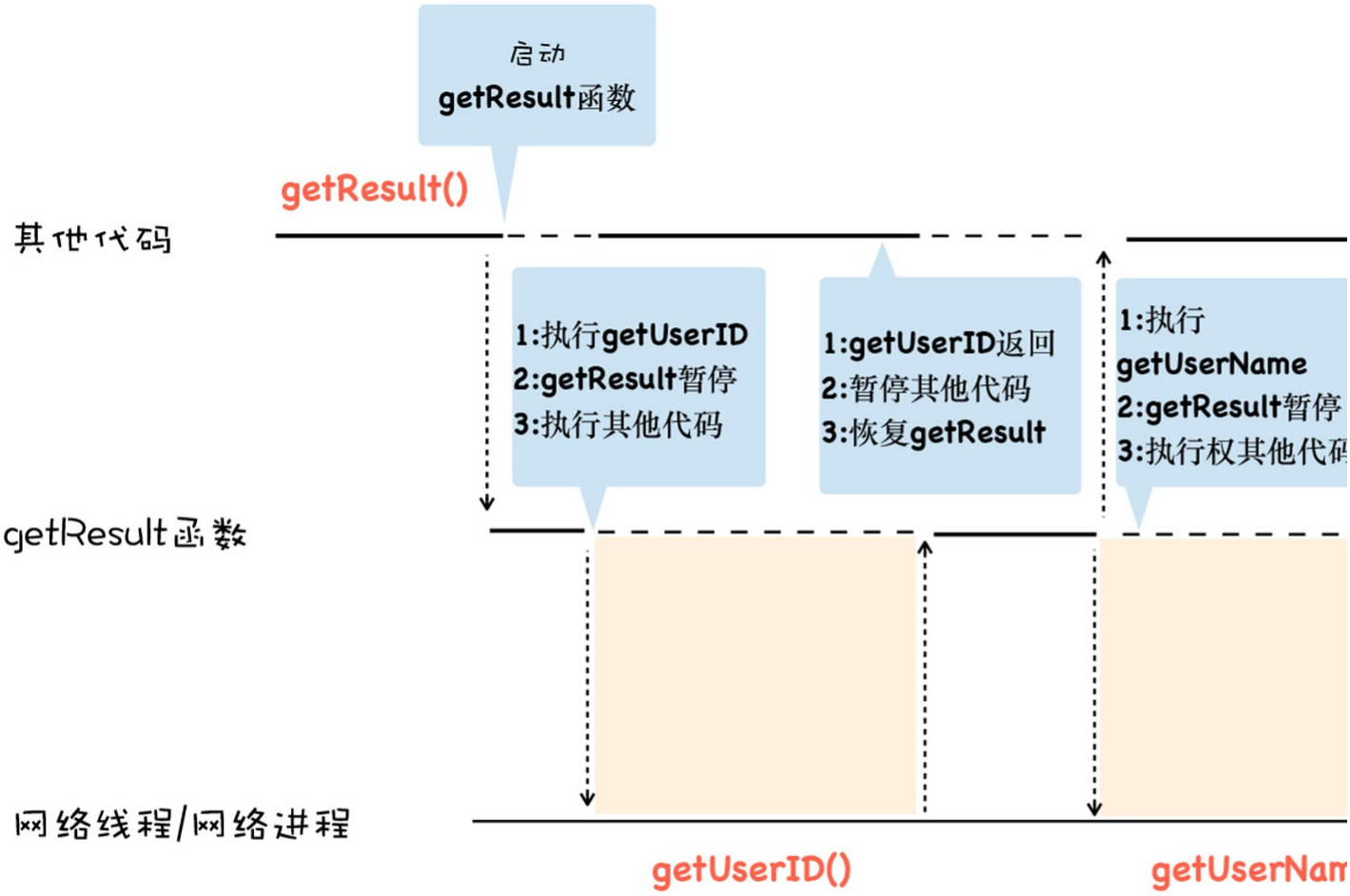
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

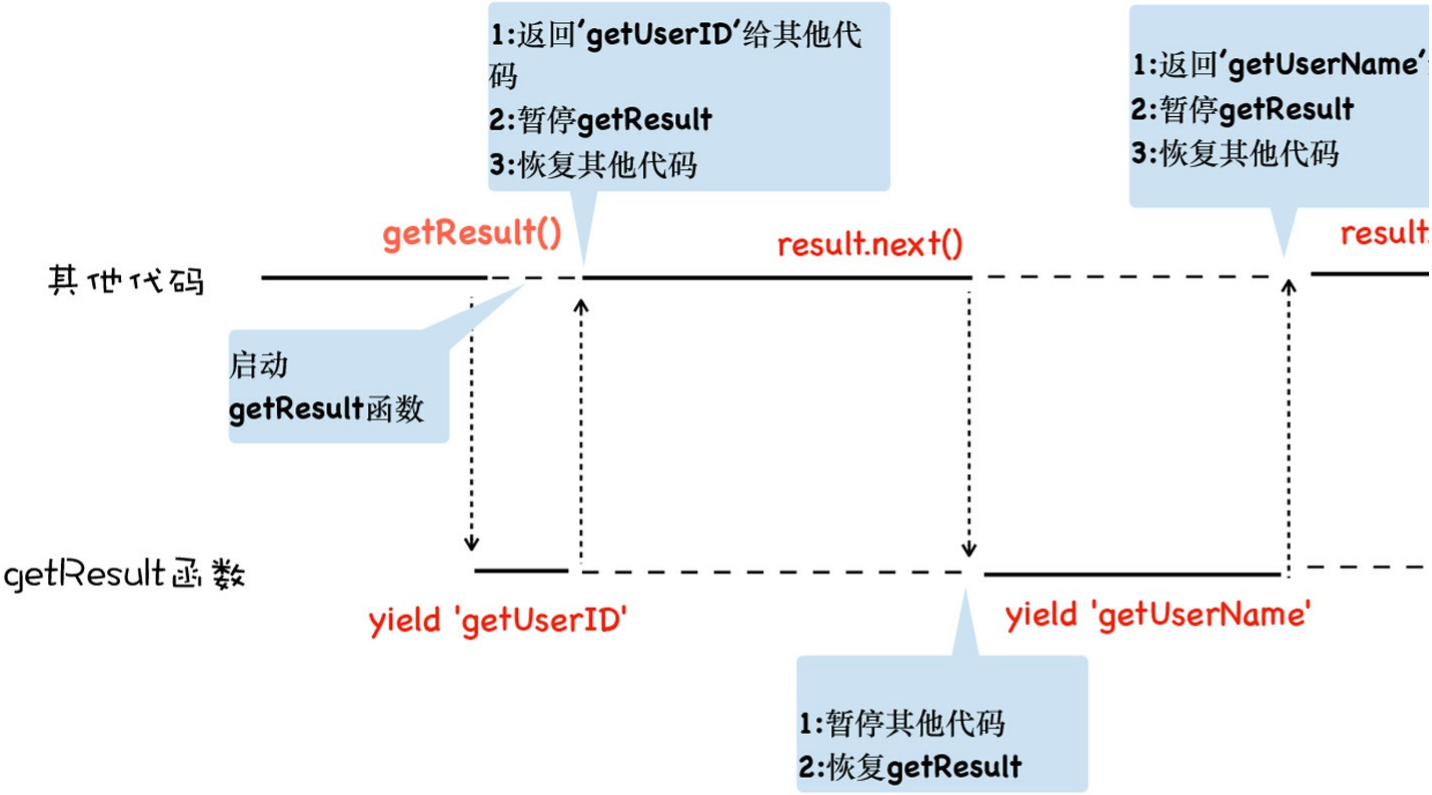
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

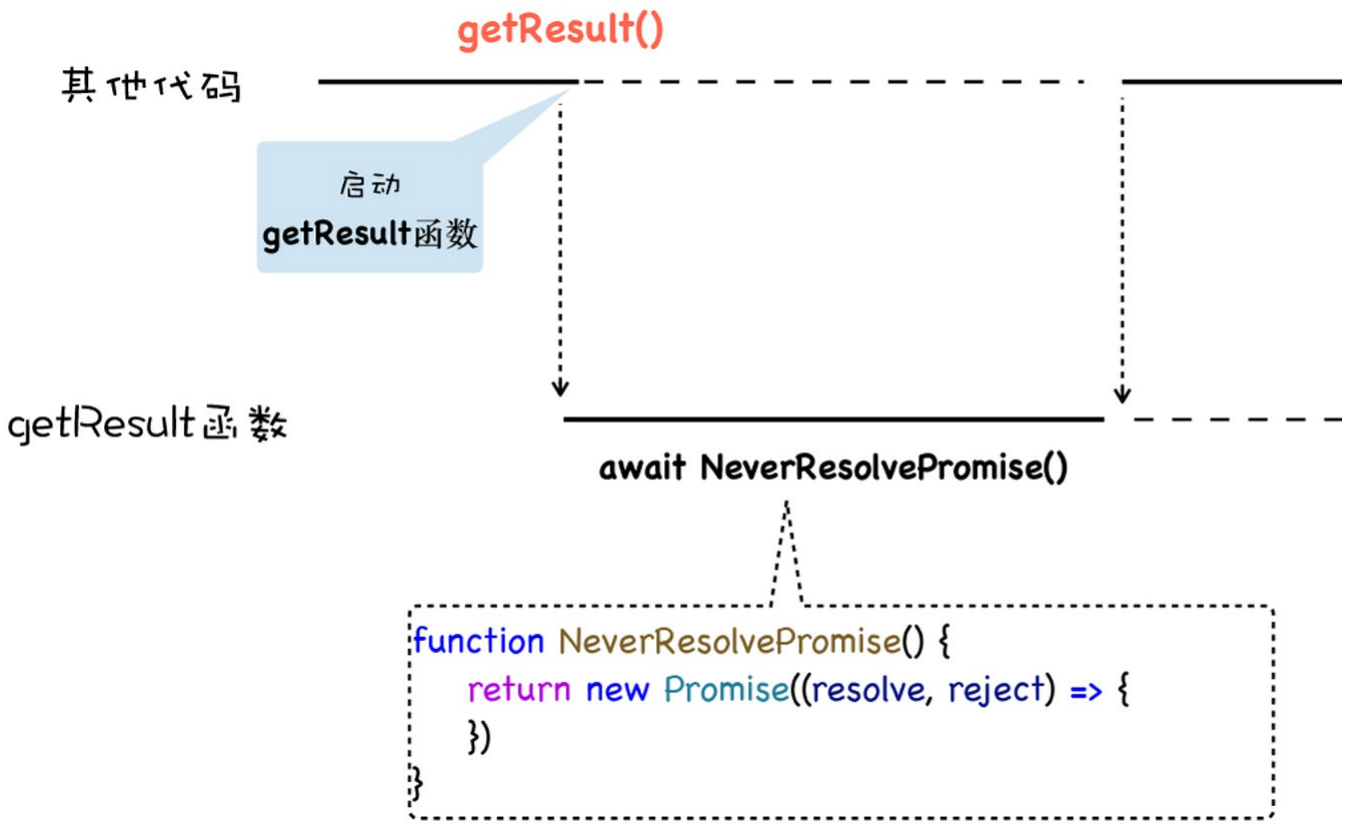
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

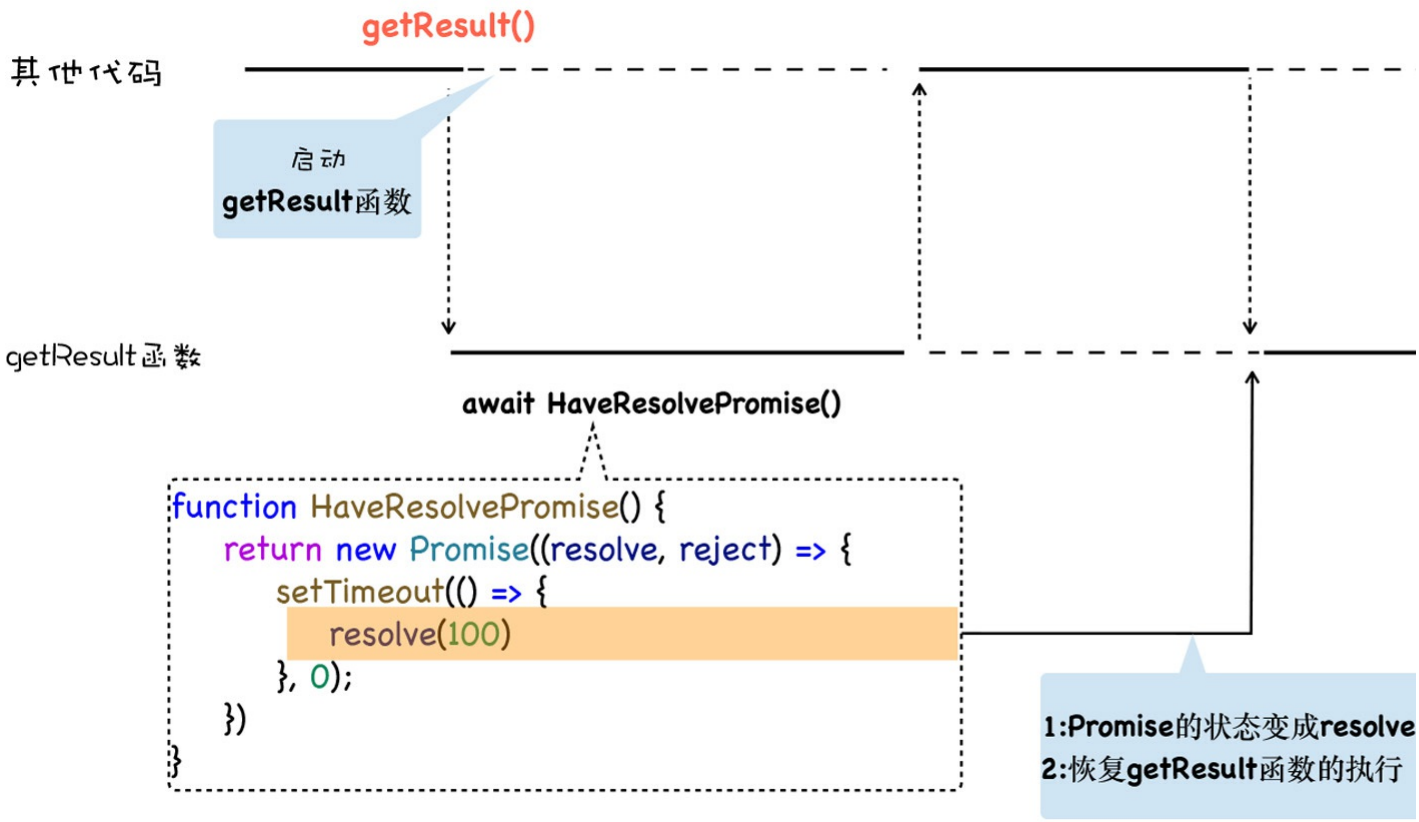


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

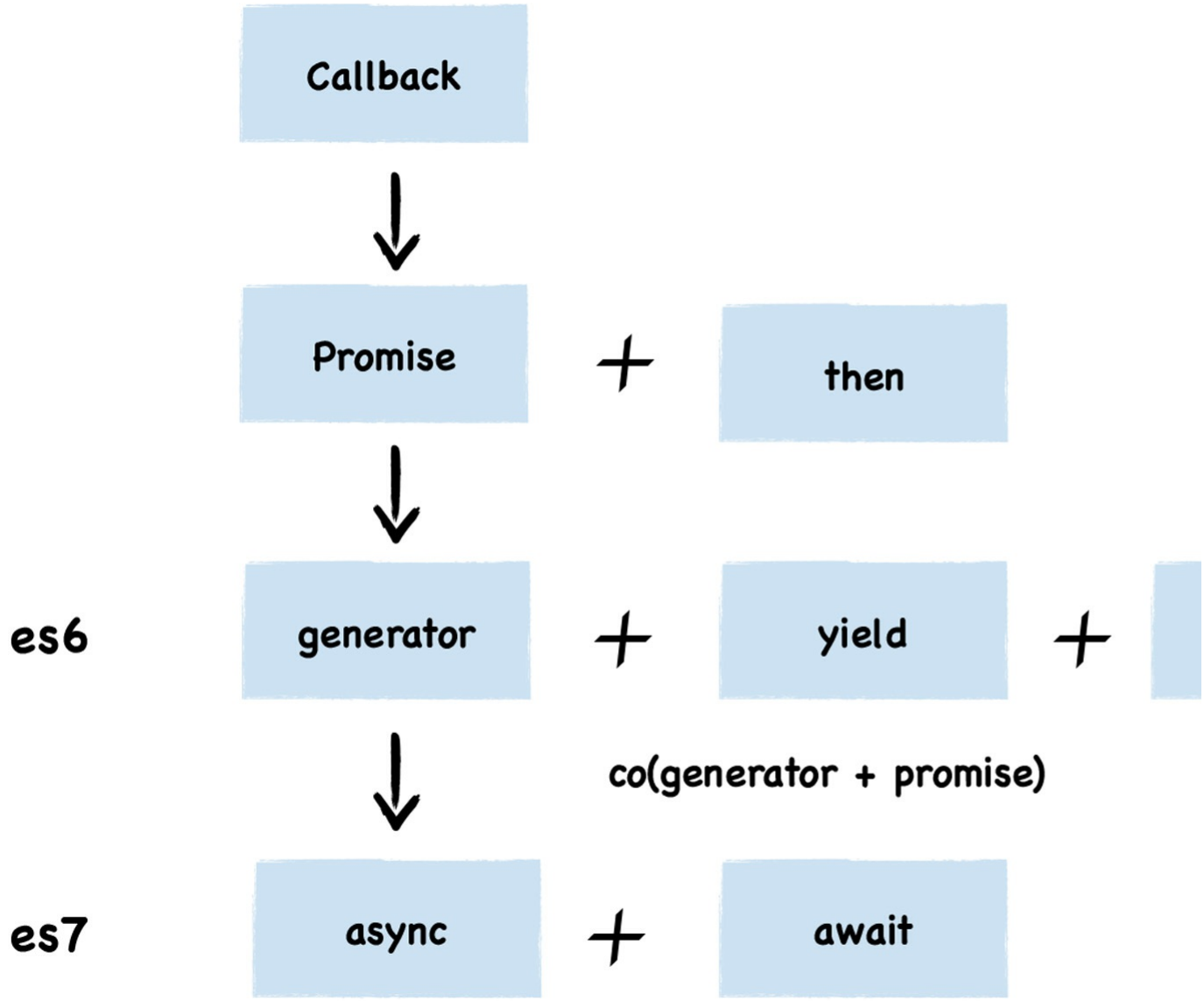
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

