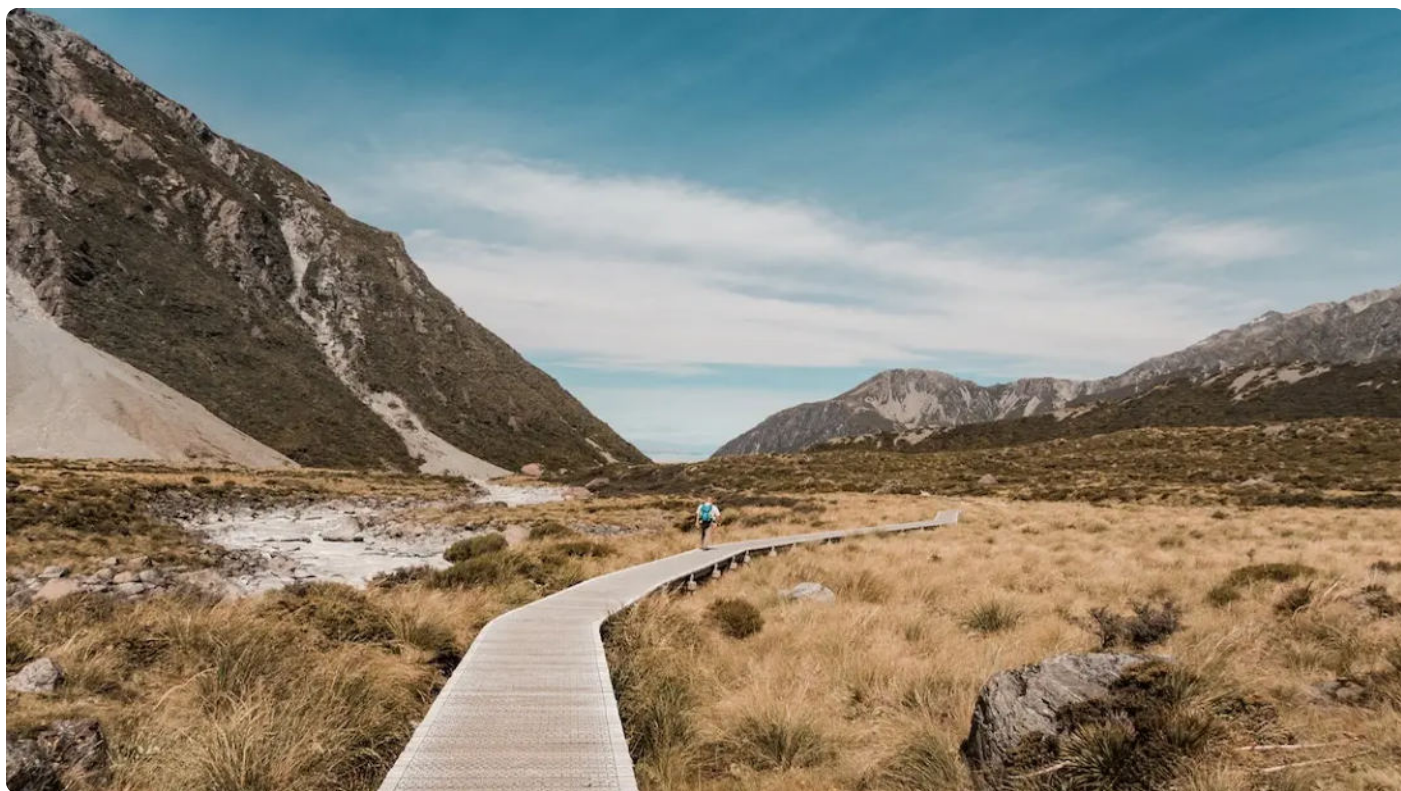


19 | Promise：使用Promise，告别回调函数

2019-09-17 李兵

《浏览器工作原理与实践》

课程介绍 >



讲述：李兵

时长 13:15 大小 15.17M



在 [上一篇](#) 文章中我们聊到了微任务是如何工作的，并介绍了 MutationObserver 是如何利用微任务来权衡性能和效率的。今天我们就接着来聊聊微任务的另外一个应用 **Promise**，DOM/BOM API 中新加入的 API 大多数都是建立在 Promise 上的，而且新的前端框架也使用了大量的 Promise。可以这么说，Promise 已经成为现代前端的“水”和“电”，很是关键，所以深入学习 Promise 势在必行。

