黑客派

Promise 技术调研 - 回调地狱的产生原因与解决方式

作者：zjhch123

原文链接：https://hacpai.com/article/1534856364622

来源网站：黑客派

<h2 id="产生原因">产生原因</h2>

在前端技术刀耕火种时代，让人闻之变色的一个词就是“回调”。因为设计原因导致 JavaScript 这门语言是单线程执行的，这就导致一些耗时的操作会阻塞当前运行线程。
 为了解决这个问题机智的开发者们引入了“同步”和“异步”这两个概念。
 打个很简单的比方，去肯德基买汉堡在柜台上排队买就是同步的，因为必须等待前面的人买好取到餐才能轮到我们；而使用手机点餐就是步的，下完单之后我们可以想干嘛干嘛，等到收银员喊到我的号的时候再去取餐即可。
 在以上景中，去取餐就是我向肯德基注册的一个回调函数，当我的餐准备就绪，收银员喊我，就相当于调用回调函数。
 但是为什么程序员们谈回调色变呢？究其原因是因为层层回调会造成所谓的“回调狱(callback hell)”
 就像这样：
<pre><code class="highlight-chroma">fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
</code></pre>
好吧，不管我们能不能理解以上代码。总之，当多个异步任务需要顺序执行的时候，在刀耕火种年代，程序员们不得不忍受着这样的煎熬。
<h2 id="解决办法">解决办法</h2>
其实解决回调地狱的办法有很多，从代码书写层面就可以将绝大部分回调代码写的尽量简单易懂但这都不是我们今天的主角，我们今天主要讲讲 <code>Promise</code>。

```
<script>
    (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<h3 id="Promise">Promise</h3>
<p><code>Promise</code> 自 ES6 起成为 JavaScript 的语言标准。但是其最早是由 JavaScript
区提出并实现的。<code>Promise</code> 规范和标准了异步操作 API，基本上所有的异步操作都
以使用 <code>Promise</code> 的写法处理。<code>Promise</code> 对象内部保存着异步操
的结果，并通过链式调用的方式避免了回调函数层层嵌套的写法。</p>
<h4 id="基本用法">基本用法</h4>
<pre><code class="highlight-chroma">new Promise((resolve, reject) =&gt; {
  setTimeout(() =&gt; {
    resolve('success')
  }, 1000)
}).then(res =&gt; {
  console.log(res)  // success
})
</code></pre>
<p><code>Promise</code> 构造函数接收一个函数作为参数，这个函数的两个参数分别为 <cod
>resolve</code> 和 <code>reject</code>。这也是两个函数，其值会由 JavaScript 传入，使用
只需要在异步操作完成时调用 <code>resolve</code> 函数并传入下一步操作所需要的值即可。使
者可以通过链式调用的方式为 <code>Promise</code> 对象添加后续操作。<br> <code>reject<
code> 函数则是在异步操作发生异常时被调用，此时 <code>Promise</code> 可以捕获到传入 <c
de>reject</code> 参数中的值。</p>
<pre><code class="highlight-chroma">new Promise((resolve, reject) =&gt; {
  setTimeout(() =&gt; {
    reject('error')
  }, 1000)
})
.then(res =&gt; console.log('进入then: ' + res))
.catch((e) =&gt; console.log('进入catch: ' + e)) // 进入catch: error
</code></pre>
<p>值得一提的是，<code>Promise</code> 代码不同于其他函数，对传入 <code>Promise</co
e> 构造方法中的函数不需要显示的调用执行，其会直接执行，且是作为同步任务被执行的。</p>
<pre><code class="highlight-chroma">setTimeout(() =&gt; console.log('timeout'))
new Promise((resolve, reject) =&gt; {
  console.log('Promise')
});
console.log('main')

/*

● Promise

● main

● timeout

*/

</code></pre>

<h4 id="改写回调函数">改写回调函数</h4>
<p>在远古时期，使用 <code>jQuery</code> 发送 <code>ajax</code> 请求的代码类似于以
：</p>
<pre><code class="highlight-chroma">$.ajax({
  url: '/api/user/getInfo',
```

```
  data: {},
  dataType: 'json',
  success: function (data) {
    // process success
  },
  error: function(err) {
    // process error
  }
})
</code></pre>
<p>如果此时需要有操作在 <code>ajax</code> 请求之后执行，则就需要在 <code>success</co
e> 上挂载回调函数。如果此时这个操作内又包含了异步操作，那代码就会变得冗长乏味，像老太太
裹脚布一般。<br> 而在有了 <code>Promise</code> 之后，我们可以将普通的 <code>ajax</co
e> 方法封装为 <code>Promise</code> 方法</p>
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></scr
pt>
<!-- 黑客派PC帖子内嵌-展示 -->
<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342"
data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></in
>
<script>
     (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<pre><code class="highlight-chroma">function ajax(url, data = {}) {
  return new Promise((resolve, reject) =&gt; {
    $.ajax({
      url,
      data,
      dataType: 'json',
      success: resolve
      error: reject
    })
  })
}
</code></pre>
<p>注意 <code>ajax</code> 函数，函数构造了 <code>Promise</code> 对象并将其 <code>r
turn</code> 出来，这是帮助我们书写可读性高的异步代码的关键。<br> 之后，我们可以使用 <co
e>Promise</code> 的链式调用方式来处理请求。</p>
<pre><code class="highlight-chroma">ajax('/api/user/getInfo')
 .then(result =&gt; {
  // process result
  return ajax('/api/user/getOrder', { id: result.userId }))
}).then(result =&gt; {
  // process result
  return ajax('/api/user/getMessage', { id: result.userId })
}).then(result =&gt; {
  // process result
  // do something ...
})
</code></pre>
<p>注意每一个 <code>then</code> 的参数函数内我们又调用了个 <code>ajax</code> 函数
即返回了一个 <code>Promise</code> 对象，这也是 <code>Promise</code> 的链式调用的关
所在。</p>
<h4 id="缺陷">缺陷</h4>
```


