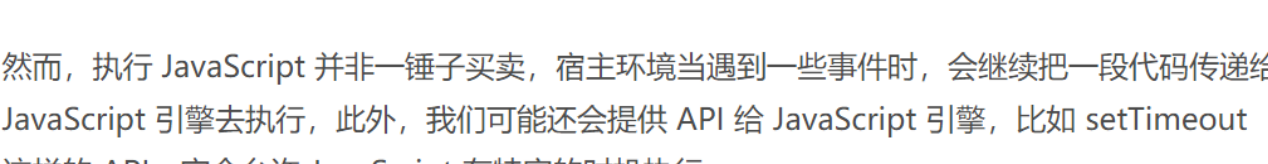


16 | JavaScript执行（一）：Promise里的代码为什么比setTimeout先执行？

winter 2019-02-23



你好，我是 winter。这一部分我们来讲一讲 JavaScript 的执行。

首先我们考虑一下，如果我们是浏览器或者 Node 的开发者，我们该如何使用 JavaScript 引擎。

当拿到一段 JavaScript 代码时，浏览器或者 Node 环境首先要做的就是；传递给 JavaScript 引擎，并且要求它去执行。

然而，执行 JavaScript 并非一锤子买卖，宿主环境当遇到一些事件时，会继续把一段代码传递给 JavaScript 引擎去执行，此外，我们可能还会提供 API 给 JavaScript 引擎，比如 setTimeout 这样的 API，它会允许 JavaScript 在特定的时机执行。

所以，我们首先应该形成一个感性的认知：一个 JavaScript 引擎会常驻于内存中，它等待着我们（宿主）把 JavaScript 代码或者函数传递给它执行。

在 ES3 和更早的版本中，JavaScript 本身还没有异步执行代码的能力，这也就意味着，宿主环境传递给 JavaScript 引擎一段代码，引擎就把代码直接顺次执行了，这个任务也就是宿主发起的任务。

但是，在 ES5 之后，JavaScript 引入了 Promise，这样，不需要浏览器的安排，JavaScript 引擎本身也可以发起任务了。

由于我们这里主要讲 JavaScript 语言，那么采纳 JSC 引擎的术语，我们把宿主发起的任务称为宏观任务，把 JavaScript 引擎发起的任务称为微观任务。

宏观和微观任务

JavaScript 引擎等待宿主环境分配宏观任务，在操作系统中，通常等待的行为都是一个事件循环，所以在 Node 术语中，也会把这个部分称为事件循环。

不过，术语本身并非我们需要重点讨论的内容，我们在这里把重点放在事件循环的原理上。在底层的 C/C++ 代码中，这个事件循环是一个跑在独立线程中的循环，我们用伪代码来表示，大概是这样的：

```
1 while(TRUE) {
2   r = wait();
3   execute(r);
4 }
5
```

我们可以看到，整个循环做的事情基本上就是反复“等待 - 执行”。当然，实际的代码中并没有这么简单，还要判断循环是否结束、宏观任务队列等逻辑，这里为了方便你理解，我就把这些都省略掉了。

这里每次的执行过程，其实都是一个宏观任务。我们可以大概理解：宏观任务的队列就相当于事件循环。

在宏观任务中，JavaScript 的 Promise 还会产生异步代码，JavaScript 必须保证这些异步代码在一个宏观任务中完成，因此，每个宏观任务中又包含了一个微观任务队列：



有了宏观任务和微观任务机制，我们就可以实现 JS 引擎级和宿主级的任务了，例如：Promise 永远在队列尾部添加微观任务。setTimeout 等宿主 API，则会添加宏观任务。

接下来，我们来详细介绍一下 Promise。

Promise

Promise 是 JavaScript 语言提供的一种标准化的异步管理方式，它的总体思想是，需要进行 io、等待或者其它异步操作的函数，不返回真实结果，而返回一个“承诺”，函数的调用方可以在合适的时机，选择等待这个承诺兑现（通过 Promise 的 then 方法的回调）。

Promise 的基本用法示例如下：

```
1 function sleep(duration) {
2   return new Promise(function(resolve, reject) {
3     setTimeout(resolve,duration);
4   });
5 }
6 sleep(1000).then( ()=> console.log("finished"));
7
```

这段代码定义了一个函数 sleep，它的作用是等候传入参数指定的时长。

Promise 的 then 回调是一个异步的执行过程，下面我们就来研究一下 Promise 函数中的执行顺序，我们来看一段代码示例：

```
1 var r = new Promise(function(resolve, reject){
2   console.log("a");
3   resolve()
4 });
5 r.then(() => console.log("c"));
6 console.log("b")
7
```

我们执行这段代码后，注意输出的顺序是 a b c。在进入 console.log(“b”)之前，毫无疑问 r 已经得到了 resolve，但是 Promise 的 resolve 始终是异步操作，所以 c 无法出现在 b 之前。

接下来我们试试跟 setTimeout 混用的 Promise。

在这段代码中，我设置了两段互不相干的异步操作：通过 setTimeout 执行 console.log(“d”)，通过 Promise 执行 console.log(“c”)。

```
1 var r = new Promise(function(resolve, reject){
2   console.log("a");
3   resolve()
4 });
5 setTimeout(()=>console.log("d"), 0)
6 r.then(() => console.log("c"));
7 console.log("b")
8
```