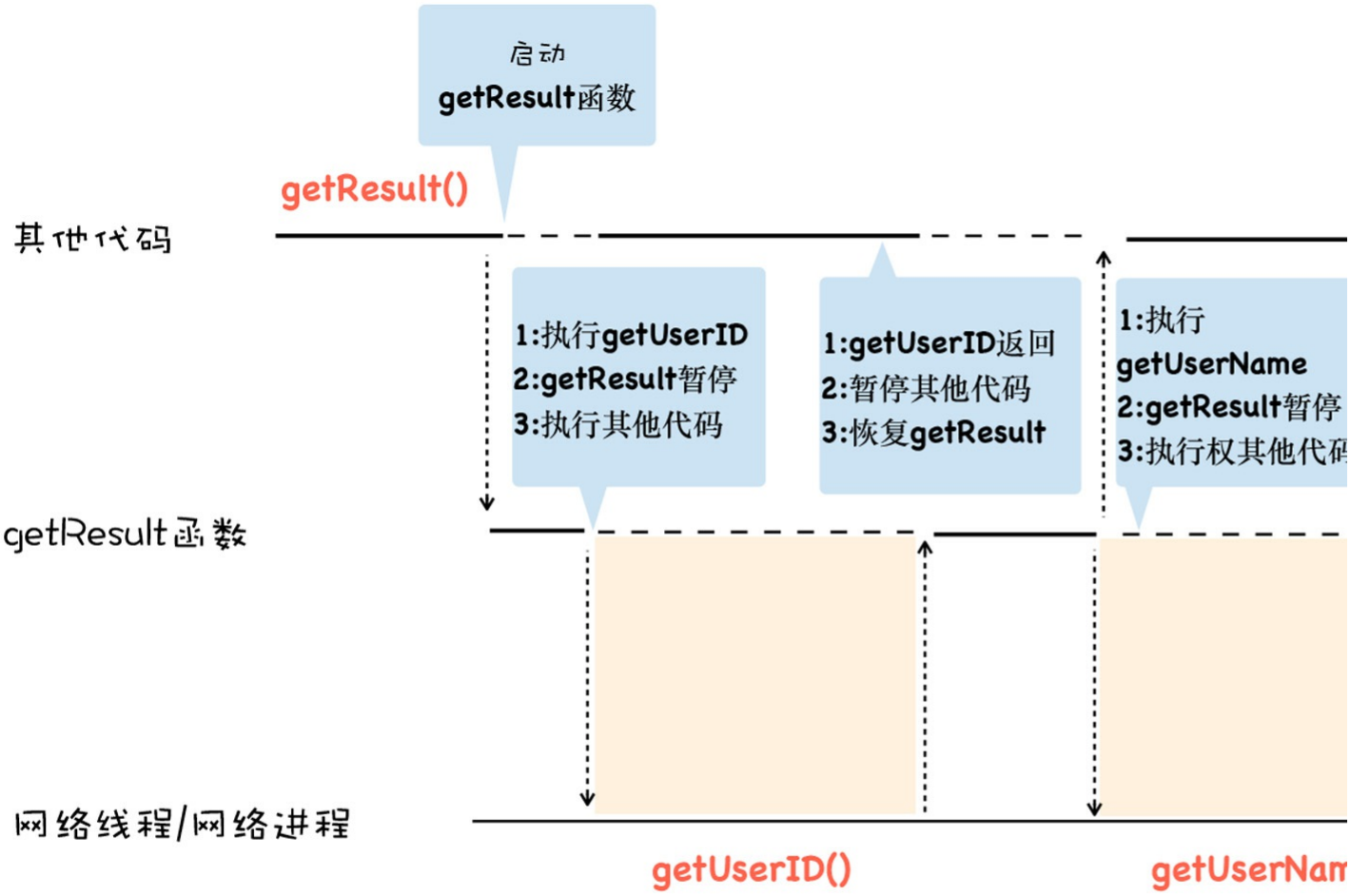
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

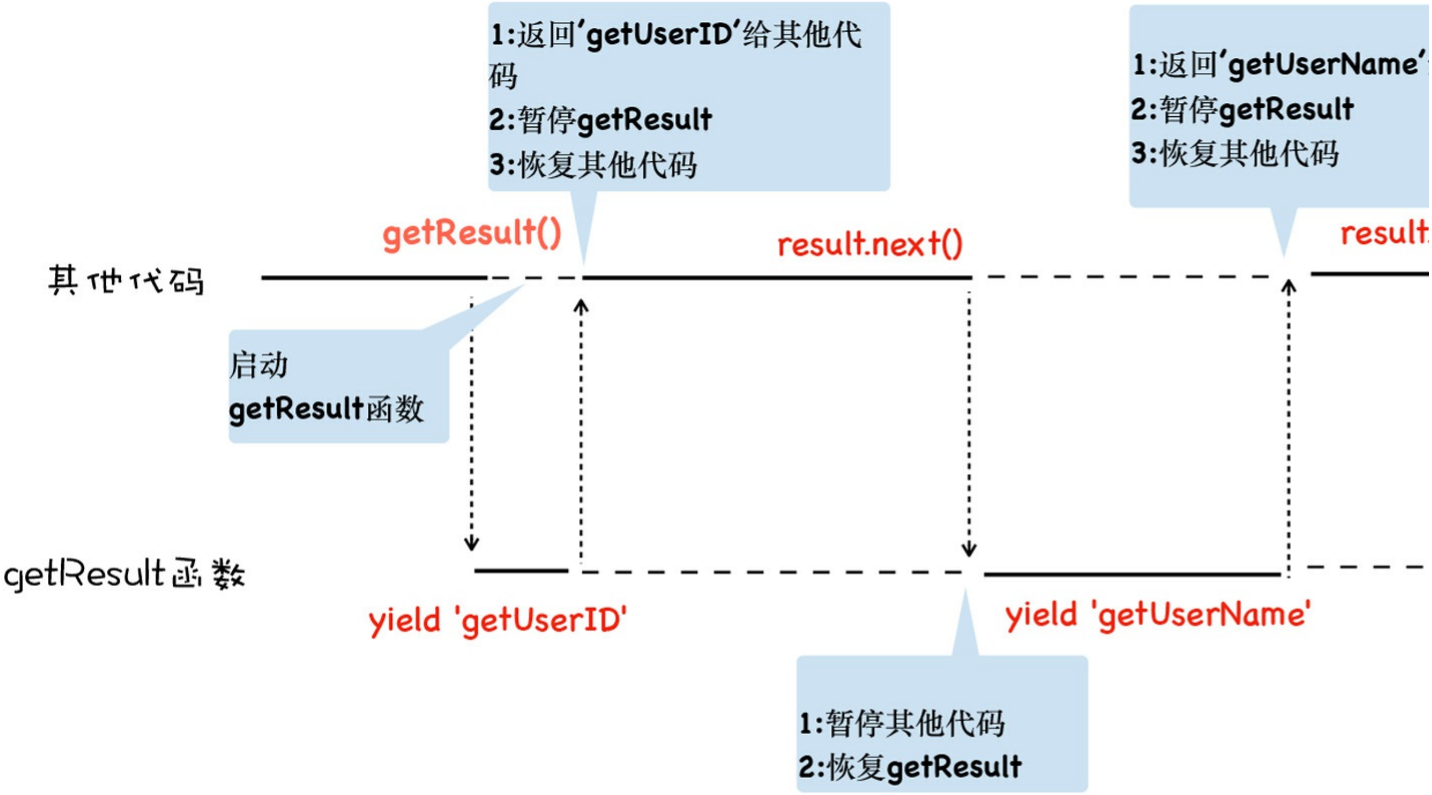
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

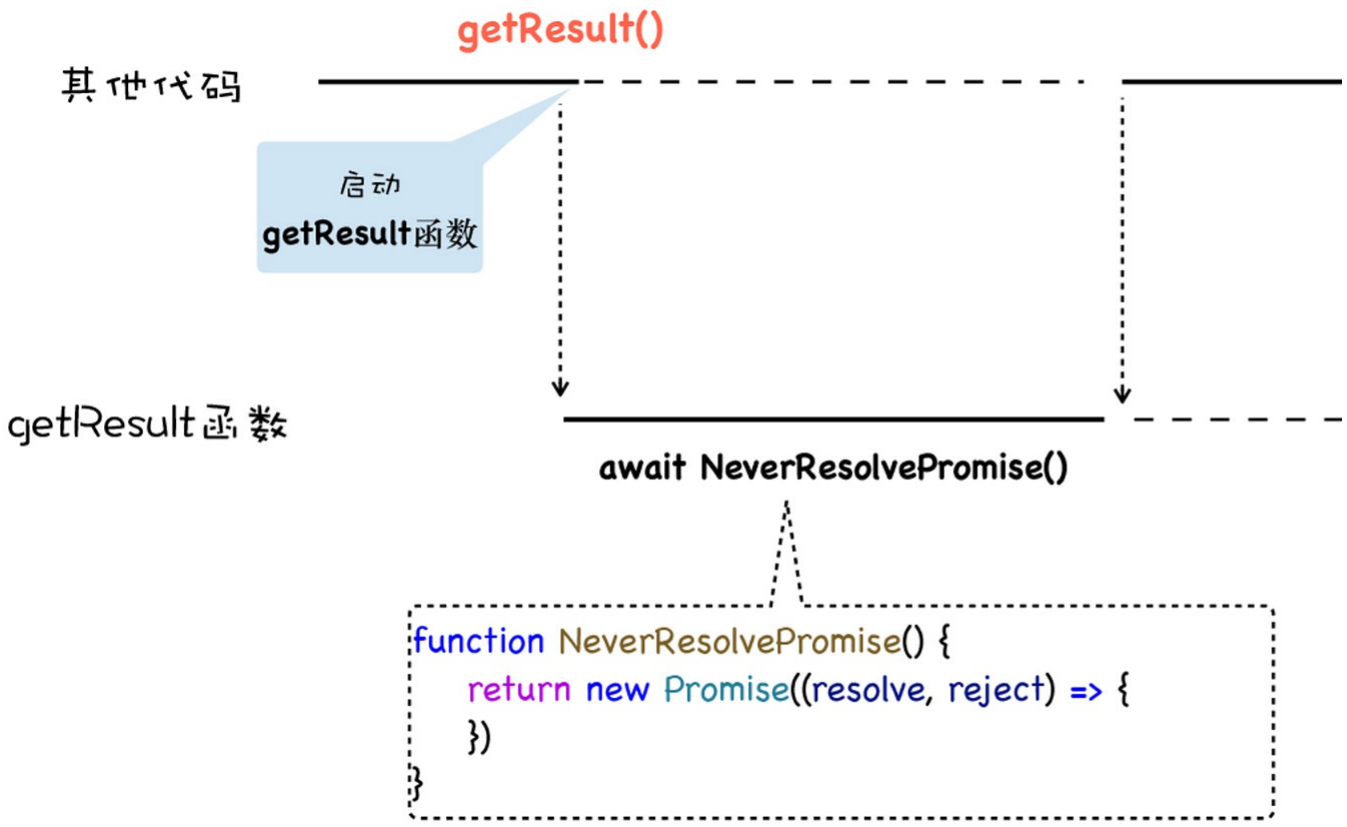
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

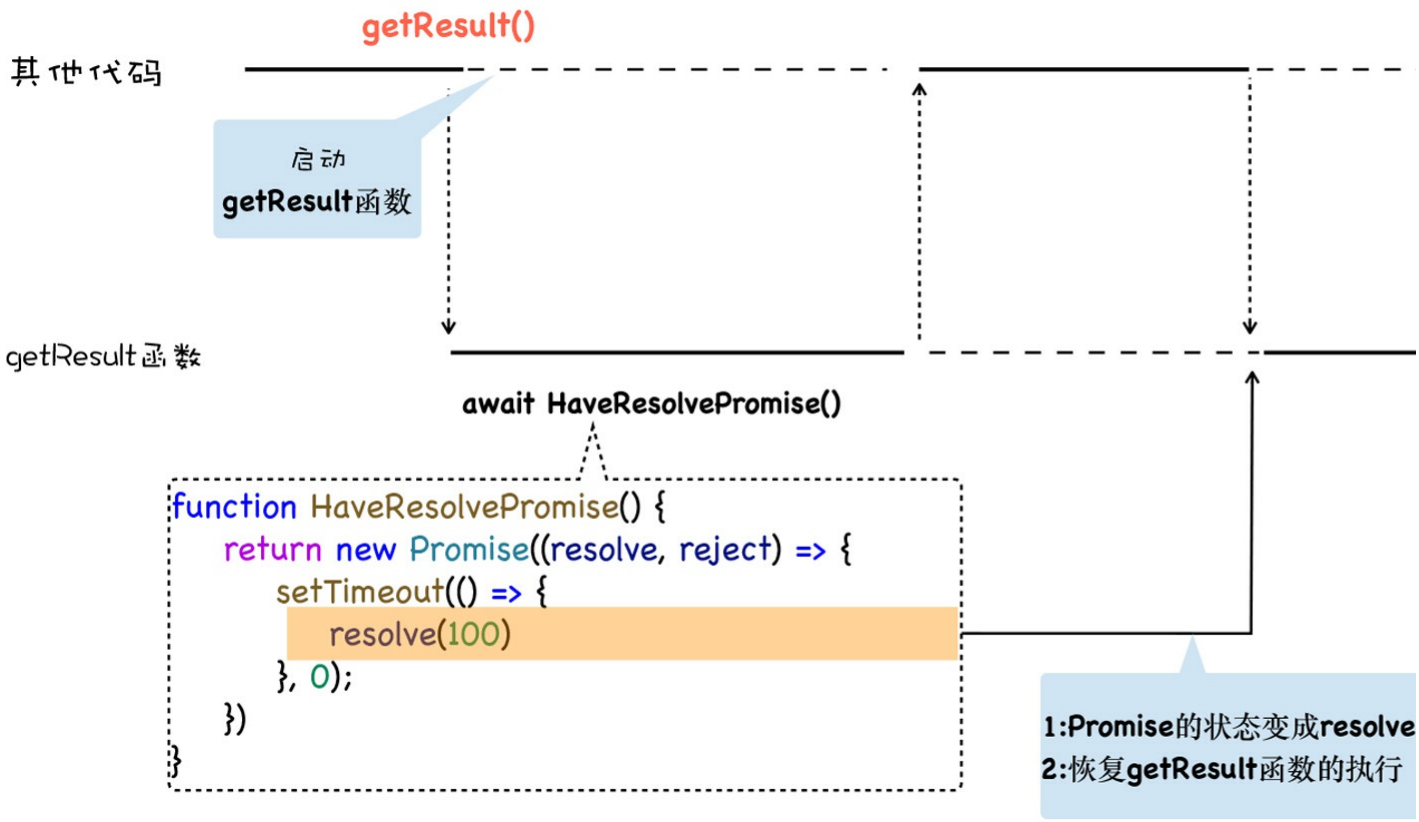


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

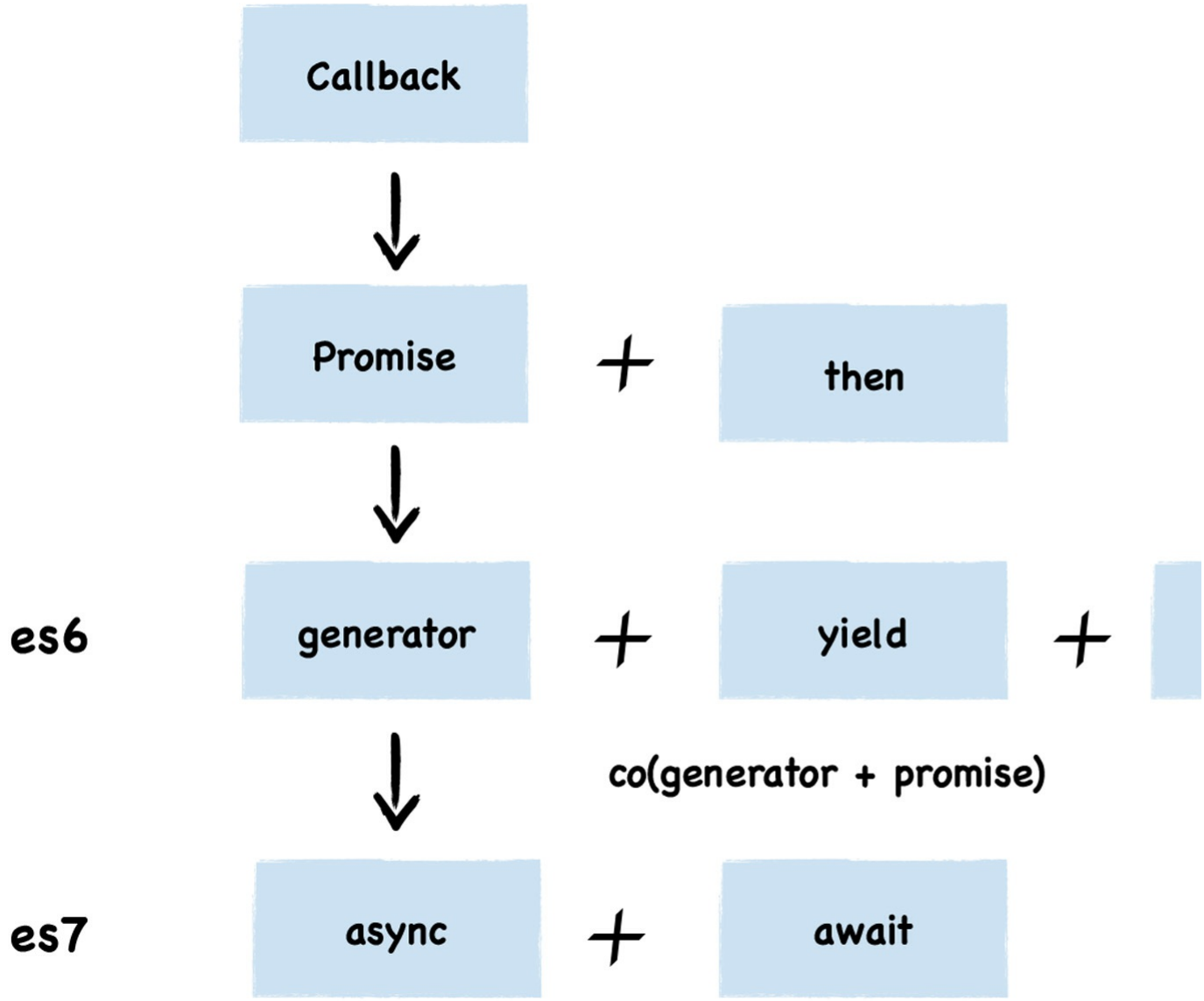
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