不过，Promise 的知识点有那么多，而我们只有一篇文章来介绍，那应该怎么讲解呢？具体讲解思路是怎样的呢？

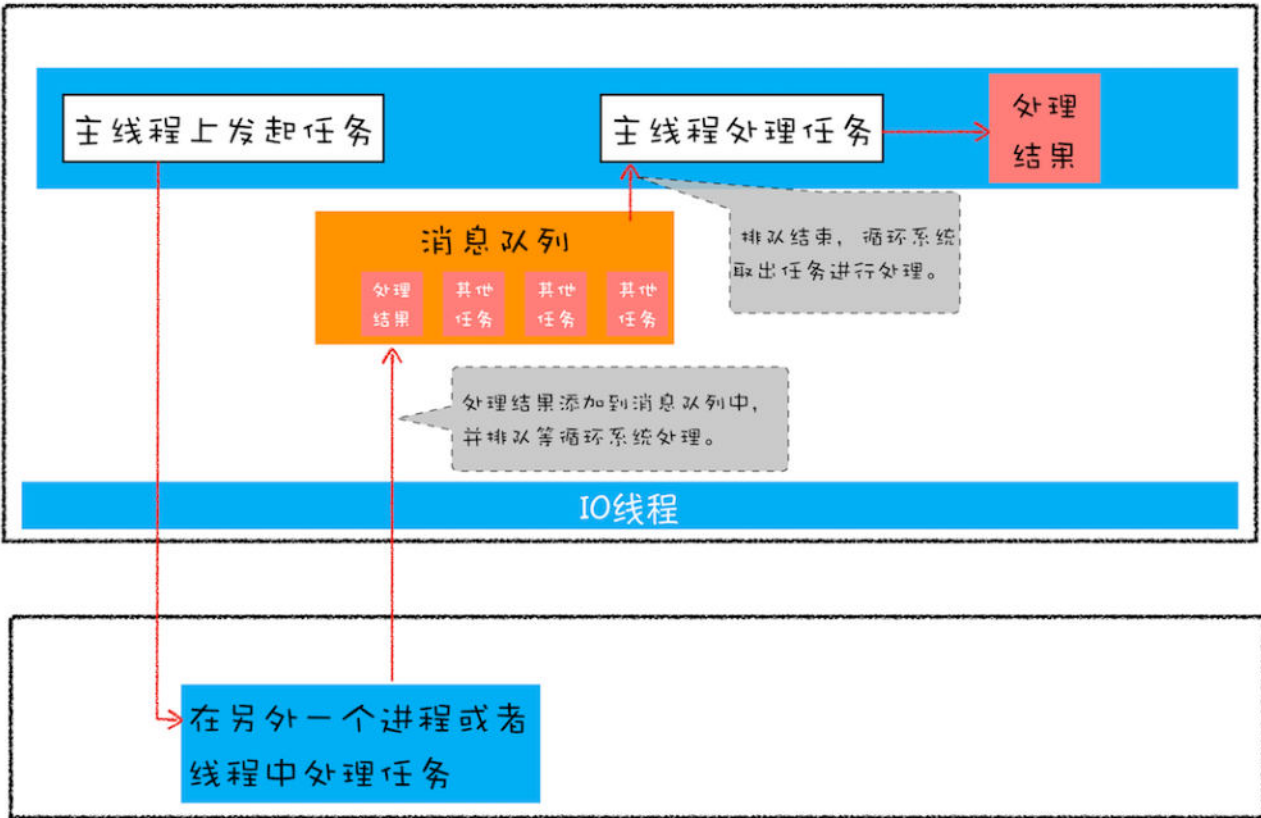
如果你想要学习一门新技术，最好的方式是先了解这门技术是如何诞生的，以及它所解决的问题是什么。了解了这些后，你才能抓住这门技术的本质。所以本文我们就来重点聊聊 JavaScript 引入 Promise 的动机，以及解决问题的几个核心关键点。



要谈动机，我们一般都是先从问题切入，那么 Promise 到底解决了什么问题呢？在正式开始介绍之前，我想有必要明确下，Promise 解决的是异步编码风格的问题，而不是一些其他的问题，所以接下来我们聊的话题都是围绕编码风格展开的。

异步编程的问题：代码逻辑不连续

首先我们来回顾下 JavaScript 的异步编程模型，你应该已经非常熟悉页面的事件循环系统了，也知道页面中任务都是执行在主线程之上的，相对于页面来说，主线程就是它整个的世界，所以在执行一项耗时的任务时，比如下载网络文件任务、获取摄像头等设备信息任务，这些任务都会放到页面主线程之外的进程或者线程中去执行，这样就避免了耗时任务“霸占”页面主线程的情况。你可以结合下图来看看这个处理过程：



Web 应用的异步编程模型

上图展示的是一个标准的异步编程模型，页面主线程发起了一个耗时的任务，并将任务交给另外一个进程去处理，这时页面主线程会继续执行消息队列中的任务。等该进程处理完这个任务后，会将该任务添加到渲染进程的消息队列中，并排队等待循环系统的处理。排队结束之后，循环系统会取出消息队列中的任务进行处理，并触发相关的回调操作。



这就是页面编程的一大特点：**异步回调**。

Web 页面的单线程架构决定了异步回调，而异步回调影响到了我们的编码方式，到底是如何影响的呢？

假设有一个下载的需求，使用 XMLHttpRequest 来实现，具体的实现方式你可以参考下面这段代码：

 复制代码

```
1 //执行状态
2 function onResolve(response){console.log(response) }
3 function onReject(error){console.log(error) }
4
5 let xhr = new XMLHttpRequest()
6 xhr.ontimeout = function(e) { onReject(e)}
7 xhr.onerror = function(e) { onReject(e) }
8 xhr.onreadystatechange = function () { onResolve(xhr.response) }
9
10 //设置请求类型，请求URL，是否同步信息
11 let URL = 'https://time.geekbang.com'
12 xhr.open('Get', URL, true);
13
14 //设置参数
15 xhr.timeout = 3000 //设置xhr请求的超时时间
16 xhr.responseType = "text" //设置响应返回的数据格式
17 xhr.setRequestHeader("X_TEST","time.geekbang")
18
19 //发出请求
20 xhr.send();
```