 <code>Promise</code> 函数改变了之前回调地狱的写法，但是在根本上还是函数套函数，起来不是那么的美观
 <code>Promise</code> 一经执行，无法中断，除非抛出异常
 在 <code>Promise</code> 外部无法通过 <code>try/catch</code> 的方式捕获 <code>Promise</code> 内部抛出的异常。

<h3 id="Async-Await">Async/Await</h3>
可以延展的说一下 <code>async/await</code>。尽管这是一个 ES7 标准内的语法。
 <code>async/await</code> 可以将 <code>Promise</code> 代码组织的更像同步代码一样，其书方式就和之前写同步代码一样，只是需要加上相应关键字。
 例如，将之前的 <code>Promise</code> 代码改写为 <code>async/await</code>
<pre><code class="highlight-chroma">async function request(id) {
  const result1 = await ajax('/api/user/getInfo', { id })
  // process result1
  const result2 = await ajax('/api/user/getOrder', { id })
  // process result2
  const result3 = await ajax('/api/user/getMessage', { id })
  // process result3
}
request(1)
</code></pre>
必须记住的是在函数上添加 <code>async</code> 关键字，从而可以在函数内使用 <code>await</code>，否则的话会报错。
 尽管 <code>async/await</code> 的书写方式很像同步代，但是这和同步代码是不同的。
 打个比方，执行一段很耗时的操作，同步的方式时 JS 会想，我在这等着你，你这个操作做完了我才能去做别的事。使用 <code>async/await</code> 时 JS 会想反正闲着也是闲着，我可以先把手头上的工作(主执行栈)停一停，看看有没有其他事情(回调队列或者它执行栈)可以做的。

<pre><code class="highlight-chroma">const asyncFunc = (n) => new Promise(res => etTimeout(() => res(n), 5000))

const call = async (n) =not found render function for node [type=NodeHTMLEntity, Tokens=]not found render function for node [type=NodeHTMLEntity, Tokens=>] {

const result = await asyncFunc(n)

console.log(result)

}

setTimeout(() =not found render function for node [type=NodeHTMLEntity, Tokens=>]ot found render function for node [type=NodeHTMLEntity, Tokens=>] {

console.log('event call!!')

}, 2000)

call(50)

```
// event call!!
// 50
</code></pre>

<p>那么这里可以引入一个问题。<br> 小张同学在看完这篇文章之后，希望使用 <code>async/awa
t</code> 改写计算斐波那契数列的函数，从而达到在程序计算时也可以执行其他执行栈的函数。小
同学的代码如下，你知道他错在哪里了吗？</p>
<pre><code class="highlight-chroma">/**
 * 小张希望的输出是：
 * event call!!
 * fib(50)的值
 * 但是运行时却不是这样的，而且程序还会卡死
 * 不是说async/await可以将函数变为异步吗？那执行结果会与预期不一致呢？
 */

const fib = (n) =not found render function for node [type=NodeHTMLEntity, Tokens=>]
ot found render function for node [type=NodeHTMLEntity, Tokens=>] n === 0 ? 1 : n === 1
 1 : fib(n - 1) + fib(n - 2)
const asyncFunc = (n) =not found render function for node [type=NodeHTMLEntity, Tokens
>]not found render function for node [type=NodeHTMLEntity, Tokens=>] new Promise(res
=not found render function for node [type=NodeHTMLEntity, Tokens=>]not found render f
nction for node [type=NodeHTMLEntity, Tokens=>] res(fib(n)))

const call = async (n) =not found render function for node [type=NodeHTMLEntity, Tokens=
]not found render function for node [type=NodeHTMLEntity, Tokens=>] {
const result = await asyncFunc(n)
console.log(result)
}

setTimeout(() =not found render function for node [type=NodeHTMLEntity, Tokens=>]
ot found render function for node [type=NodeHTMLEntity, Tokens=>] {
console.log('event call!!')
}, 2000)

call(50)
</code></pre>

<h2 id="展望未来">展望未来</h2>
<p>或许管道流式操作可能成为异步处理方式的新宠？<br> <img src="https://static.hacpai.com/
mages/img-loading.svg" width="400" data-src="https://image.hduzplus.xyz/image/db9977ec
19a8-474c-b37a-16a965e3443c.png"></p>
<h4 id="推荐阅读">推荐阅读</h4>
<ol>
 <li><a href="https://link.hacpai.com/forward?goto=https%3A%2F%2Fblog.hduzplus.xyz%2F
s-event-loop" target="_blank" rel="nofollow ugc">说说 Event Loop - Jiahao.Zhang’s Blog</
></li>
 <li><a href="https://link.hacpai.com/forward?goto=https%3A%2F%2Fwww.liaoxuefeng.co
%2Fwiki%2F001434446689867b27157e896e74d51a89c25cc8b43bdb3000%2F001434500853
155e93fc16046d4bb7854943814c4f9dc2000" target="_blank" rel="nofollow ugc">Promise -
廖雪峰的官方网站</a></li>
```

Javascript 异步编程的 4 种方法 - 阮一峰的网络日志
ECMAScript 6 入门
Promise - JavaScript | MDN