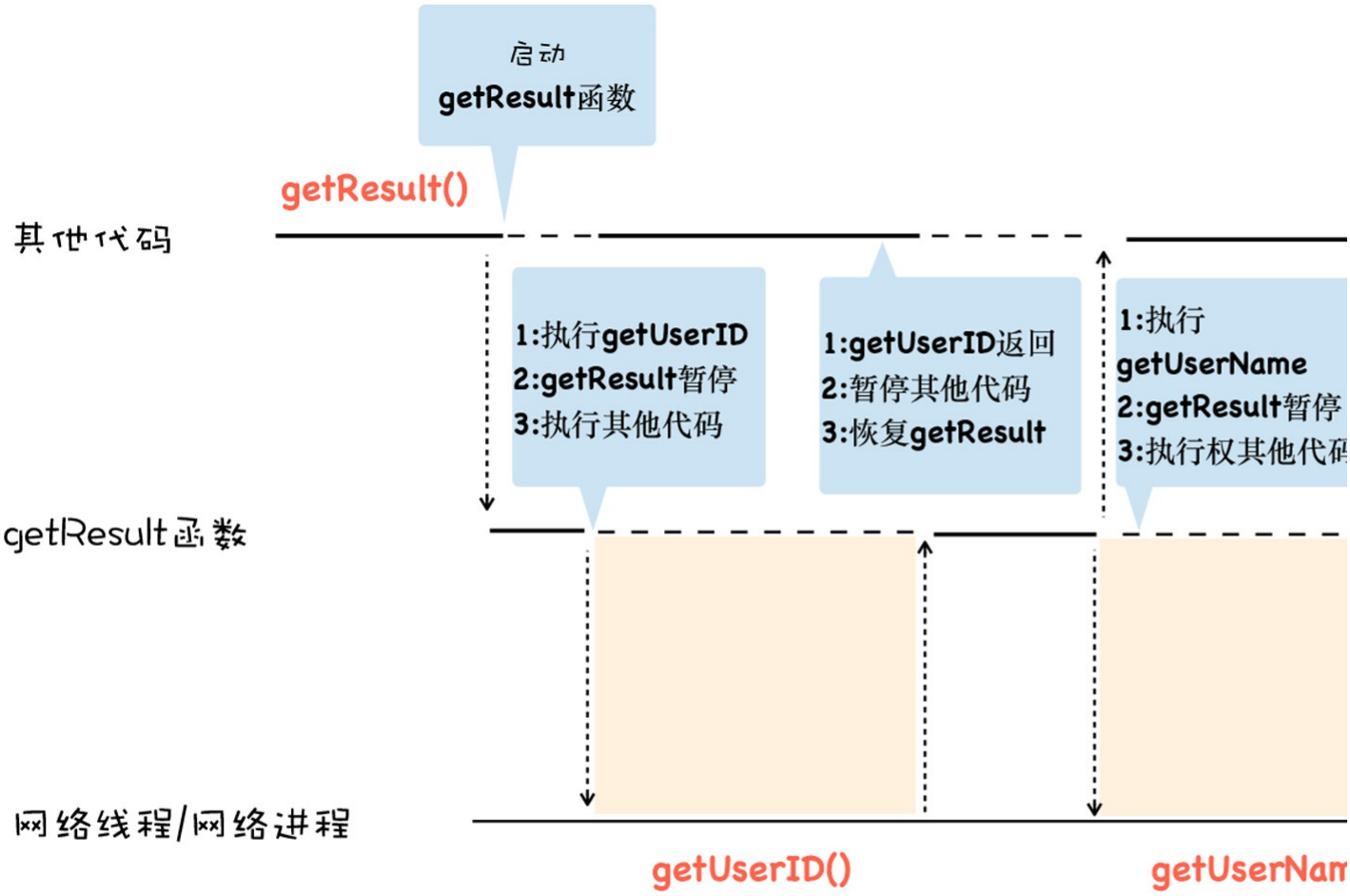
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

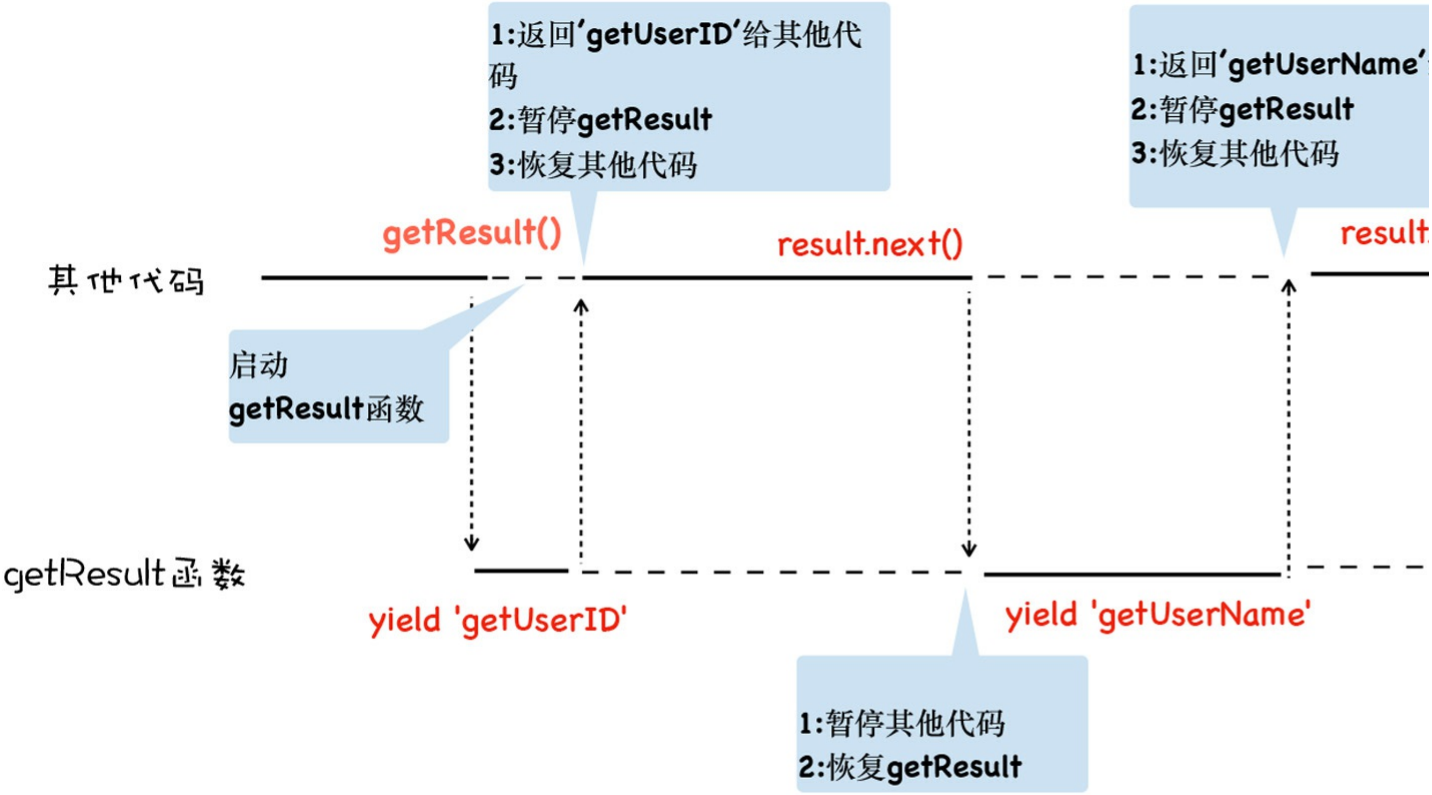
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

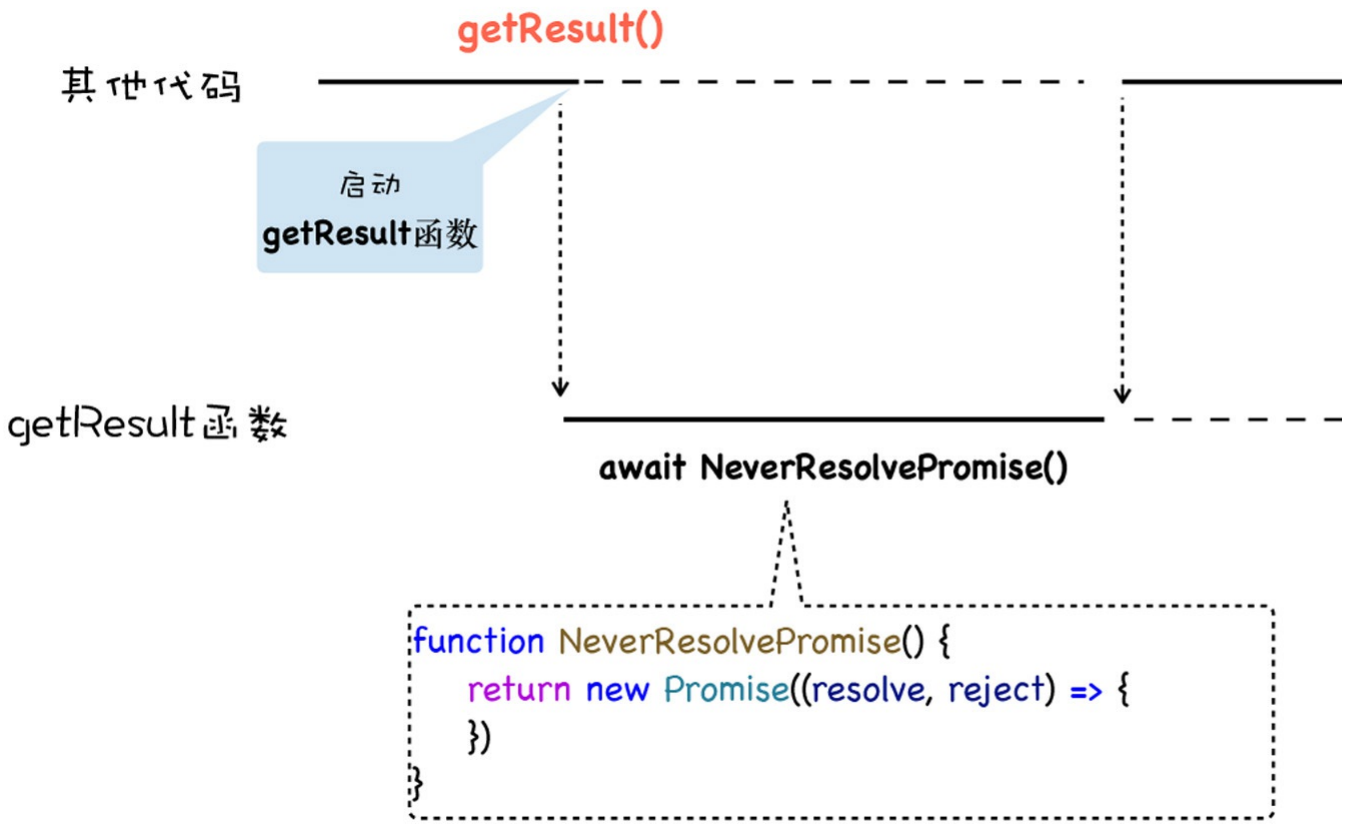
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

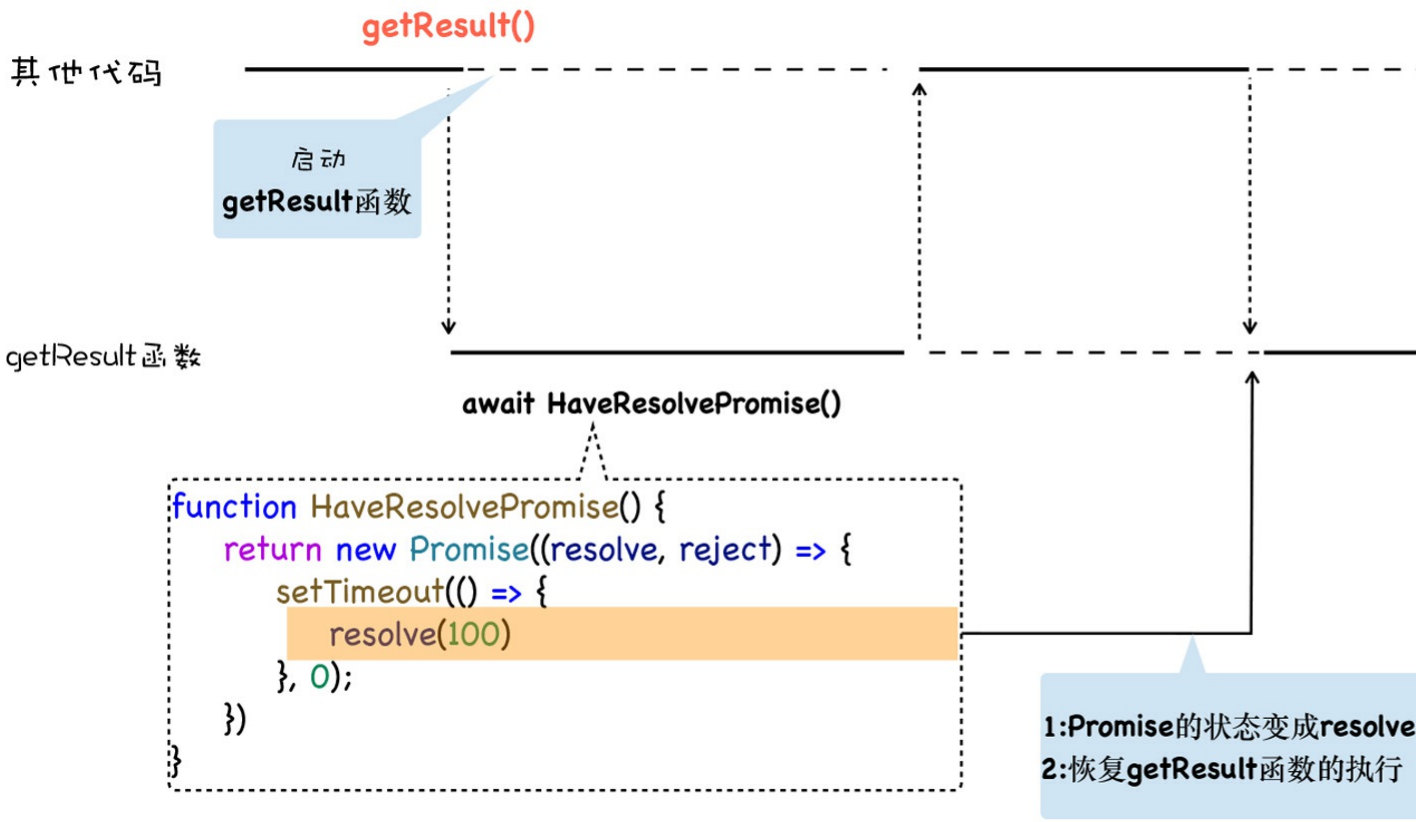


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

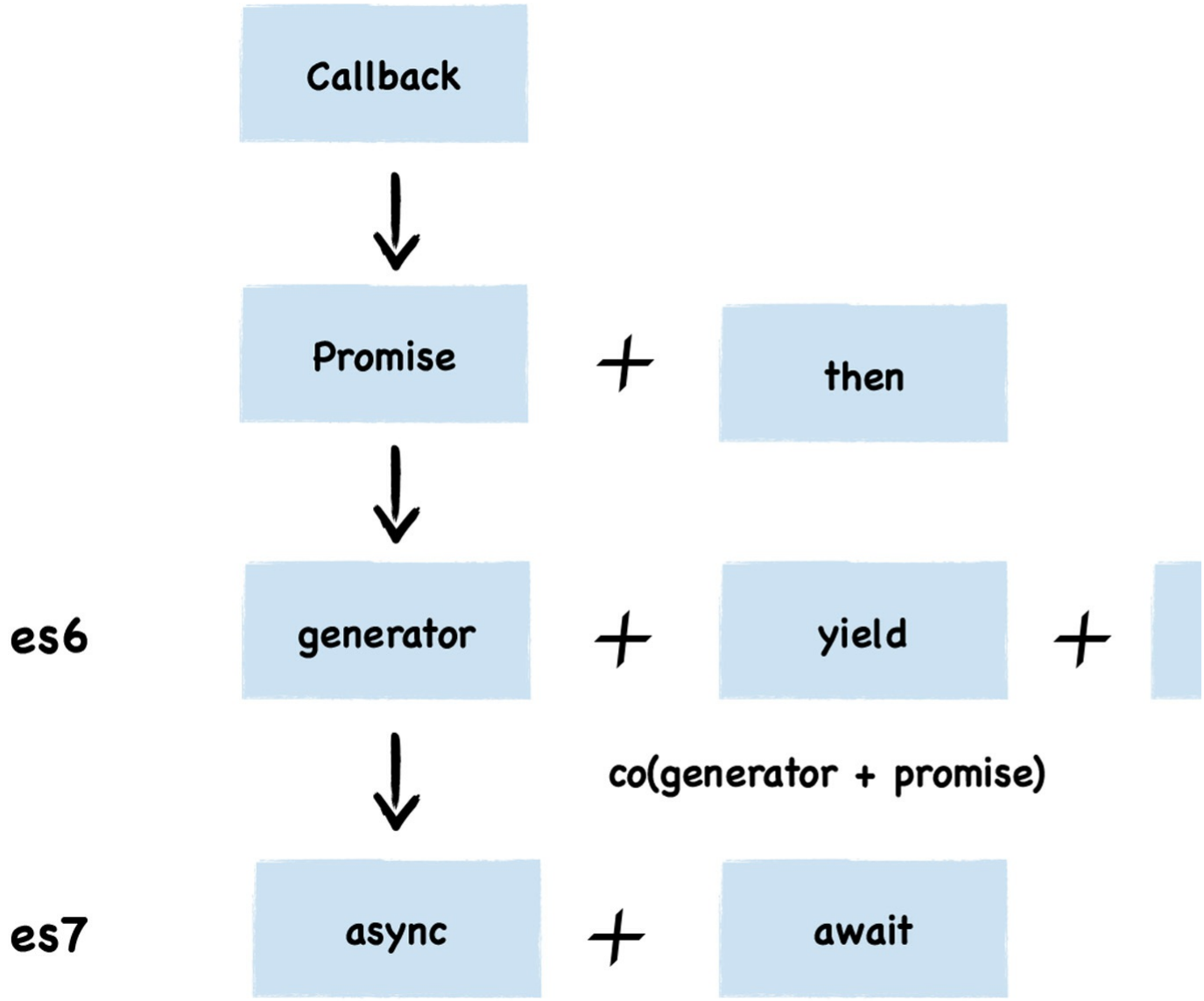
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