我们发现，不论代码顺序如何，d 必定发生在 c 之后，因为 Promise 产生的是 JavaScript 引擎内部的微任务，而 setTimeout 是浏览器 API，它产生宏任务。

为了理解微任务始终先于宏任务，我们设计一个实验：执行一个耗时 1 秒的 Promise。

```
1 setTimeout(()=>console.log("d"), 0)
2 var r = new Promise(function(resolve, reject){
3   resolve()
4 });
5 r.then(() => {
6   var begin = Date.now();
7   while(Date.now() - begin < 1000);
8   console.log("c1")
9   new Promise(function(resolve, reject){
10    resolve()
11  }).then(() => console.log("c2"))
12 });
13
```

这里我们强制了 1 秒的执行耗时，这样，我们可以确保任务 c2 是在 d 之后被添加到任务队列。

我们可以看到，即使耗时一秒的 c1 执行完毕，再 enqueue 的 c2，仍然先于 d 执行了，这很好地解释了微任务优先的原理。

通过一系列的实验，我们可以总结一下如何分析异步执行的顺序：

- 首先我们分析有多少个宏任务；
- 在每个宏任务中，分析有多少个微任务；
- 根据调用次序，确定宏任务中的微任务执行次序；
- 根据宏任务的触发规则和调用次序，确定宏任务的执行次序；

我们再来看一个稍微复杂的例子：

```
1 function sleep(duration) {
2   return new Promise(function(resolve, reject) {
3     console.log("a");
4     setTimeout(resolve,duration);
5   })
6 }
7 console.log("a");
8 sleep(5000).then(()=>console.log("c"))
9
```

这是一段非常常用的封装方法，利用 Promise 把 setTimeout 封装成可以用于异步的函数。

我们首先来看，setTimeout 把整个代码分割成了 2 个宏观任务，这里不论是 5 秒还是 0 秒，都是一样的。

第一个宏观任务中，包含了先后同步执行的 console.log(“a”)和 console.log(“b”)。

setTimeout 后，第二个宏观任务执行调用了 resolve，然后 then 中的代码异步得到执行，所以调用了 console.log(“c”)，最终输出的顺序才是：a b c。

Promise 是 JavaScript 中的一个定义，但是实际编写代码时，我们可以发现，它似乎并不比回调的方式书写更简单，但是从 ES6 开始，我们有了 async/await，这个语法改进跟 Promise 配合，能够有效地改善代码结构。

新特性：async/await

async/await 是 ES2016 新加入的特性，它提供了用 for、if 等代码结构来编写异步的方式。它的运行时基础是 Promise，面对这种比较新的特性，我们先来看一下基本用法。

async 函数必定返回 Promise，我们把所有返回 Promise 的函数都可以认为是异步函数。

async 函数是一种特殊语法，特征是在 function 关键字之前加上 async 关键字，这样，就定义了一个 async 函数，我们可以在其中使用 await 来等待一个 Promise。

```
1 function sleep(duration) {
2   return new Promise(function(resolve, reject) {
3     setTimeout(resolve,duration);
4   })
5 }
6 async function foo(){
7   console.log("a")
8   await sleep(2000)
9   console.log("b")
10 }
11
```

这段代码利用了我们之前定义的 sleep 函数。在异步函数 foo 中，我们调用 sleep。

async 函数强大之处在于，它是可以嵌套的。我们在定义了一批原子操作的情况下，可以利用 async 函数组合出新的 async 函数。

```
1 function sleep(duration) {
2   return new Promise(function(resolve, reject) {
3     setTimeout(resolve,duration);
4   })
5 }
6 async function foo(name){
7   await sleep(2000)
8   console.log(name)
9 }
10 async function foo2(){
11   await foo("a");
12   await foo("b");
13 }
14
```

这里 foo2 用 await 调用了两次异步函数 foo，可以看到，如果我们将 sleep 这样的异步操作放入某一个框架或者库中，使用者几乎不需要了解 Promise 的概念即可进行异步编程了。

此外，generator/iterator 也常常被跟异步一起来讲，我们必须说明 generator/iterator 并非异步代码，只是在缺少 async/await 的时候，一些框架（最著名的要数 co）使用这样的特性来模拟 async/await。

但是 generator 并非被设计成实现异步，所以有了 async/await 之后，generator/iterator 来模拟异步的方法应该被废弃。

结语

在今天的文章里，我们学习了 JavaScript 执行部分的知识，首先我们学习了 JavaScript 的宏观任务和微观任务相关的知识。我们把宿主发起的任务称为宏观任务，把 JavaScript 引擎发起的任务称为微观任务。许多的微观任务的队列组成了宏观任务。

除此之外，我们还展开介绍了用 Promise 来添加微观任务的方式，并且介绍了 async/await 这个语法的改进。

最后，留给你一个小练习：我们现在要实现一个红绿灯，把一个圆形 div 按照绿色 3 秒，黄色 1 秒，红色 2 秒循环改变背景色，你会怎样编写这个代码呢？欢迎你留言讨论。

重学前端

每天10分钟，重构你的前端知识体系

winter 主讲
前手书 海宝拉姆 负责

新版升级，点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。