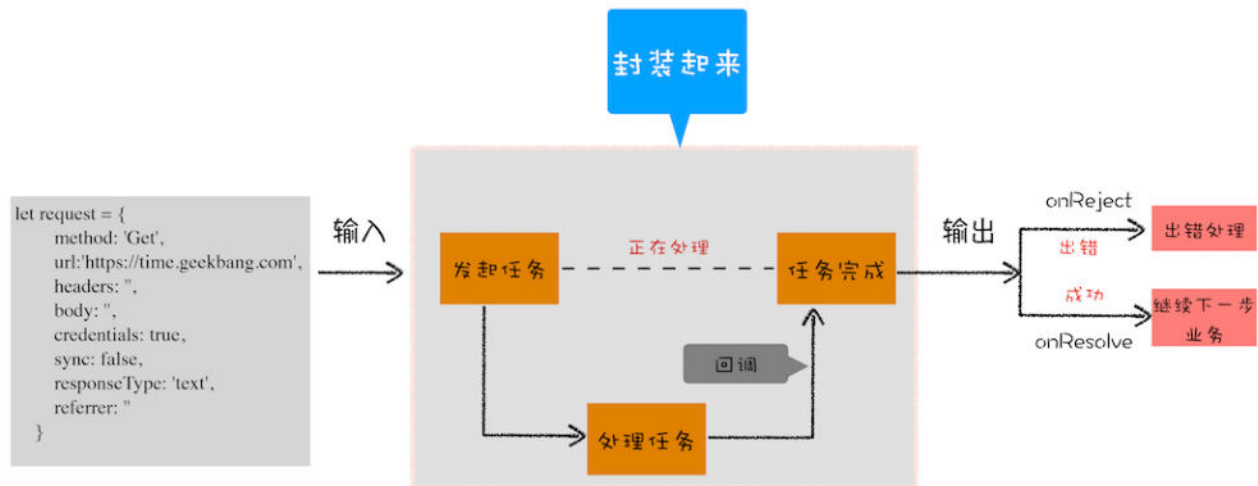
我们执行上面这段代码，可以正常输出结果的。但是，这短短的一段代码里面竟然出现了五次回调，这么多的回调会导致代码的逻辑不连贯、非线性，非常不符合人的直觉，这就是异步回调影响到我们的编码方式。

那有什么方法可以解决这个问题吗？当然有，我们可以封装这堆凌乱的代码，降低处理异步回调的次数。

封装异步代码，让处理流程变得线性

由于我们重点关注的是**输入内容（请求信息）**和**输出内容（回复信息）**，至于中间的异步请求过程，我们不想在代码里面体现太多，因为这会干扰核心的代码逻辑。整体思路如下图所示：





封装请求过程

从图中你可以看到，我们将 XMLHttpRequest 请求过程的代码封装起来了，重点关注输入数据和输出结果。

那我们就按照这个思路来改造代码。首先，我们把输入的 HTTP 请求信息全部保存到一个 request 的结构中，包括请求地址、请求头、请求方式、引用地址、同步请求还是异步请求、安全设置等信息。request 结构如下所示：

复制代码

```
1 //makeRequest用来构造request对象
2 function makeRequest(request_url) {
3     let request = {
4         method: 'Get',
5         url: request_url,
6         headers: '',
7         body: '',
8         credentials: false,
9         sync: true,
10        responseType: 'text',
11        referrer: ''
12    }
13    return request
14 }
```

然后就可以封装请求过程了，这里我们将所有的请求细节封装进 XFetch 函数，XFetch 代码如下所示：

```

1 // [in] request, 请求信息, 请求头, 延时值, 返回类型等
2 // [out] resolve, 执行成功, 回调该函数
3 // [out] reject 执行失败, 回调该函数
4 function XFetch(request, resolve, reject) {
5     let xhr = new XMLHttpRequest()
6     xhr.ontimeout = function (e) { reject(e) }
7     xhr.onerror = function (e) { reject(e) }
8     xhr.onreadystatechange = function () {
9         if (xhr.status = 200)
10             resolve(xhr.response)
11     }
12     xhr.open(request.method, URL, request.sync);
13     xhr.timeout = request.timeout;
14     xhr.responseType = request.responseType;
15     // 补充其他请求信息
16     // ...
17     xhr.send();
18 }

```

这个 XFetch 函数需要一个 request 作为输入，然后还需要两个回调函数 resolve 和 reject，当请求成功时回调 resolve 函数，当请求出现问题时回调 reject 函数。

有了这些后，我们就可以来实现业务代码了，具体的实现方式如下所示：

```

1 XFetch(makeRequest('https://time.geekbang.org'),
2     function resolve(data) {
3         console.log(data)
4     }, function reject(e) {
5         console.log(e)
6     })

```

新的问题：回调地狱

上面的示例代码已经比较符合人的线性思维了，在一些简单的场景下运行效果也是非常好的，不过一旦接触到稍微复杂点的项目时，你就会发现，如果嵌套了太多的回调函数就很容易使得自己陷入了回调地狱，不能自拔。你可以参考下面这段让人凌乱的代码：



```

1 XFetch(makeRequest('https://time.geekbang.org/?category'),
2     function resolve(response) {
3         console.log(response)
4     }