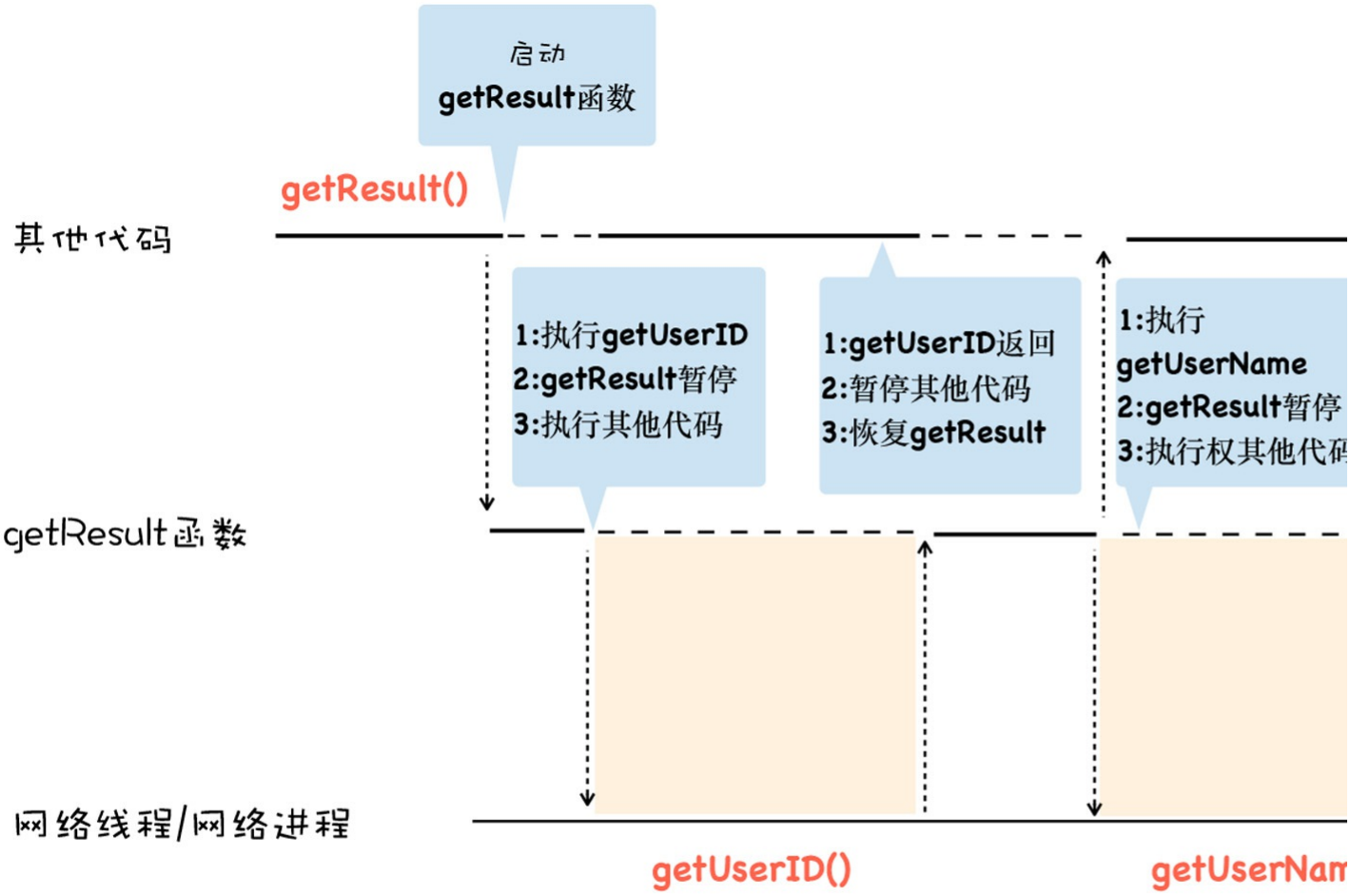
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

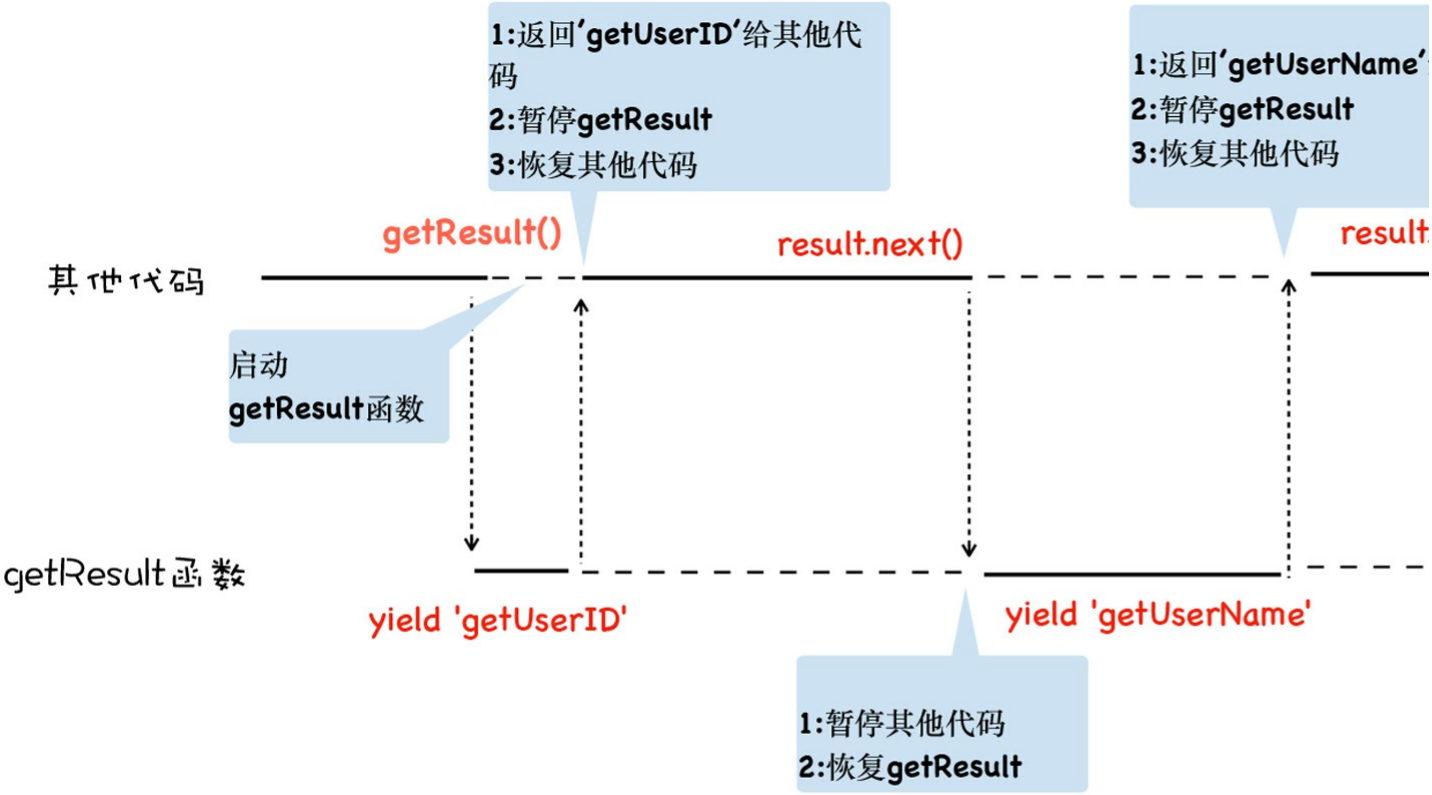
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

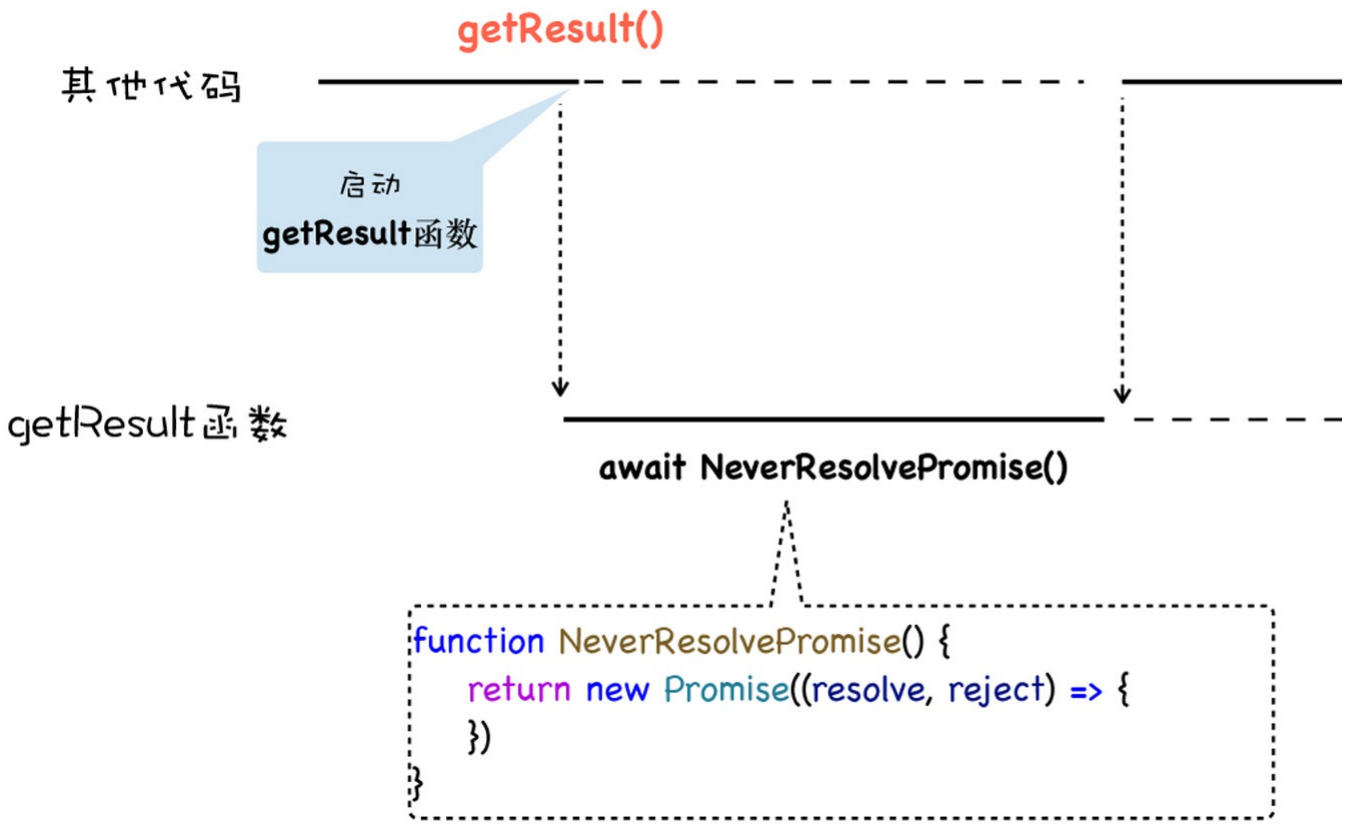
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

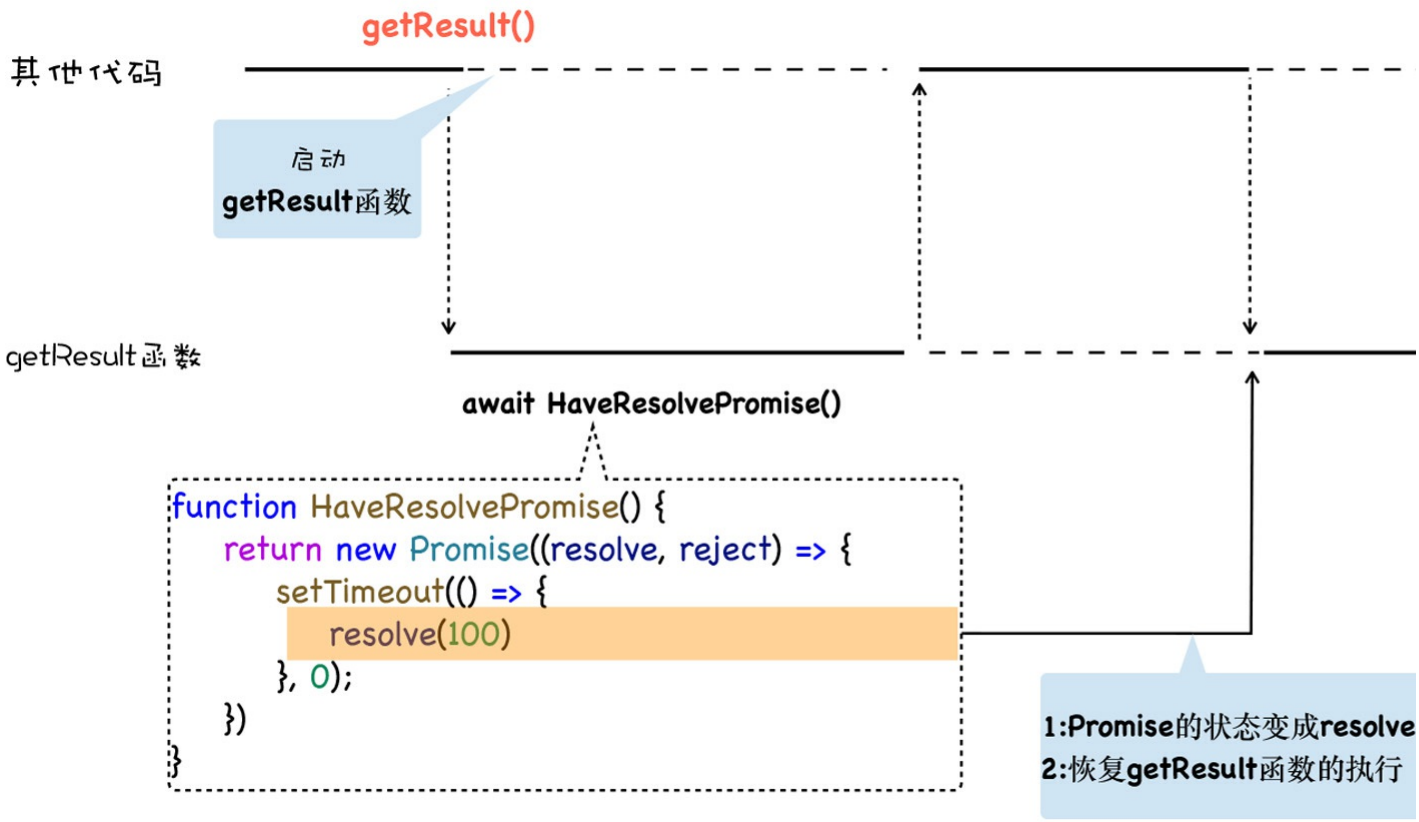


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

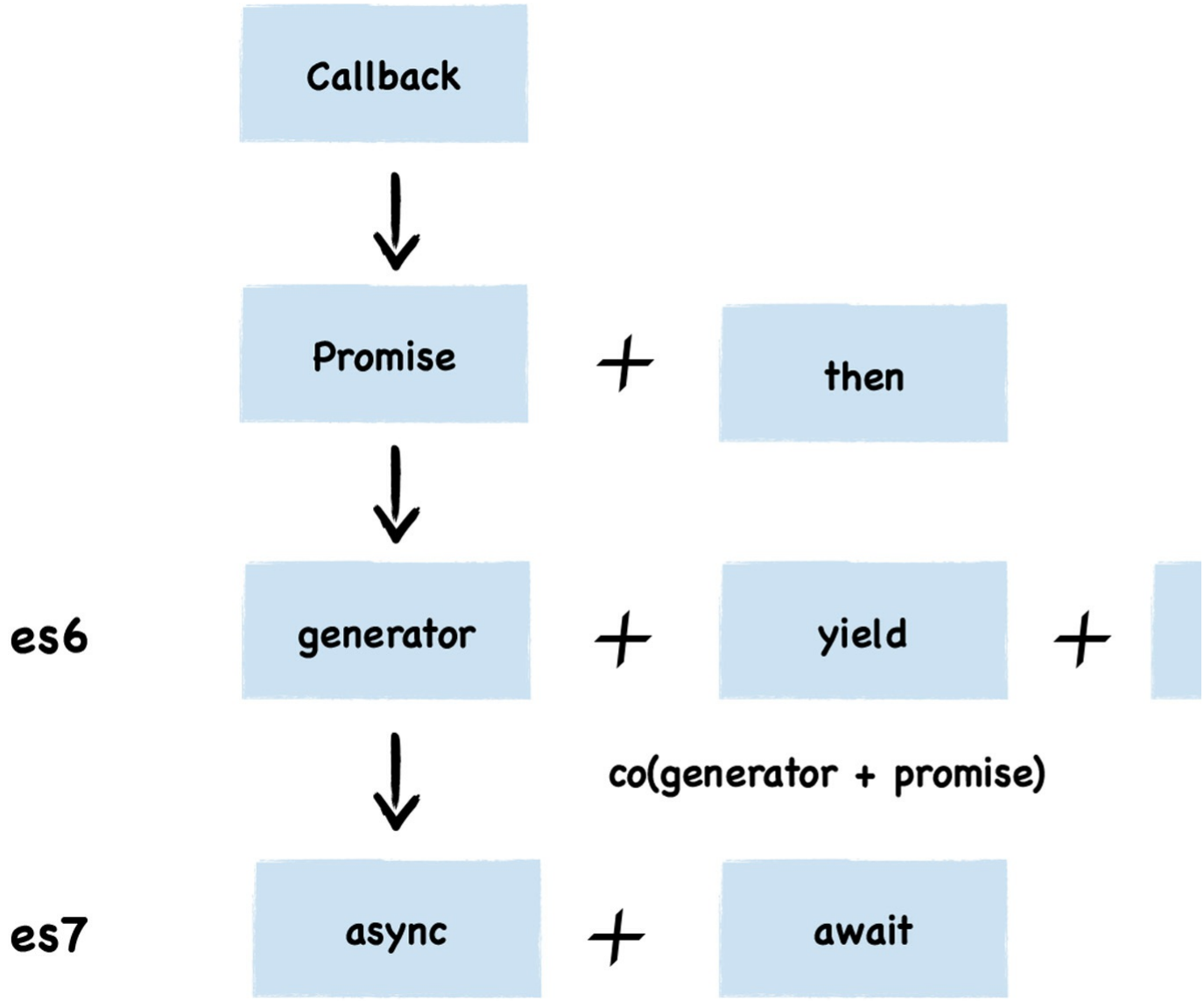
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