```

```

5         XFetch(makeRequest('https://time.geekbang.org/column'),
6             function resolve(response) {
7                 console.log(response)
8                 XFetch(makeRequest('https://time.geekbang.org'))
9                     function resolve(response) {
10                         console.log(response)
11                     }, function reject(e) {
12                         console.log(e)
13                     })
14             }, function reject(e) {
15                 console.log(e)
16             })
17     }, function reject(e) {
18         console.log(e)
19     })

```

这段代码是先请求time.geekbang.org/?category，如果请求成功的话，那么再请求time.geekbang.org/column，如果再次请求成功的话，就继续请求time.geekbang.org。也就是说这段代码用了三层嵌套请求，就已经让代码变得混乱不堪，所以，我们还需要解决这种嵌套调用后混乱的代码结构。

这段代码之所以看上去很乱，归结其原因有两点：

- **第一是嵌套调用**，下面的任务依赖上个任务的请求结果，并在上个任务的回调函数内部执行新的业务逻辑，这样当嵌套层次多了之后，代码的可读性就变得非常差了。
- **第二是任务的不确定性**，执行每个任务都有两种可能的结果（成功或者失败），所以体现在代码中就需要对每个任务的执行结果做两次判断，这种对每个任务都要进行一次额外的错误处理的方式，明显增加了代码的混乱程度。

原因分析出来后，那么问题的解决思路就很清晰了：

- **第一是消灭嵌套调用；**
- **第二是合并多个任务的错误处理。**

这么讲可能有点抽象，不过 Promise 已经帮助我们解决了这两个问题。那么接下来我们就来看看 Promise 是怎么消灭嵌套调用和合并多个任务的错误处理的。



Promise：消灭嵌套调用和多次错误处理

首先，我们使用 Promise 来重构 XFetch 的代码，示例代码如下所示：

 复制代码

```
1 function XFetch(request) {
2   function executor(resolve, reject) {
3     let xhr = new XMLHttpRequest()
4     xhr.open('GET', request.url, true)
5     xhr.ontimeout = function (e) { reject(e) }
6     xhr.onerror = function (e) { reject(e) }
7     xhr.onreadystatechange = function () {
8       if (this.readyState === 4) {
9         if (this.status === 200) {
10            resolve(this.responseText, this)
11          } else {
12            let error = {
13              code: this.status,
14              response: this.response
15            }
16            reject(error, this)
17          }
18        }
19      }
20      xhr.send()
21    }
22    return new Promise(executor)
23  }
```

接下来，我们再利用 XFetch 来构造请求流程，代码如下：

 复制代码

```
1 var x1 = XFetch(makeRequest('https://time.geekbang.org/?category'))
2 var x2 = x1.then(value => {
3   console.log(value)
4   return XFetch(makeRequest('https://www.geekbang.org/column'))
5 })
6 var x3 = x2.then(value => {
7   console.log(value)
8   return XFetch(makeRequest('https://time.geekbang.org'))
9 })
10 x3.catch(error => {
11   console.log(error)
12 })
```



你可以观察上面这两段代码，重点关注下 Promise 的使用方式。

- 首先我们引入了 Promise，在调用 XFetch 时，会返回一个 Promise 对象。
- 构建 Promise 对象时，需要传入一个 **executor 函数**，XFetch 的主要业务流程都在 executor 函数中执行。
- 如果运行在 excutor 函数中的业务执行成功了，会调用 resolve 函数；如果执行失败了，则调用 reject 函数。
- 在 excutor 函数中调用 resolve 函数时，会触发 promise.then 设置的回调函数；而调用 reject 函数时，会触发 promise.catch 设置的回调函数。

以上简单介绍了 Promise 一些主要的使用方法，通过引入 Promise，上面这段代码看起来就非常线性了，也非常符合人的直觉，是不是很酷？基于这段代码，我们就可以来分析 Promise 是如何消灭嵌套回调和合并多个错误处理了。

我们先来看看 Promise 是怎么消灭嵌套回调的。产生嵌套函数的一个主要原因是在发起任务请求时会带上回调函数，这样当任务处理结束之后，下个任务就只能在回调函数中来处理了。

Promise 主要通过下面两步解决嵌套回调问题的。

首先，Promise 实现了回调函数的延时绑定。回调函数的延时绑定在代码上体现就是先创建 Promise 对象 x1，通过 Promise 的构造函数 executor 来执行业务逻辑；创建好 Promise 对象 x1 之后，再使用 x1.then 来设置回调函数。示范代码如下：

 复制代码

```
1 //创建Promise对象x1，并在executor函数中执行业务逻辑
2 function executor(resolve, reject){
3     resolve(100)
4 }
5 let x1 = new Promise(executor)
6
7
8 //x1延迟绑定回调函数onResolve
9 function onResolve(value){
10     console.log(value)
11 }
12 x1.then(onResolve)
```

其次，需要将回调函数 onResolve 的返回值穿透到最外层。因为我们会根据 onResolve 函数的传入值来决定创建什么类型的 Promise 任务，创建好的 Promise 对象需要返回到最外层，