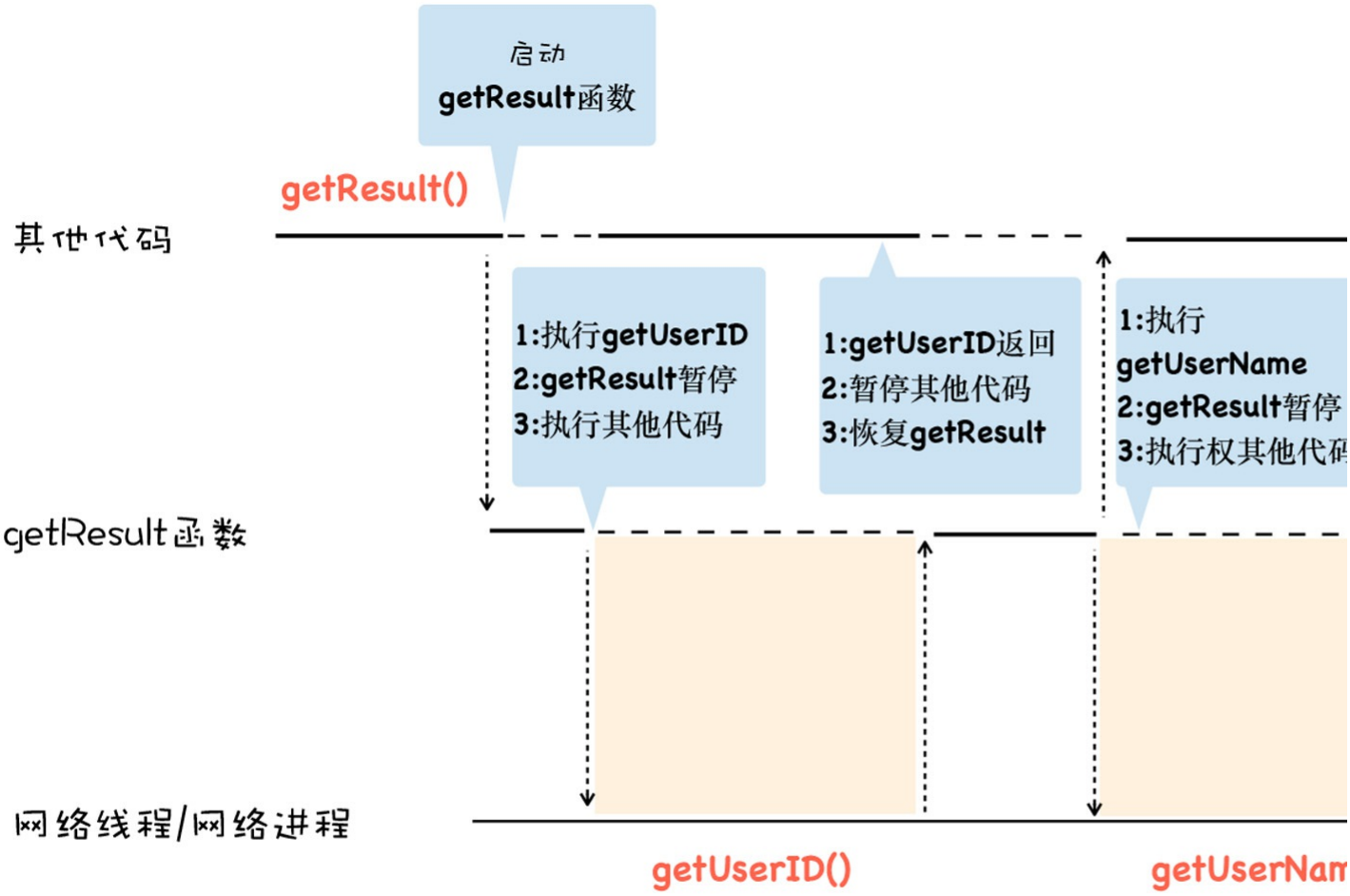
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

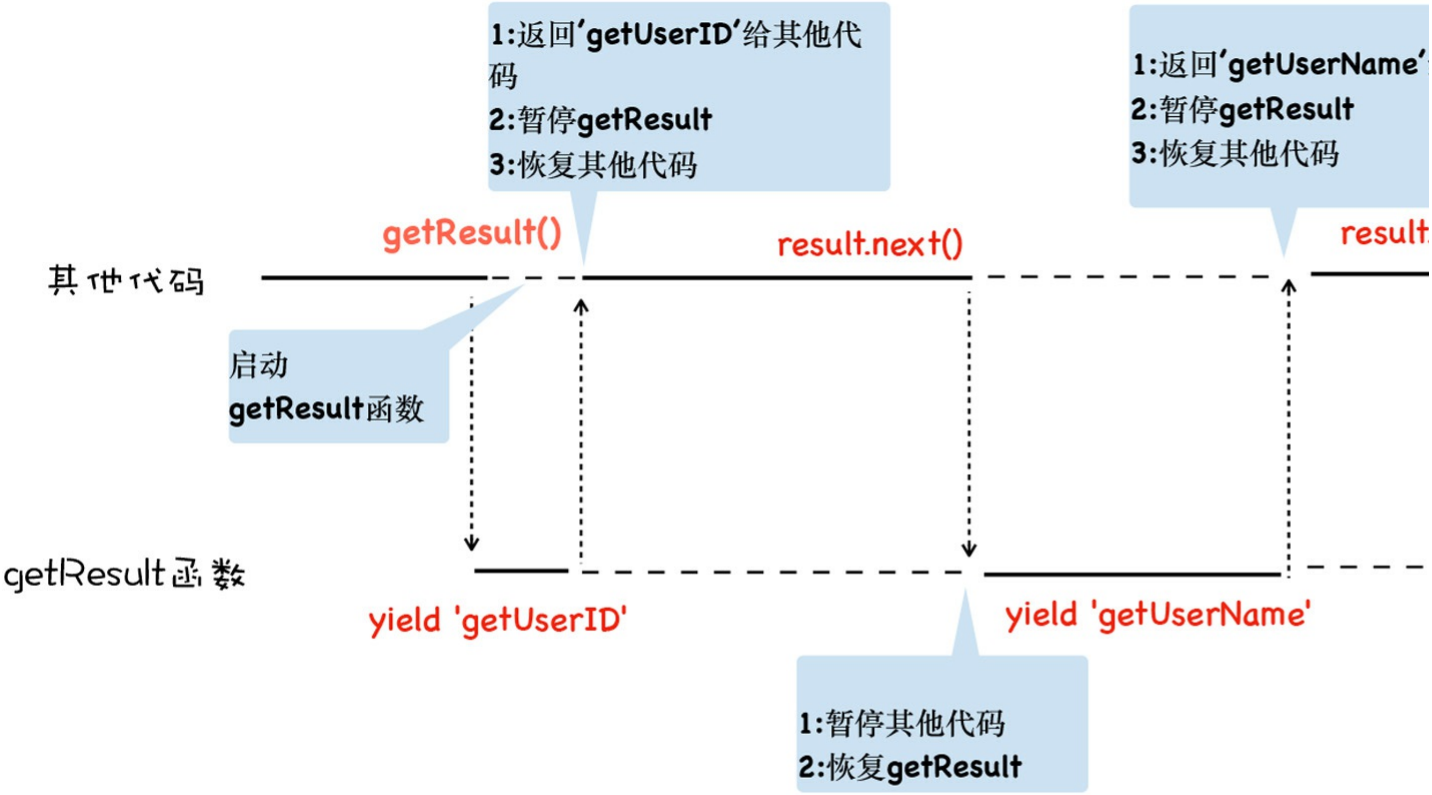
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

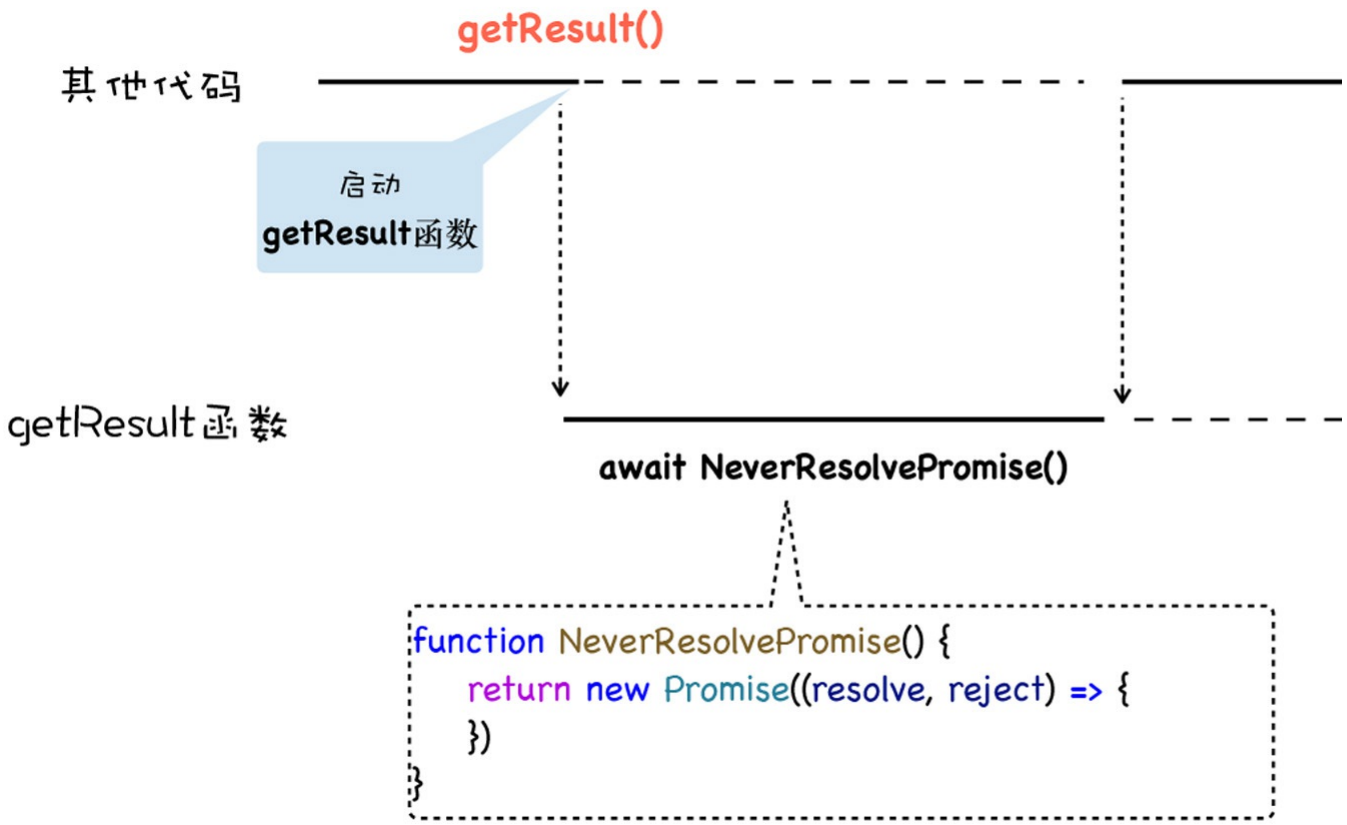
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

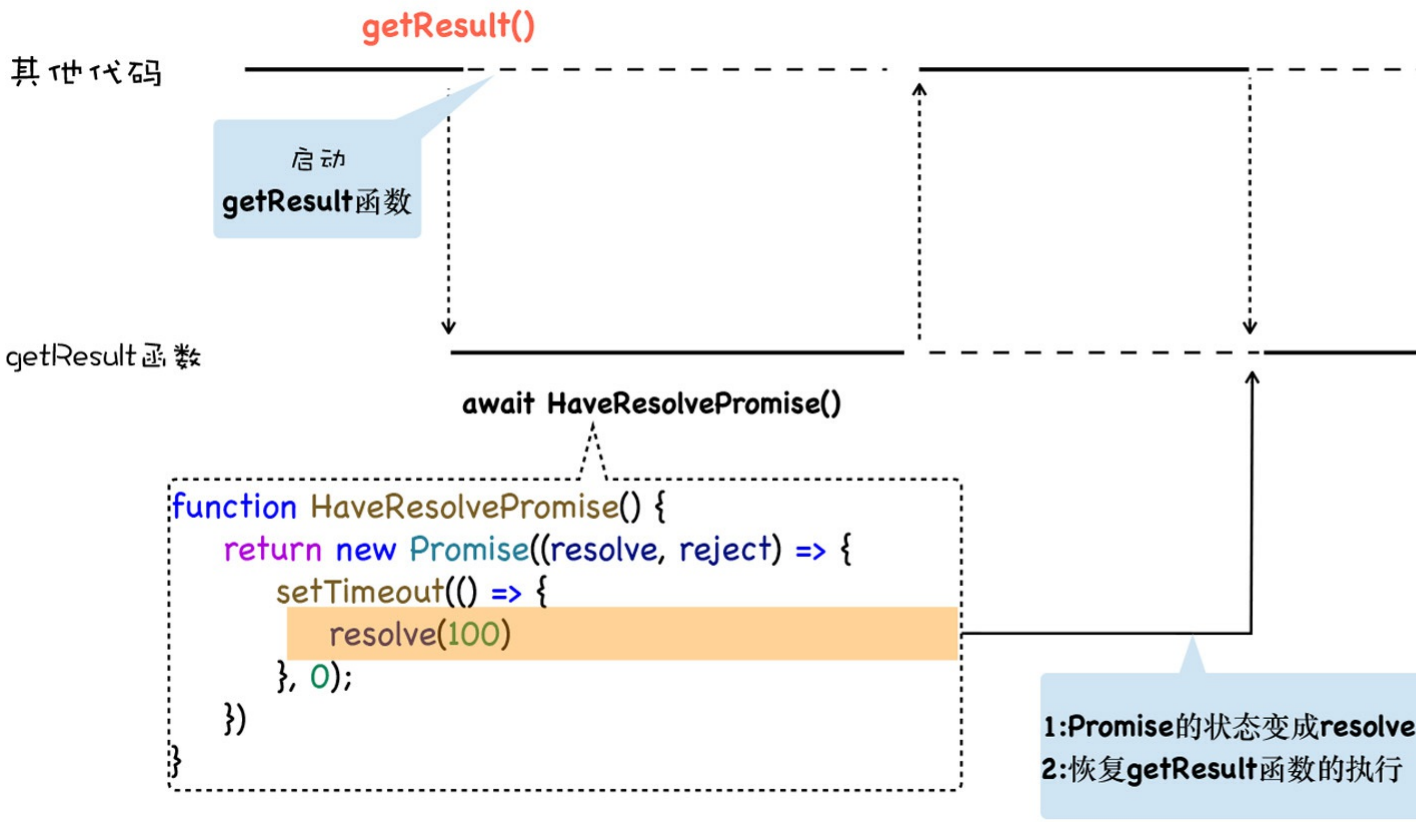


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

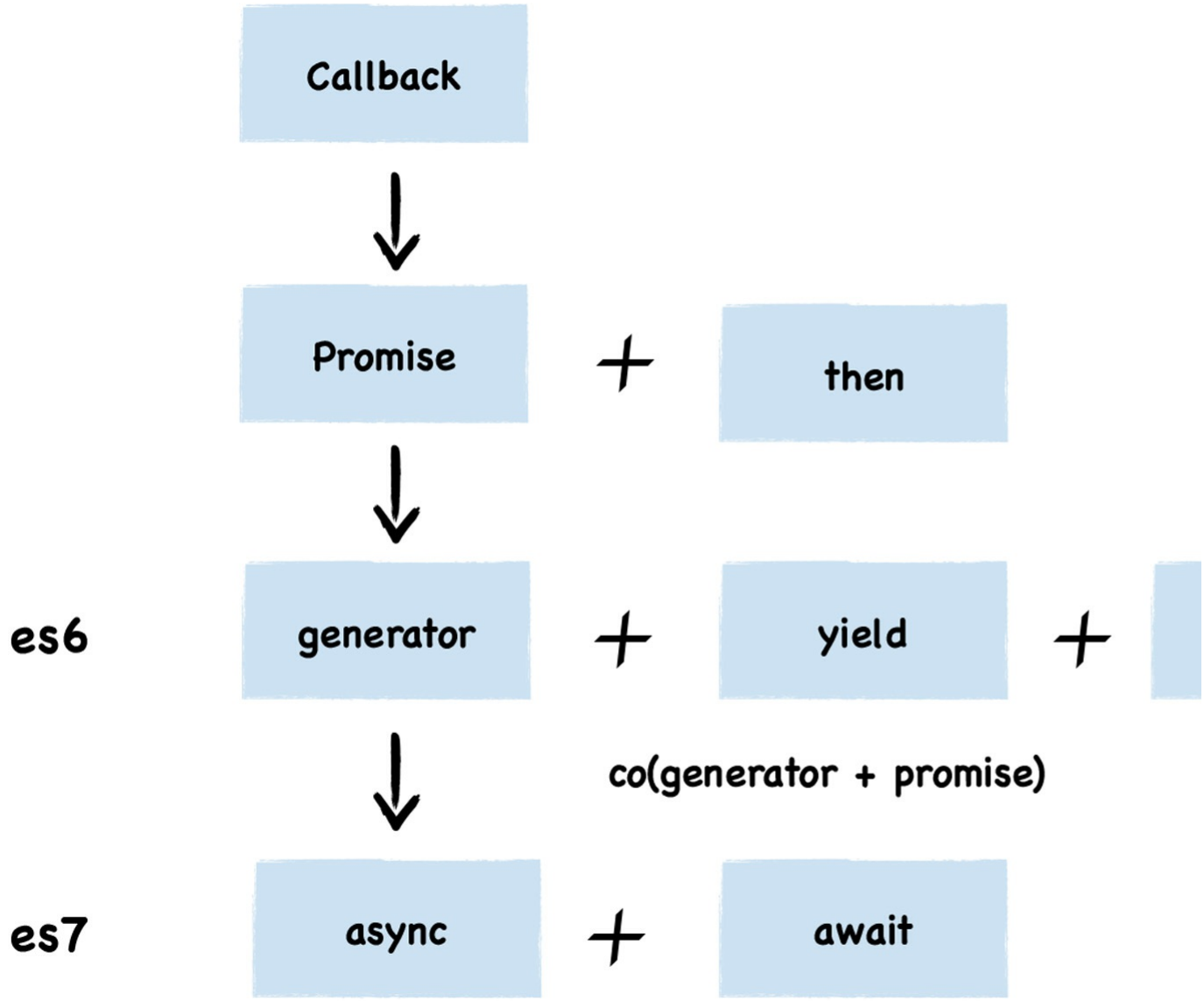
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

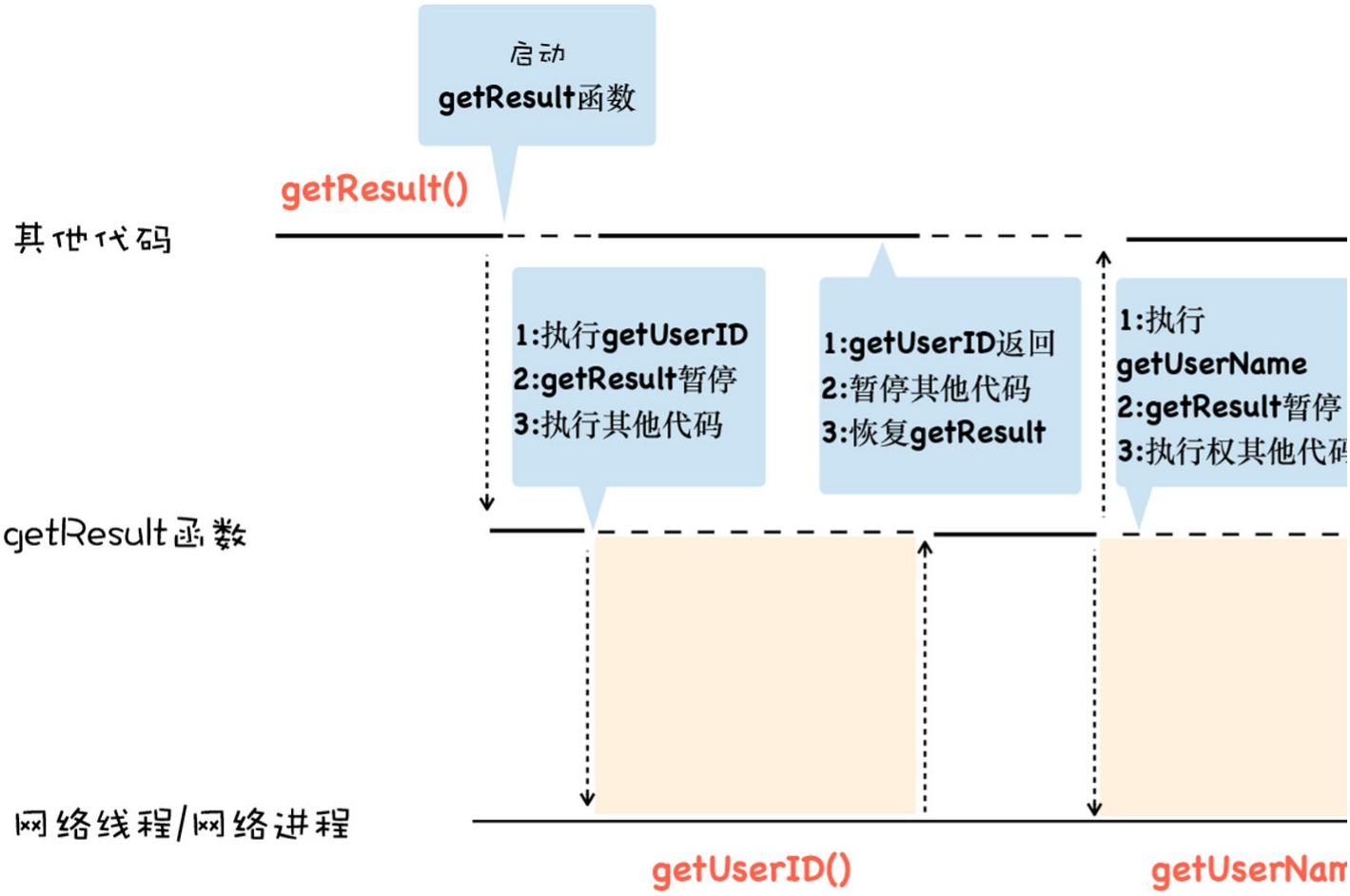
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

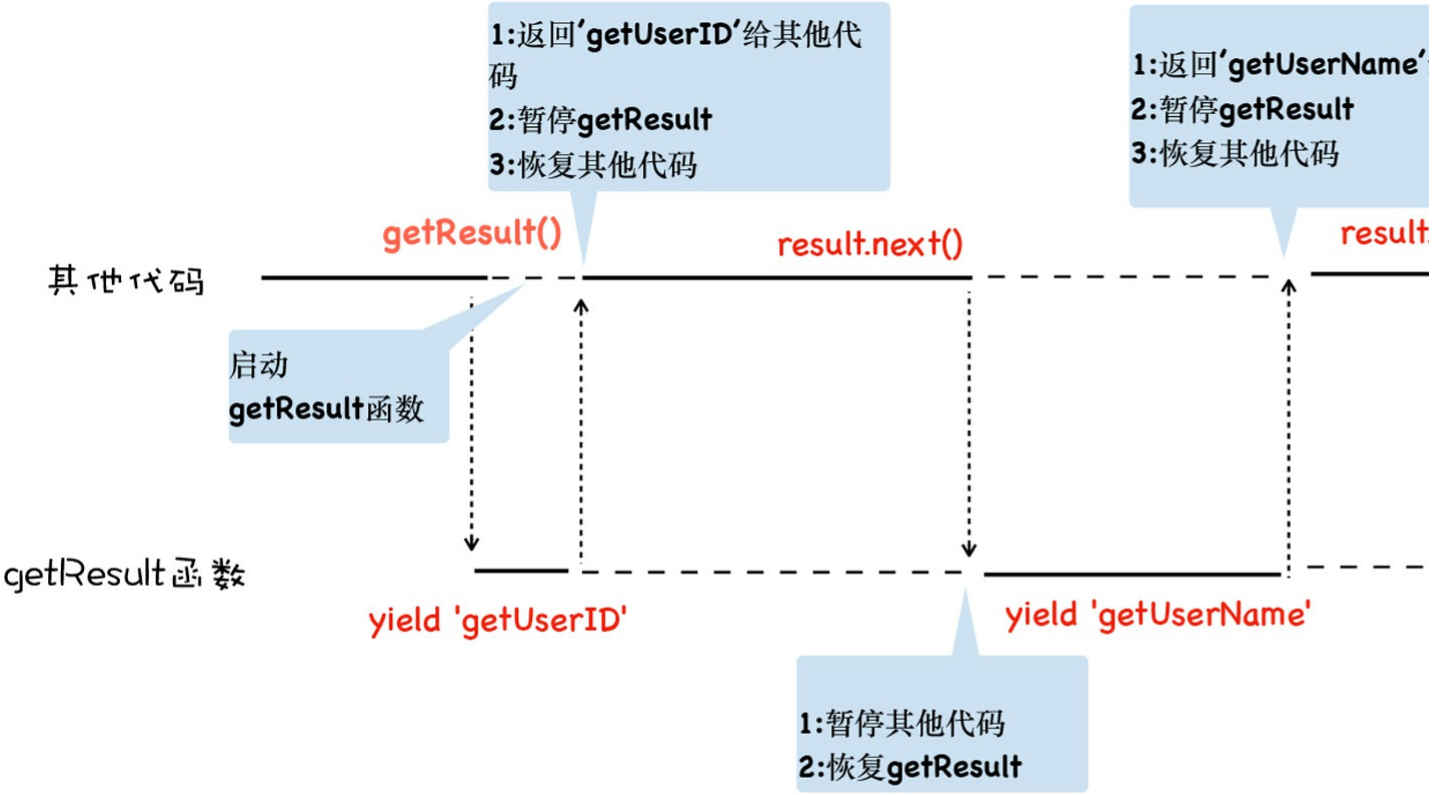
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

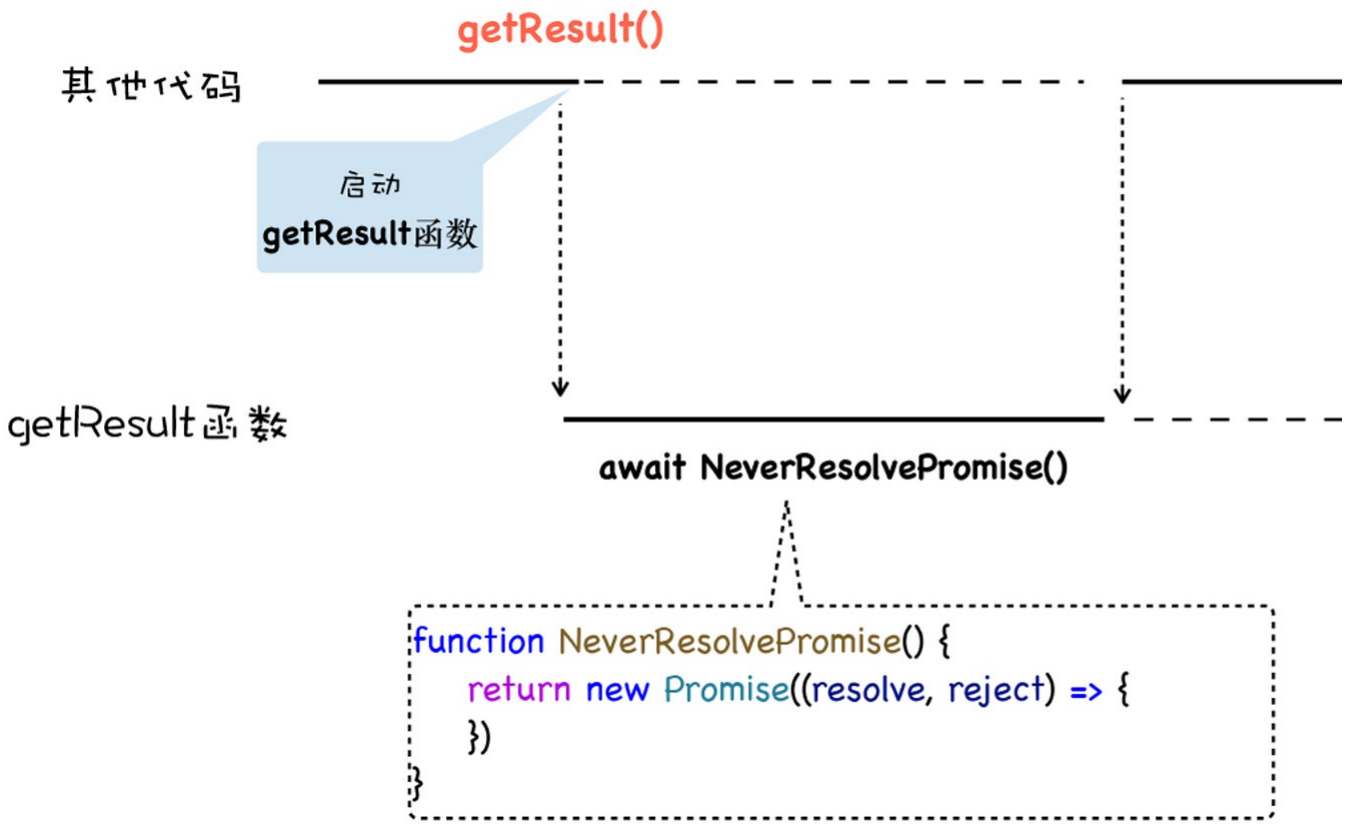
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

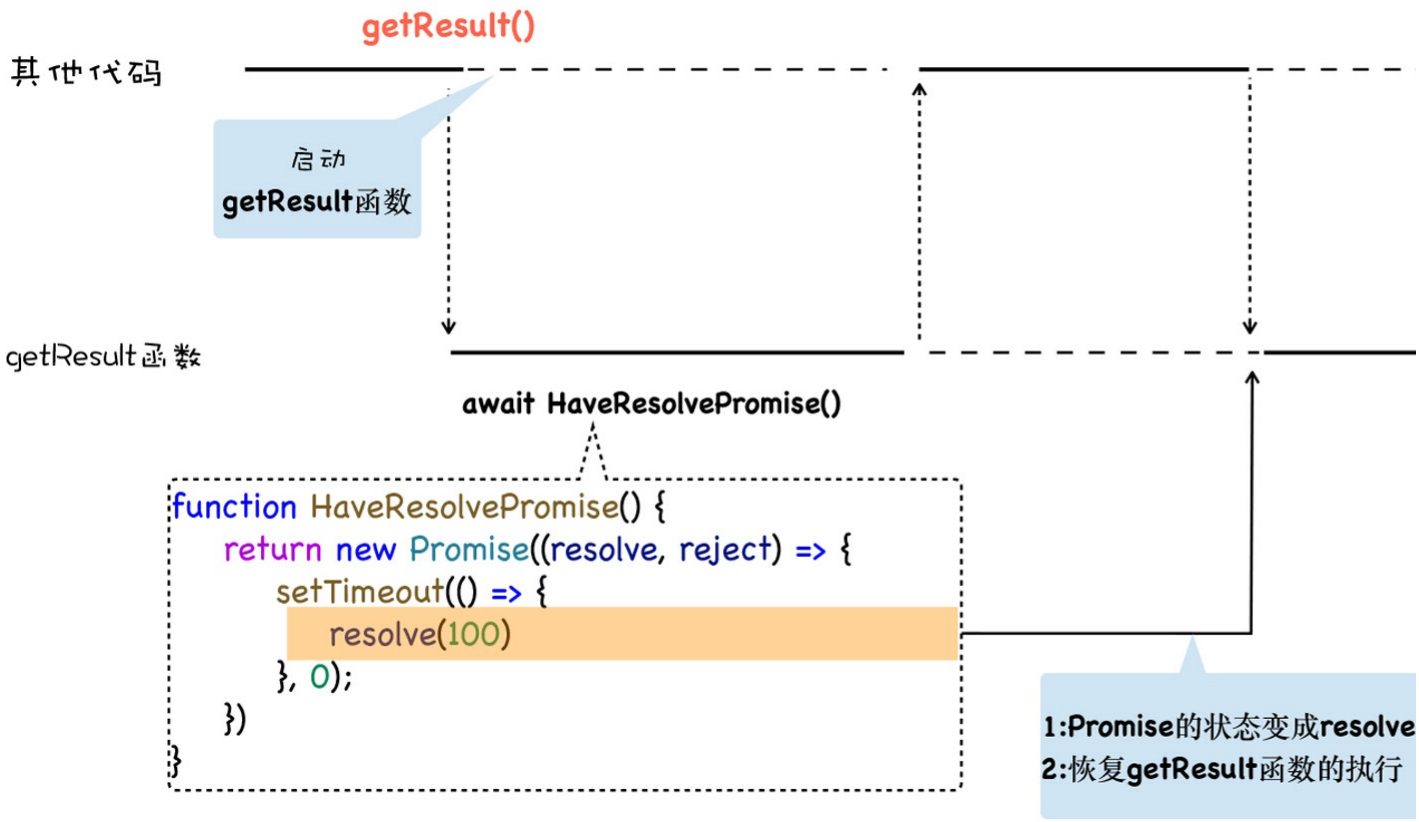


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

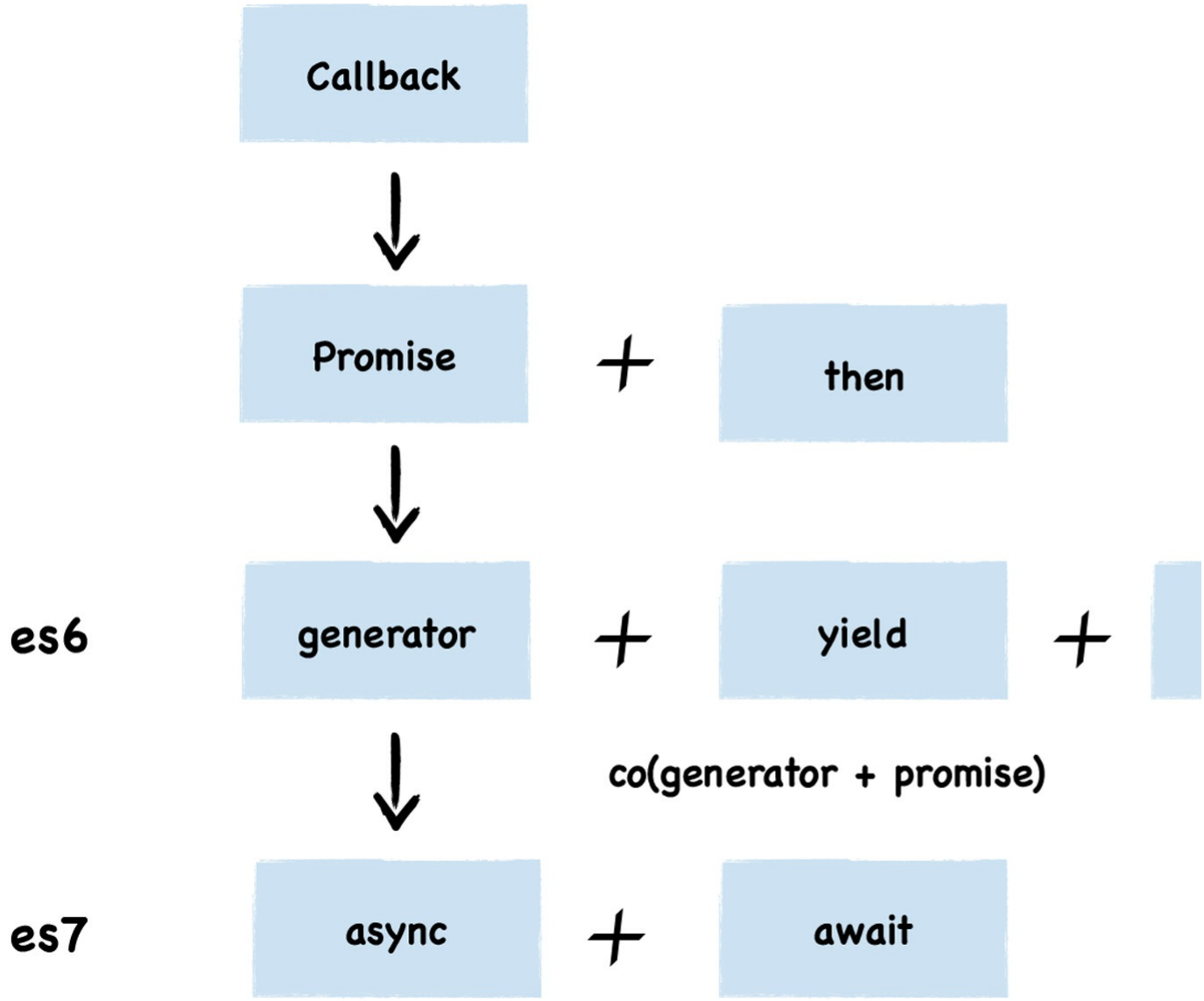
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

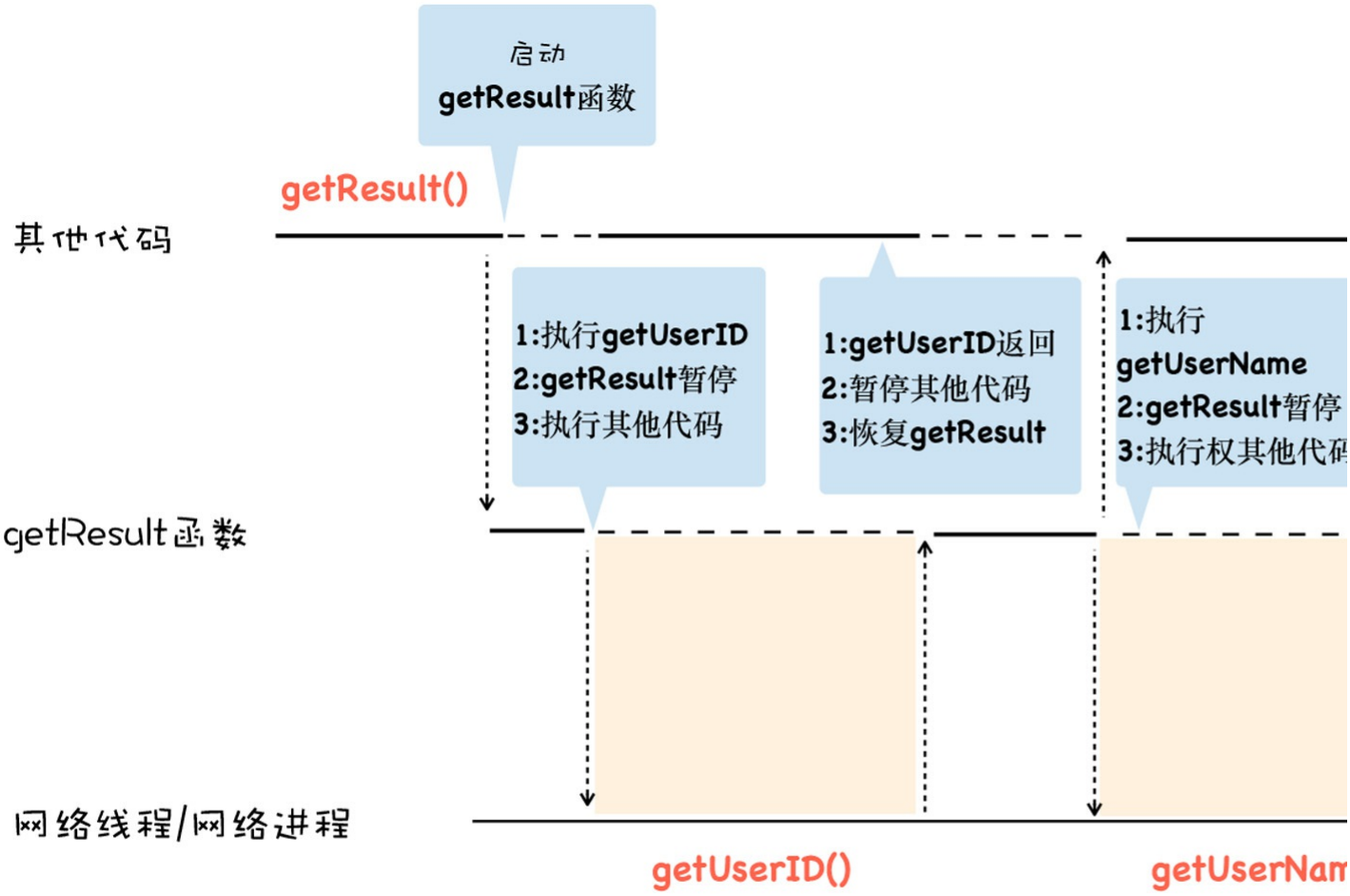
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