这样就可以摆脱嵌套循环了。你可以先看下面的代码：

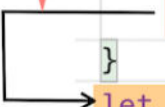
```
//创建Promise对象x1, 并在executor函数中执行业务逻辑
function executor(resolve, reject){
    resolve(100)
}
let x1 = new Promise(executor)

//x1延迟绑定回调函数onResovle
function onResovle(value){
    console.log(value)
    let x2 = new Promise((resolve, reject) => {
        resolve(value + 1)
    })
    console.log(x2)
    return x2
}

let x2 = x1.then(onResovle)
console.log(x2)

x2.then((value) => {
    console.log(value)
    console.log(x2)
})
```

内部返回值，
穿透到最外层



回调函数返回值穿透到最外层

现在我们知道 Promise 通过回调函数延迟绑定和回调函数返回值穿透的技术，解决了循环嵌套。

那接下来我们再来看看 Promise 是怎么处理异常的，你可以回顾 [上篇文章](#) 思考题留的那段代码，我把这段代码也贴在文中了，如下所示：

```
1 function executor(resolve, reject) {
2     let rand = Math.random();
3     console.log(1)
4     console.log(rand)
5     if (rand > 0.5)
6         resolve()
```

复制代码



```
7     else
8         reject()
9 }
10 var p0 = new Promise(executor);
11
12 var p1 = p0.then((value) => {
13     console.log("succeed-1")
14     return new Promise(executor)
15 })
16
17 var p3 = p1.then((value) => {
18     console.log("succeed-2")
19     return new Promise(executor)
20 })
21
22 var p4 = p3.then((value) => {
23     console.log("succeed-3")
24     return new Promise(executor)
25 })
26
27 p4.catch((error) => {
28     console.log("error")
29 })
30 console.log(2)
```

这段代码有四个 Promise 对象：p0~p4。无论哪个对象里面抛出异常，都可以通过最后一个对象 p4.catch 来捕获异常，通过这种方式可以将所有 Promise 对象的错误合并到一个函数来处理，这样就解决了每个任务都需要单独处理异常的问题。

之所以可以使用最后一个对象来捕获所有异常，是因为 Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被 onReject 函数处理或 catch 语句捕获为止。具备了这样“冒泡”的特性后，就不需要在每个 Promise 对象中单独捕获异常了。至于 Promise 错误的“冒泡”性质是怎么实现的，就留给你课后思考了。

通过这种方式，我们就消灭了嵌套调用和频繁的错误处理，这样使得我们写出来的代码更加优雅，更加符合人的线性思维。

Promise 与微任务

讲了这么多，我们似乎还没有将微任务和 Promise 关联起来，那么 Promise 和微任务的关系到底体现哪里呢？

我们可以结合下面这个简单的 Promise 代码来回答这个问题：



```
1 function executor(resolve, reject) {
2     resolve(100)
3 }
4 let demo = new Promise(executor)
5
6 function onResolve(value){
7     console.log(value)
8 }
9 demo.then(onResolve)
```

对于上面这段代码，我们需要重点关注下它的执行顺序。

首先执行 `new Promise` 时，`Promise` 的构造函数会被执行，不过由于 `Promise` 是 V8 引擎提供的，所以暂时看不到 `Promise` 构造函数的细节。

接下来，`Promise` 的构造函数会调用 `Promise` 的参数 `executor` 函数。然后在 `executor` 中执行了 `resolve`，`resolve` 函数也是在 V8 内部实现的，那么 `resolve` 函数到底做了什么呢？我们知道，执行 `resolve` 函数，会触发 `demo.then` 设置的回调函数 `onResolve`，所以可以推测，`resolve` 函数内部调用了通过 `demo.then` 设置的 `onResolve` 函数。

不过这里需要注意一下，由于 `Promise` 采用了回调函数延迟绑定技术，所以在执行 `resolve` 函数的时候，回调函数还没有绑定，那么只能推迟回调函数的执行。


这样按顺序陈述可能把你绕晕了，下面来模拟实现一个 `Promise`，我们会实现它的构造函数、`resolve` 方法以及 `then` 方法，以方便你能看清楚 `Promise` 的背后都发生了什么。这里我们就把这个对象称为 `Bromise`，下面就是 `Bromise` 的实现代码：

```
1 function Bromise(executor) {
2     var onResolve_ = null
3     var onReject_ = null
4     //模拟实现resolve和then，暂不支持reject
5     this.then = function (onResolve, onReject) {
6         onResolve_ = onResolve
7     };
8     function resolve(value) {
9         //setTimeout(()=>{
10             onResolve_(value)
11             // },0)
12     }
```



```
13     executor(resolve, null);
14 }
```

观察上面这段代码，我们实现了自己的构造函数、resolve、then 方法。接下来我们使用 Bromise 来实现我们的业务代码，实现后的代码如下所示：

 复制代码

```
1 function executor(resolve, reject) {
2     resolve(100)
3 }
4 //将Promise改成我们自己的Bromsie
5 let demo = new Bromise(executor)
6
7 function onResolve(value){
8     console.log(value)
9 }
10 demo.then(onResolve)
```

执行这段代码，我们发现执行出错，输出的内容是：

 复制代码

```
1 Uncaught TypeError: onResolve_ is not a function
2     at resolve (<anonymous>:10:13)
3     at executor (<anonymous>:17:5)
4     at new Bromise (<anonymous>:13:5)
5     at <anonymous>:19:12
```

之所以出现这个错误，是由于 Bromise 的延迟绑定导致的，在调用到 onResolve_ 函数的时候，Bromise.then 还没有执行，所以执行上述代码的时候，当然会报“onResolve_ is not a function”的错误了。

也正是因为此，我们要改造 Bromise 中的 resolve 方法，让 resolve 延迟调用 onResolve_。

要让 resolve 中的 onResolve_ 函数延后执行，可以在 resolve 函数里面加上一个定时器，让其延时执行 onResolve_ 函数，你可以参考下面改造后的代码：



 复制代码

```
1 function resolve(value) {
2     setTimeout(()=>{
```

```
3         onResolve_(value)
4     }, 0)
5 }
```

上面采用了定时器来推迟 onResolve 的执行，不过使用定时器的效率并不是太高，好在我们有微任务，所以 Promise 又把这个定时器改造成了微任务了，这样既可以让 onResolve_ 延时被调用，又提升了代码的执行效率。这就是 Promise 中使用微任务的原由了。

总结

好了，今天我们就聊到这里，下面我来总结下今天所讲的内容。

首先，我们回顾了 Web 页面是单线程架构模型，这种模型决定了我们编写代码的形式——异步编程。基于异步编程模型写出来的代码会把一些关键的逻辑点打乱，所以这种风格的代码不符合人的线性思维方式。接下来我们试着把一些不必要的回调接口封装起来，简单封装取得了一定的效果，不过，在稍微复制点的场景下依然存在着回调地狱的问题。然后我们分析了产生回调地狱的原因：

1. 多层嵌套的问题；
2. 每种任务的处理结果存在两种可能性（成功或失败），那么需要在每种任务执行结束后分别处理这两种可能性。

Promise 通过回调函数延迟绑定、回调函数返回值穿透和错误“冒泡”技术解决了上面的两个问题。

最后，我们还分析了 Promise 之所以要使用微任务是由 Promise 回调函数延迟绑定技术导致的。

思考时间

终于把 Promise 讲完了，这一篇文章非常有难度，所以需要课后慢慢消消化，再次提醒，Promise 非常重要。那么今天我给你留三个思考题：

1. Promise 中为什么要引入微任务？
2. Promise 中是如何实现回调函数返回值穿透的？




3. Promise 出错后，是怎么通过“冒泡”传递给最后那个捕获异常的函数？

这三个问题你不用急着完成，可以先花一段时间查阅材料，然后再来一道一道解释。搞清楚了这三道题目，你也就搞清楚了 Promise。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 22  提建议

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

上一篇 18 | 宏任务和微任务：不是所有任务都是一个待遇

下一篇 20 | async/await：使用同步的方式去写异步代码

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费 





Geek_Jamorex

2019-09-17

这三个题目非常重要，就跟做笔记一样回答了

1、Promise 中为什么要引入微任务？

由于promise采用.then延时绑定回调机制，而new Promise时又需要直接执行promise中的方法，即发生了先执行方法后添加回调的过程，此时需等待then方法绑定两个回调后才能继续执行方法回调，便可将回调添加到当前js调用栈中执行结束后的任务队列中，由于宏任务较多容易堵塞，则采用了微任务

2、Promise 中是如何实现回调函数返回值穿透的？

首先Promise的执行结果保存在promise的data变量中，然后是.then方法返回值为使用resolved或rejected回调方法新建的一个promise对象，即例如成功则返回new Promise (resolved)，将前一个promise的data值赋给新建的promise

3、Promise 出错后，是怎么通过“冒泡”传递给最后那个捕获

promise内部有resolved_和rejected_变量保存成功和失败的回调，进入.then (resolved, rejected) 时会判断rejected参数是否为函数，若是函数，错误时使用rejected处理错误；若不是，则错误时直接throw错误，一直传递到最后的捕获，若最后没有被捕获，则会报错。可通过监听unhandledrejection事件捕获未处理的promise错误

共 8 条评论 >

👍 77



空间

2019-09-18

异步AJAX请求是宏任务吧？Promise是微任务，那么用Promise进行的异步Ajax调用时宏任务还是微任务？

作者回复: ajax就是xmlHttpRequest，必然是宏任务！

准确地说，Promise在执行resolve或者reject时，触发微任务，所以在Promise的executor函数中调用xmlHttpRequest会触发宏任务。

如果xmlHttpRequest请求成功了，通过resolve触发微任务



如果xmlHttpRequest请求失败了，通过reject触发微任务

共 2 条评论 >

👍 55



皮皮大神

2019-09-18

老师，我觉得这章没有前面的讲得透彻，手写的bromise非常不完整，希望老师答疑的时候可以带我们写一遍完整promise源码，三种状态的切换，还有.then为什么可以连续调用，内部如何解决多层异步嵌套，我觉得都很值得讲解，老师带我们飞。

作者回复: 这个加餐可以有!