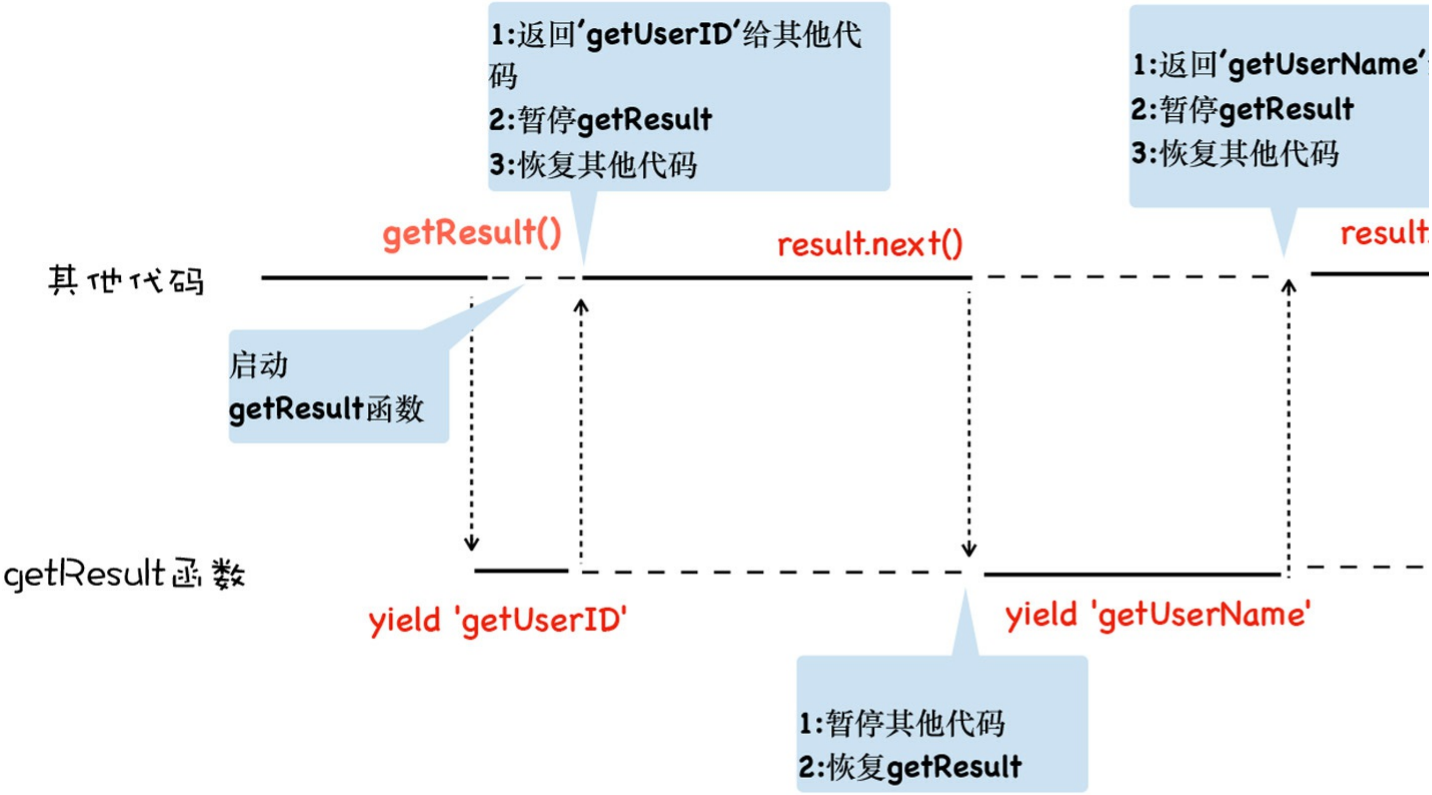
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

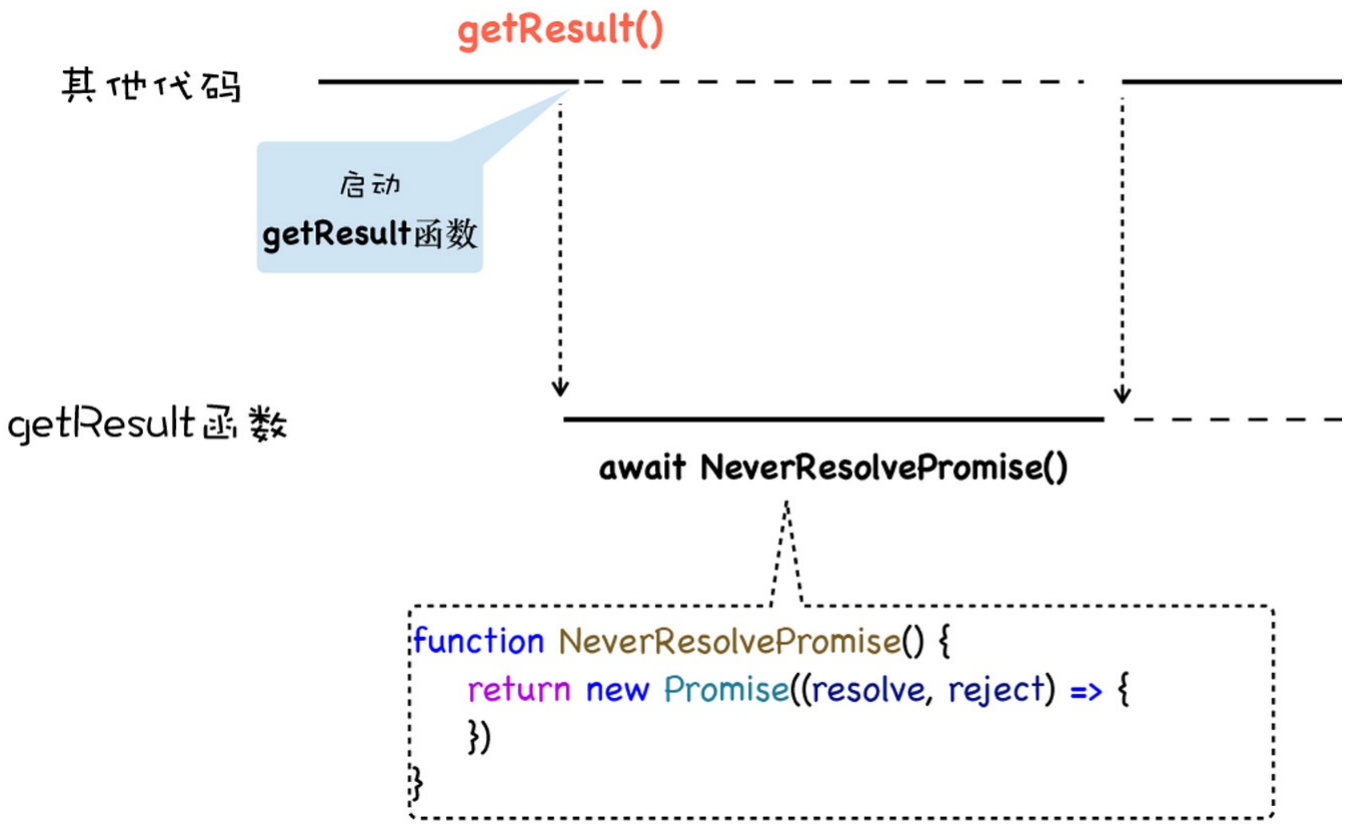
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

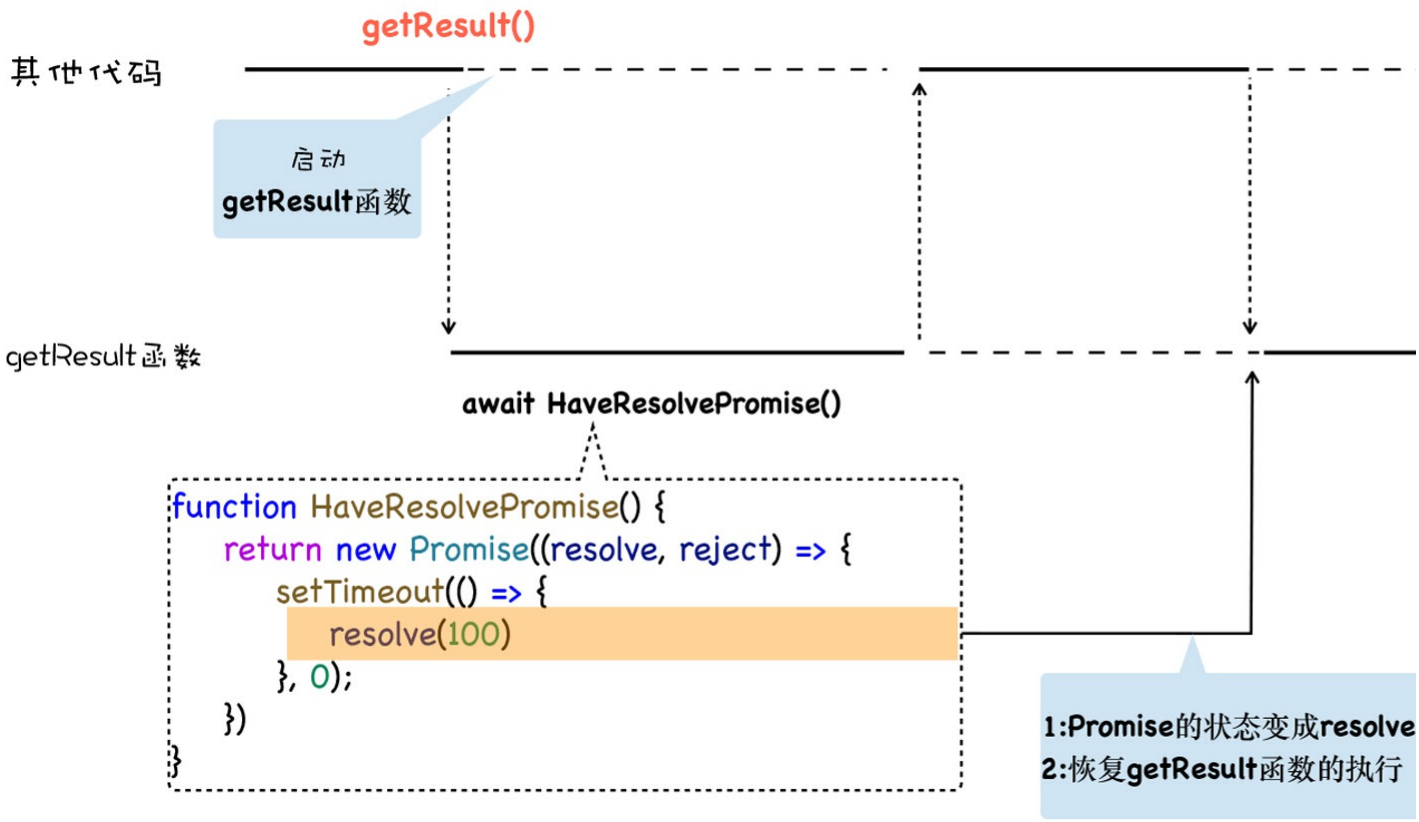


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

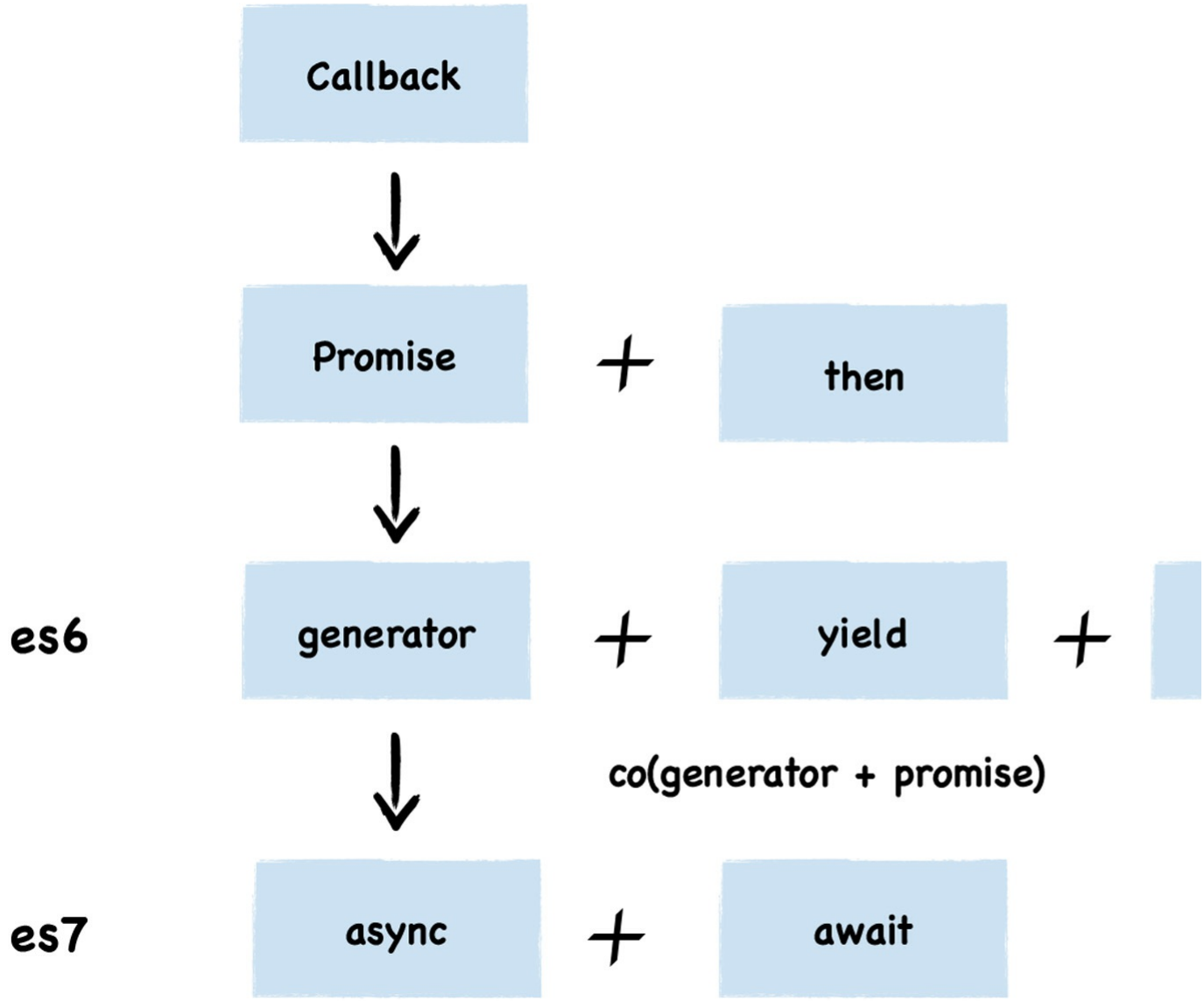
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。

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：

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

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

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：

```
//result_callback：下载结果的回调函数
//url：需要获取URL的内容
function GetUrlContent(result_callback,url) {
    let request = new XMLHttpRequest()

request.open('GET', url)

request.responseType = 'text'

request.onload = function () {
```

```
result_callback(request.response)

}

request.send()

}

function IDCallback(id) {

console.log(id)

let new_name_url = name_url + "?id="+id

GetUrlContent(NameCallback,new_name_url)

}

function NameCallback(name) {

console.log(name)

}

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代码逻辑的不连贯、不线性，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：

```
fetch(id_url)

.then((response) => {

return response.text()

})

.then((response) => {

let new_name_url = name_url + "?id=" + response

return fetch(new_name_url)

}).then((response) => {

return response.text()

}).then((response) => {

console.log(response)//输出最终的结果

})
```

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不线性的问题。

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不线性的问题，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。

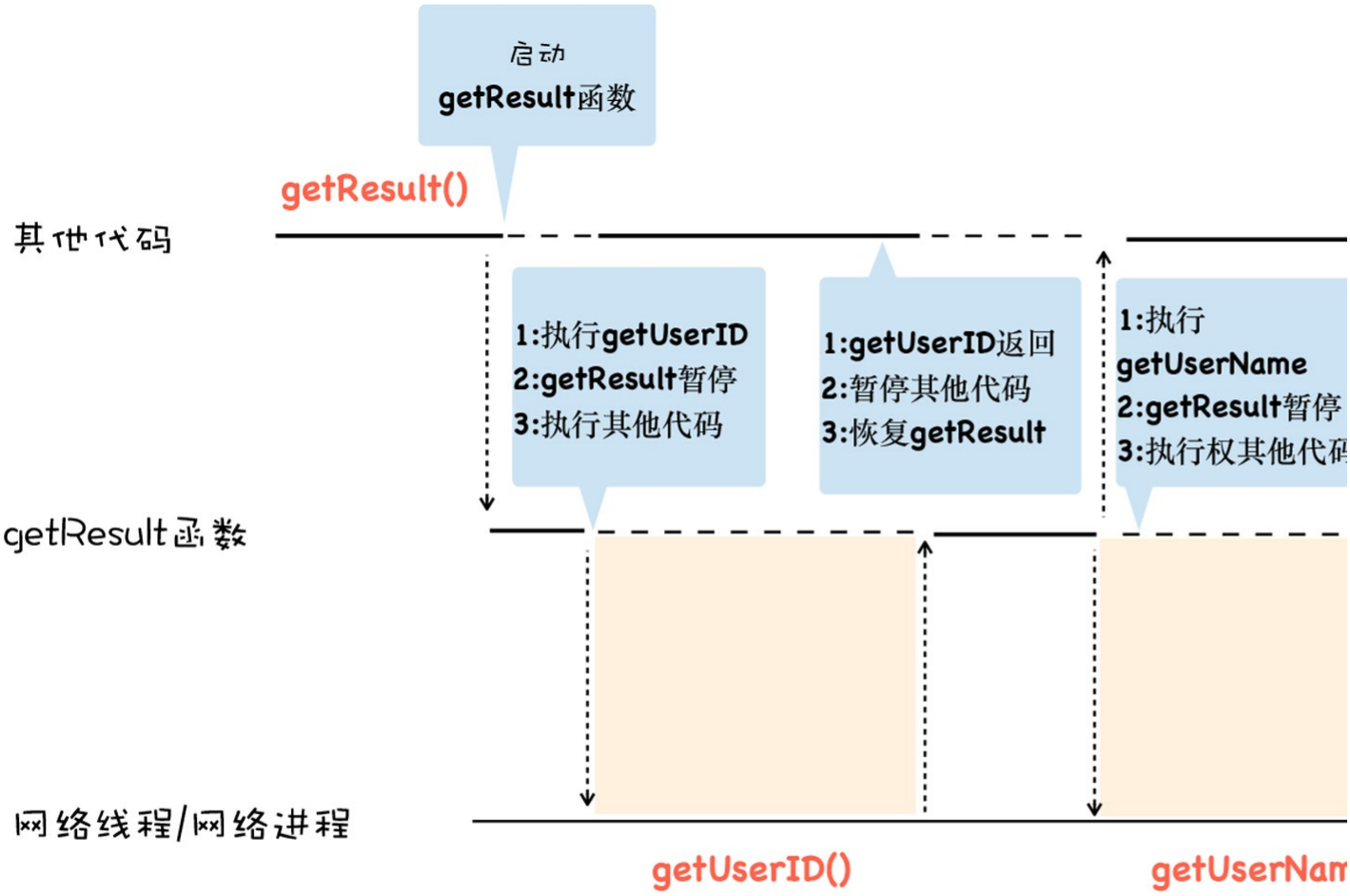
那么我们就需要思考，能不能更进一步，像编写同步代码的方式来编写异步代码，比如：