这篇问题主要在宏观视角建立对Promise的认知。

关于手写 或者细节内容课程结束之后我们慢慢聊。

目前被编辑追稿子压力大，好多问题没及时回复还望理解哈。课程结束之后我会相信回复大家的问题的。

共 11 条评论 >

👍 23



穿秋裤的男孩

2020-04-17

promise.then是订阅者，订阅promise状态的改变，并且负责回掉；promise.resolve和promise.reject为发布者，发布promise的状态改变的信息。



👍 9



Rapheal

2019-10-05

Promise的改进版，测试过也无问题。之前使用闭包存放所有回调函数有些问题，所有的Promise对象都是共享，这样会造成全局数据结构有问题。当前是基于回调函数数组传递在Promise对象之间传递实现。

```
function _Promise(executor) {  
  this._resolve = [];  
  this._reject = [];  
  this._catch;
```

/*临时保存引用*/



```
let self = this;
```

```
this.then = function (resolve, reject) {  
  resolve && this._resolve.push(resolve);  
  reject && this._reject.push(reject);  
  return this;  
}
```

```
this.resolve = function (data) {  
  setTimeout(() => {  
    let callback = self._resolve.shift();  
    self._reject && self._reject.shift();  
    let pro;  
    callback && (pro = callback(data));  
    self._resolve && (pro._resolve = self._resolve);  
    self._reject && (pro._reject = self._reject);  
    self._catch && (pro._catch = self._catch);  
  
    }, 0)  
}
```

```
this.reject = function (error) {  
  setTimeout(() => {  
    let callback;  
    self._reject && (callback = self._reject.shift());  
    callback && callback(error);  
    callback || self._catch(error);  
  
    }, 0);  
}
```

```
this.catch = function (callback) {  
  this._catch = callback;  
  return this;  
}  
executor(this.resolve, this.reject);  
}
```

```
function executor(resolve, reject) {
```



```
let rand = Math.random();
console.log(1)
console.log(rand)
if (rand > 0.5)
  resolve(rand)
else
  reject(rand)
}

var p0 = new _Promise(executor);

var p1 = p0.then((value) => {
  console.log("succeed-1")
  return new _Promise(executor)
})

var p3 = p1.then((value) => {
  console.log("succeed-2")
  return new _Promise(executor)
})

var p4 = p3.then((value) => {
  console.log("succeed-3")
  return new _Promise(executor)
})

p4.catch((error) => {
  console.log("error")
})
```

console.log(2)



9



Angus

2019-09-20

看完这节之后我自己去实现了手写Promise，回顾了一下Promise，关于这方面的文章很多，我觉得老师大可不必在这里花大量篇幅去讲。专栏的名字是浏览器工作原理与实践，所以我希望老师能够更加着重这一方面的讲解。



共 2 条评论 >



6

芥小爷
2020-04-15

不是说要手写promise吗，，怎么在教程里面没有看到

共 1 条评论 >

👍 6

Geek_be6d3f

2020-03-11

请问以下，在“异步编程的问题：代码逻辑不连续”这段中，代码下方的第一行，老师说“这短短的一段代码里面竟然出现了五次回调”，可是我怎么数，都只有三次回调啊，还有两次在哪里？

共 1 条评论 >

👍 4

許敲敲

2019-09-17

面试手写promise也不怕了

作者回复: 手写我加餐来讲

共 8 条评论 >

👍 3

张萌

2020-05-24

一个简易版的 Promise

```
class Promise {
  constructor(fn) {
    this.state = Promise.PENDING;
    this.value = undefined;
    this.reason = null;
    this.onFulfilledCallbacks = [];
    this.onRejectedCallbacks = [];
    fn(this.resolve.bind(this), this.reject.bind(this));
  }

  then(onFulfilled, onRejected) {
    if (this.state === Promise.FULFILLED) {
      onFulfilled(this.value);
    } else if (this.state === Promise.REJECTED) {
      onRejected(this.reason);
    } else {
      this.onFulfilledCallbacks.push(onFulfilled);
      this.onRejectedCallbacks.push(onRejected);
    }
  }
}
```



```

    }
    return this;
  }

  resolve(value) {
    this.state = Promise.FULFILLED;
    this.value = value;
    if (value.constructor === this.constructor) {
      value.onFulfilledCallbacks = [...this.onFulfilledCallbacks];
      value.onRejectedCallbacks = [...this.onRejectedCallbacks];
    } else {
      this.onFulfilledCallbacks.forEach((item) => {
        if (typeof item === 'function') {
          item(value);
        }
      });
    }
  }
}

```

```

  reject(reason) {
    this.state = Promise.REJECTED;
    this.reason = reason;
    this.onRejectedCallbacks.forEach((item) => {
      if (typeof item === 'function') {
        item(reason);
      }
    });
  }
}

```

```

Promise.PENDING = 'pending';
Promise.FULFILLED = 'fulfilled';
Promise.REJECTED = 'rejected';

```

```

module.exports = Promise;