```
function getResult(){
  let id = getUserID()
  let name = getUserName(id)
  return name
}
```

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：

```
function* getResult() {

yield 'getUserID'

yield 'getUserName'

return 'name'

}

let result = getResult()

console.log(result.next().value)

console.log(result.next().value)
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。

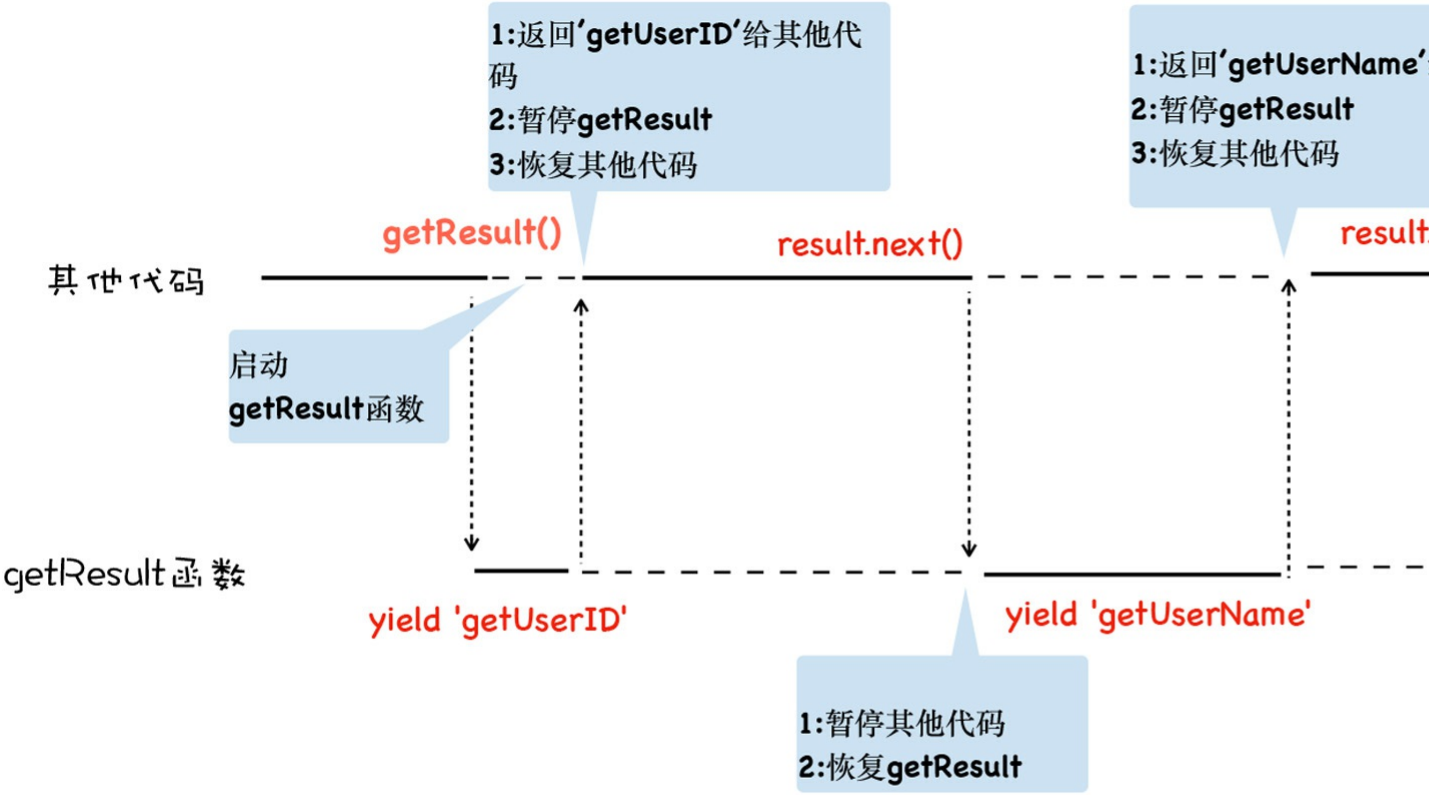
其实这就是生成器函数的特性，在生成器内部，如果遇到yield关键字，那么V8将返回关键字后面的内容给外部，并暂停该生成器函数的执行。生成器暂停执行后，外部的代码便开始执行，外部代码如果想要恢复生成器的执行，可以使用result.next方法。

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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}


let result = getResult()
result.next().value.then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    return result.next(response).value
}).then((response) => {
    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：

```
function* getResult() {
    let id_res = yield fetch(id_url);
    console.log(id_res)
    let id_text = yield id_res.text();
    console.log(id_text)


    let new_name_url = name_url + "?id=" + id_text
    console.log(new_name_url)


    let name_res = yield fetch(new_name_url)
    console.log(name_res)
    let name_text = yield name_res.text()
    console.log(name_text)
}
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：

```
async function getResult() {
```

```
    try {
        let id_res = await fetch(id_url)
        let id_text = await id_res.text()
        console.log(id_text)

        let new_name_url = name_url+"?id="+id_text
        console.log(new_name_url)


        let name_res = await fetch(new_name_url)
        let name_text = await name_res.text()
        console.log(name_text)
    } catch (err) {
        console.error(err)
    }
}
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。

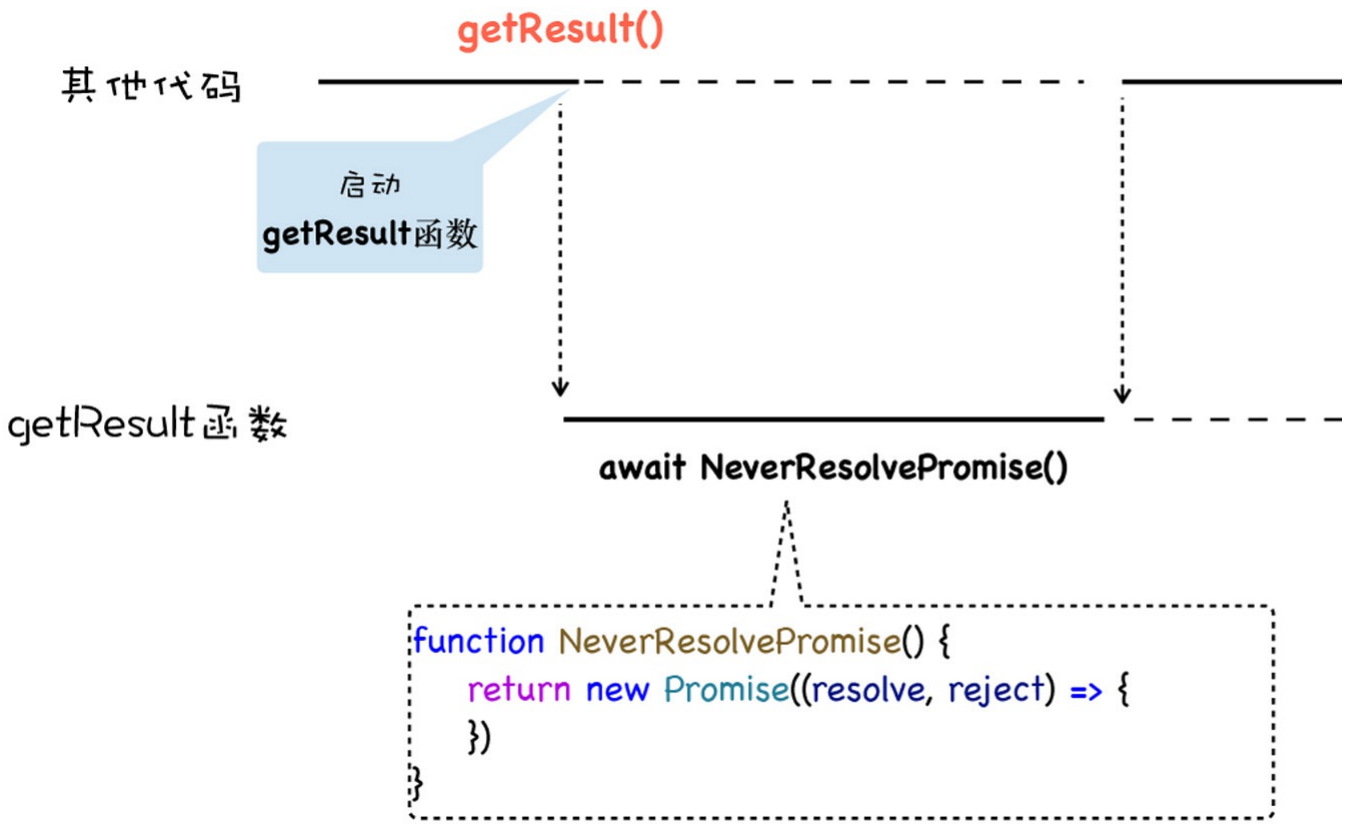
我们看下面这样一段代码：

```
function NeverResolvePromise(){
    return new Promise((resolve, reject) => {})
}
async function getResult() {
    let a = await NeverResolvePromise()
    console.log(a)
}
getResult()
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：

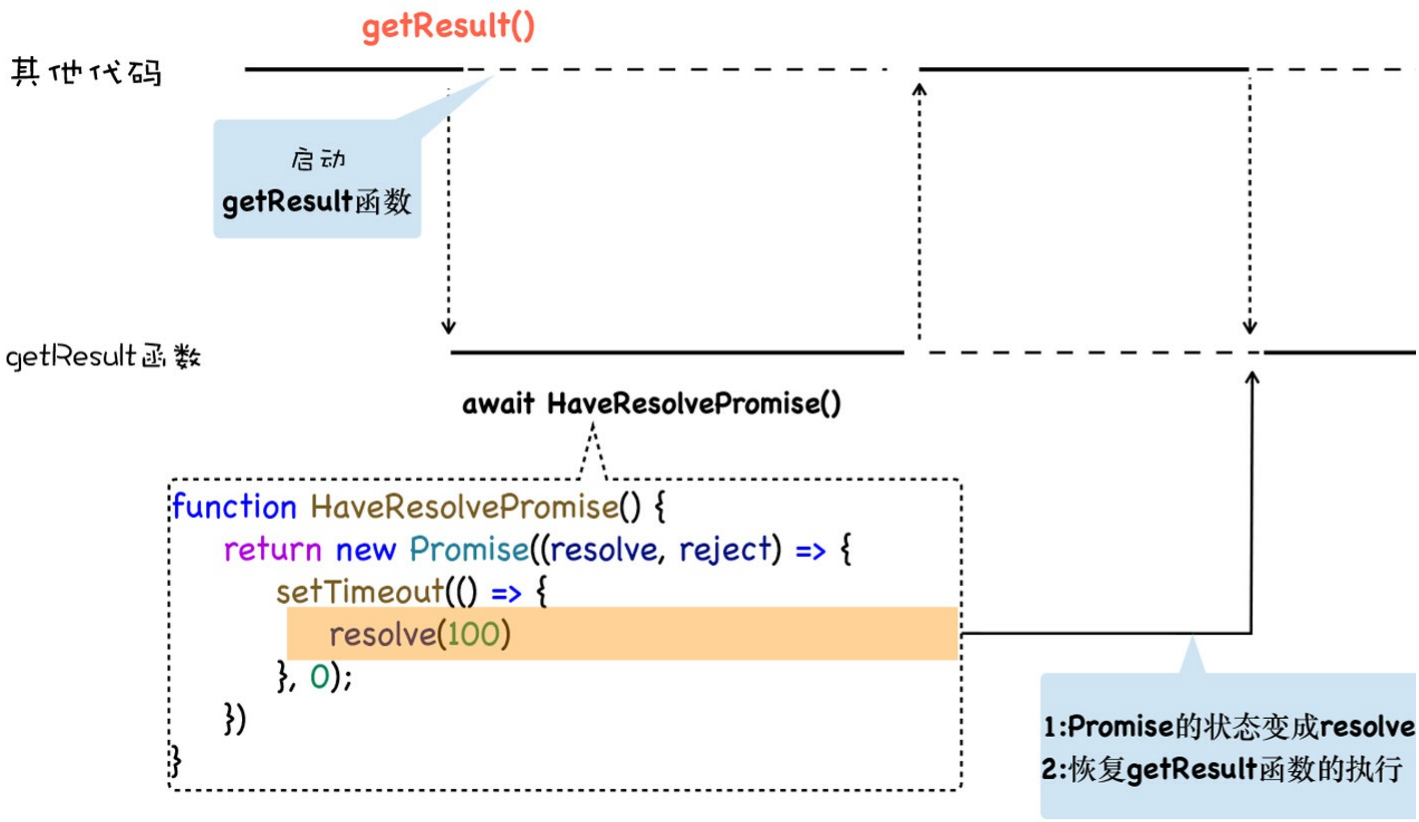


如果await等待的对象已经变成了resolve状态，那么V8就会恢复该协程的执行，我们可以修改下上面的代码，来证明下这个过程：

```
function HaveResolvePromise(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(100)
        }, 0);
    })
}
async function getResult() {
    console.log(1)
    let a = await HaveResolvePromise()
    console.log(a)
    console.log(2)
}
```

```
console.log(0)
getResult()
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：

```
function ResolvePromise(){
    return new Promise((resolve, reject) => {
            resolve(100)
      })
}
async function getResult() {
    let a = await ResolvePromise()
    console.log(a)
}
getResult()
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，使得代码变得不线性、不易阅读，这就是我们所说的回调地狱问题。

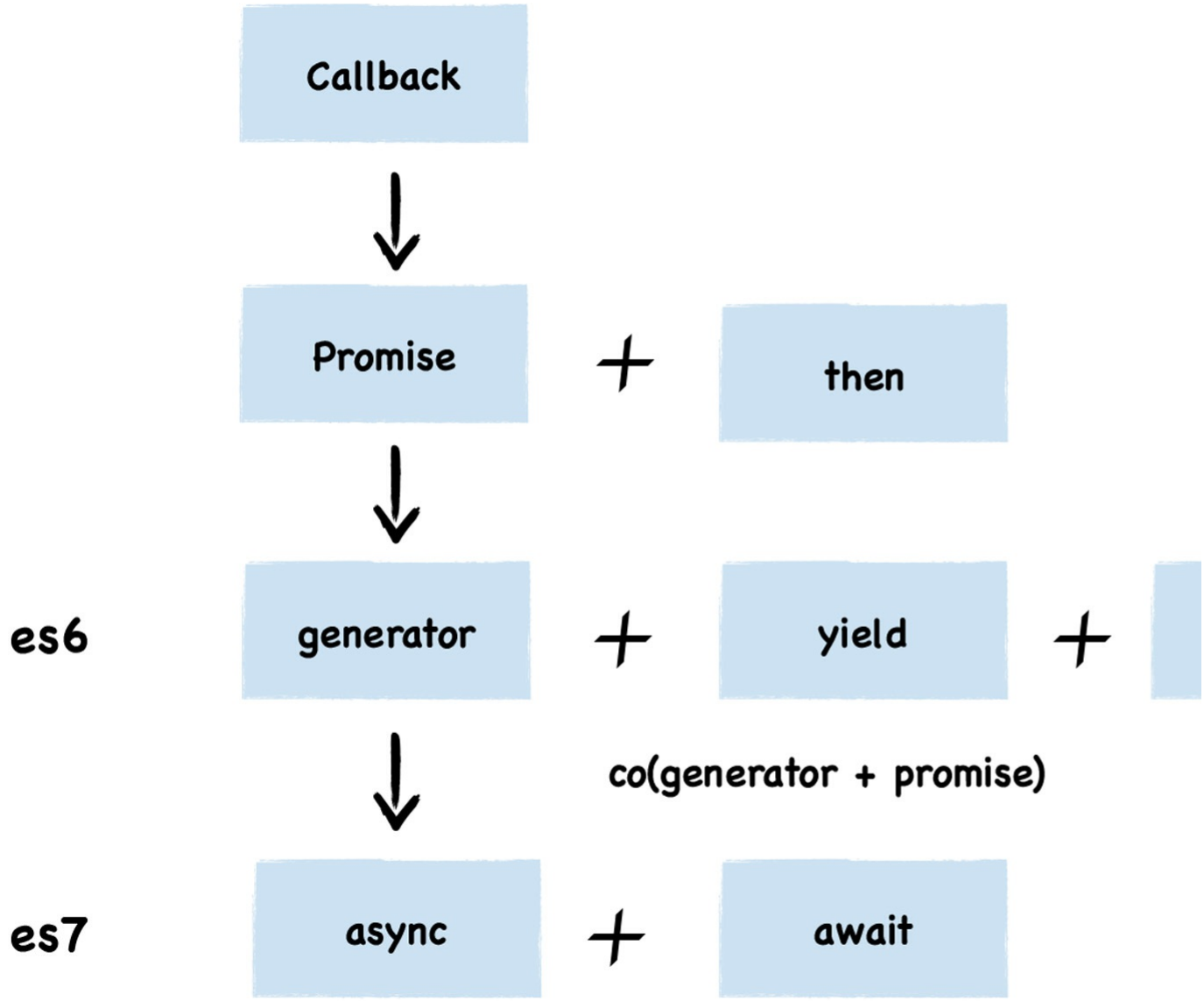
使用Promise能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

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。