```

共 2 条评论 >

👍 2



Hurry

2019-09-20

这个太赞了 “ Promise 通过回调函数延迟绑定、回调函数返回值穿透和和错误“冒泡”技术

“，之前看到别人手写实现 Promise，代码虽然可以看懂，但是理解不深，所以关键还是看如何实现这个三个点 回调函数延迟绑定、回调函数返回值穿透和和错误“冒泡”，结合这三个点和 promise API，手写一个 Promise, So easy

```
``js
class PromiseSimple {
  constructor(executionFunction) {
    this.promiseChain = []; // 1.通过数组存储 callback，实现callback 延迟执行
    this.handleError = () => {};

    this.onResolve = this.onResolve.bind(this);
    this.onReject = this.onReject.bind(this);

    executionFunction(this.onResolve, this.onReject);
  }

  then(onResolve) {
    this.promiseChain.push(onResolve);

    return this;
  }

  catch(handleError) {
    this.handleError = handleError;

    return this;
  }

  onResolve(value) {
    let storedValue = value;

    try {
      this.promiseChain.forEach((nextFunction) => {
        storedValue = nextFunction(storedValue); // 2.循环，实现 callback 值传递
      });
    } catch (error) { // 3. try catch, 实现错误值冒泡
      this.promiseChain = [];

      this.onReject(error);
    }
  }
}
```



```
onReject(error) {  
  this.handleError(error);  
}  
}  
...
```

共 1 条评论 >

👍 3



林高鸿

2021-08-02

老师，我理解是这样，这里感觉大部分人（可能也包括老师）都弄错了，我理解的是：

对我们（普通使用者）来说，用 Promise 是因为有宏任务问题（AJAX，SetTimeout）需要解决，而专注问题解决时是不需要考虑工具（Promise）自身实现原理（微任务）的

简言之，对普通使用者来说，把 Promise 和微任务联系起来是本末倒置

PS：其实，如果能保证用 Promise 解决的是异步问题（宏任务/微任务），那 Promise 自身实现原理也不需要微任务来“延迟绑定”（因为异步回来要 resolve 时，then 一定已经执行绑定...）



👍 1



Condor Hero 

2021-05-10

```
```js  
new Promise((r, rj) => {
 r();
}).then(() => {
 new Promise((r, rj) => {
 r();
 }).then(() => {
 console.log("inner then1")
 return new Promise((r, rj) => {
 r();
 })
 }).then(() => {
 console.log("inner then2")
 })
})
})
```



```
.then(() => {
 console.log("outer then2");
})
.then(() => {
 console.log("outer then3");
})
.then(() => {
 console.log("outer then4");
})
.then(() => {
 console.log("outer then5");
})
.then(() => {
 console.log("outer then6");
})
...
```

输出结果：

```
...
inner then1
outer then2
outer then3
outer then4
inner then2
outer then5
outer then6
...
```

老师能帮忙解释下这段代码的输出逻辑吗？

搞不懂输出 `outer then2` 之后紧接着输出了 `outer then3` 和 `outer then4`，然后才输出 `inner then2`。

而我理解的是 `outer then2` 之后直接输出 `inner then2` 才对，我理解的顺序：

```
...
inner then1
outer then2
inner then2
outer then3
outer then4
outer then5
```



outer then6

...

共 1 条评论 >

👍 2



悠米

2020-08-31

老师，手写 Promise 什么时候会有？非常期待~



👍 1



胖虎

2019-11-22

"回调函数返回值穿透到最外层" 这句话配合老师您讲的例子是错误的 return x2 这个x2按照您那种方式没办法返回到最外层 最外层的x2和里面函数return 出来的x2根本就是两个东西



👍 1



Chao

2019-09-17

老师 你有答疑环节吗

作者回复: 有，现在写稿子时间紧，等我主要稿件写完会抽出大把时间来专门解答问题。

共 2 条评论 >

👍 1



撒哈拉

2021-10-25

promise 为了解决嵌套调用，我理解。为了解决成功和错误多次处理我是没想到的



👍



袭

2021-05-27

延迟绑定和返回值穿透，可以理解成是提供了新的思路：原思路是嵌套处理任务结果。这种写法是先声明任务，再声明任务的返回处理流程，实际上是创造了独立性，把原来的嵌套变成了分割的两步，再用微任务把两部分连起来



👍



袭

2021-05-27

1.因为需要提供resolve函数给executor,而resolve并未指名，所以需要通过微任务机制等then中绑定后再执行。



2.executor生成了业务相关数据，而resolve进行了返回值的返回，因此在then绑定时才明确了返回值是什么，从而实现从resolve函数穿透executor



**tobemaster**

2021-05-10

有个问题非常困惑。

```
const p1 = new Promise(resolve => {
 console.log('begin');
 resolve('then1');
}).then(v => {
 console.log(v);
 return 'then2';
})

// then 链式调用，和微任务的产生关系
new Promise(resolve => {
 console.log(1);
 resolve();
})
 .then(() => {
 console.log(2);
 })
 .then(() => {
 console.log(3);
 })
 .then(() => {
 console.log(4);
 syncSleep(1000)
 })
 .then(() => {
 console.log(5);
 })

const p2 = new Promise(resolve => {

 resolve(p1);
})

p2.then(v => console.log(v));
```



这段代码的实际结果和我预期的不一致。

我的预期是

begin

1

then1

2

then2

3

4

5

但是实际结果是

begin

1

then1

2

3

4

then2

5

主要是 P1 变成接受状态 (fullfill) 后, P2 究竟何时 变成接受状态, 这个时机没太懂

