



## iNoob's Notes

每一个你讨厌的现在，都有一个不努力的曾经

### Promise实现原理

📅 2019-02-12 | 📁 js | 👁 102

**Promise** 是 **es6** 引入的异步处理方案，让我们可以采用链式的写法注册回调函数，摆脱多层异步回调函数嵌套的情况，使代码更加简洁。而理解 **Promise** 内部实现原理也十分重要，我们可以从简单的模型开始，考虑不同的边界情况，一步一步的往最终结果实现。

#### 一个简单的雏形

如下我们可以新建一个 **Promise** 对象, 然后马上执行成功的回调。

```
1 var a = new Promise(function(resolve) {
2   resolve(1);
3 })
4
5 a.then(x => console.log(x))
```

可以看出，**Promise** 是一个构造函数，接收一个函数参数，其中函数参数的参数 **resolve** 在 **Promise** 构造函数内部实现。构造函数有一个 **then** 的方法，注册成功的回调，在调用 **resolve** 时执行。因此可以得到如下一个简单的模型：

```
1 function _Promise(fn) {
2   var self = this;
3   this.value = null;
4   this.callbacks = [];
5
6   this.then = function(onFulfilled) {
7     // 注册一个成功的回调函数
8     this.callbacks.push(onFulfilled);
```

```
9   }
10
11   function resolve(value) {
12     self.value = value;
13     // 让所有回调函数进入下一个事件循环执行
14     setTimeout(function(){
15       self.callbacks.forEach(function(callback) {
16         callback(value);
17       })
18     },0);
19   }
20
21   fn(resolve)
22 }
```

构造函数里面的属性 **value** 表示成功的最终值，**callbacks** 表示通过 **then** 注册的成功回调方法，类型是一个数组是因为 **Promise** 对象支持注册多个成功回调函数。在 **resolve** 中加入 **setTimeout** 延时是让所有回调函数在下一轮事件循环中执行，从而保证所有在当前执行队列的回调函数注册成功。

#### 引入状态

前面实现的雏形可以让当前执行队列的回调函数成功执行，但是在下一轮或者之后注册的回调函数将无效。比如：

```
1 a.then(x => console.log(x))
2
3 setTimeout(function(){
4   a.then(x => console.log(x+1))
5 }, 1000);
```

上面只会输出第一个回调结果。所以，我们需要引入一个状态属性 **state**，表示 **Promise** 对象当前的状态。当状态为 **pending** 的时候，注册的回调函数才压进 **callbacks** 中。当调用 **resolve** 后状态变为已解决 **fulfilled**，此时通过 **then** 注册的成功回调会马上执行。如下：

```
1 function _Promise(fn) {
2   var self = this;
3
4   this.state = 'pending';
5   this.value = null;
6   this.callbacks = [];
7
8   this.then = function(onFulfilled) {
9     if(this.state === 'pending') {
10       this.callbacks.push(onFulfilled);
11     }else {
12       onFulfilled(this.value);
```

```

13     }
14   }
15
16   function resolve(value) {
17     self.state = 'fulfilled';
18     self.value = value;
19     // 让所有回调函数进入下一个事件循环执行
20     setTimeout(function(){
21       self.callbacks.forEach(function(callback) {
22         callback(value);
23       })
24     },0);
25   }
26
27   fn(resolve)
28 }

```

## 链式调用

我们知道原生的 `Promise` 可以支持链式调用，如下：

```

1  var a = new Promise(function(resolve) {
2    resolve(1);
3  })
4
5  a.then(x => {
6    console.log(x);
7    return x+1;
8  }).then(x => console.log(x))

```

可以看出第一个 `Promise` 对象回调中返回的值会最为新对象回调的参数，相当于返回一个立即 `resovle(前者返回值)` 的新 `Promise` 对象，所以上面会输出1和2。

现在，我们就可以把 `then` 函数修改成返回一个新的 `Promise` 对象，并且和当前的 `Promise` 对象做关联。如下：

```

1  function _Promise(fn) {
2    var self = this;
3
4    this.state = 'pending';
5    this.value = null;
6    this.callbacks = [];
7
8    this.then = function(onFulfilled) {
9      // 返回一个新的Promise对象
10     return new _Promise(function(resolve) {
11       handleCallback({
12         onFulfilled: onFulfilled || null,

```

```

13     resolve: resolve // 让当前的promise对象和新的promise对象关联
14   })
15 })
16 }
17
18 function handleCallback(callback) {
19   if(self.state === 'pending') {
20     self.callbacks.push(callback);return;
21   }
22
23   var res = callback.onFulfilled(self.value);
24   // 调用新的promise对象的resolve
25   callback.resolve(res);
26 }
27
28 function resolve(value) {
29   self.state = 'fulfilled';
30   self.value = value;
31   // 让所有回调函数进入下一个事件循环执行
32   setTimeout(function(){
33     self.callbacks.forEach(function(callback) {
34       handleCallback(callback);
35     })
36   },0);
37 }
38
39 fn(resolve)
40 }

```

有上面代码可以看出，`handleCallback` 方法是关联两个就行 `Promise` 对象的关键，该方法的参数是一个对象，对象的 `onFulfilled` 属性是老 `Promise` 对象的回调函数，`resolve` 属性是新对象的构造函数的 `resolve` 方法，也可以说是新对象的 `resolve` 方法。因为构造函数的 `resolve` 函数是一个闭包，里面的 `self` 保存的是对应实例化的 `Promise` 对象。

当第一个对象的 `onFulfilled` 函数为空，直接把一个对象的终值 `value` 作为第二个对象的 `resolve` 参数。

```

1  var a = new Promise(function(resolve) {
2    resolve(2);
3  })
4
5  a.then().then(x => console.log(x)) // => 2

```

于是 `handleCallback` 函数修改成：

```

1  function handleCallback(callback) {
2    if(self.state === 'pending') {

```

```

3         self.callbacks.push(callback);return;
4     }
5
6     if(!callback.onFulfilled) {
7         callback.resolve(self.value);return;
8     }
9
10    var res = callback.onFulfilled(self.value);
11    // 调用新的promise对象的resolve
12    callback.resolve(res);
13 }

```

我们前面提到第一个对象的回调函数返回值等于第二个对象的 `resolve` 参数，它等同于下面形式：

```

1 var a = new Promise(function(resolve) {
2     resolve(1);
3 })
4
5 a.then(x => {
6     return new Promise(function(resolve){
7         resolve(x+1)
8     })
9 }).then(x => console.log(x))

```

于是要考虑调用第一个对象回调会返回 `thenable` 对象的情况，这个时候应该把由 `a.then()` 创建的对象的 `resolve` 对象这个 `thenable` 对象的成功回调，状态受到里面 `thenable` 对象的状态影响，所以终值始终等于这个 `thenable` 对象的终值。于是，`resolve` 修改成：

```

1 function resolve(endValue) {
2     if(endValue && (typeof endValue === 'object') && typeof endValue.then === 'function') {
3         // 让新的promise对象的resolve作为thenable对象的成功回调
4         endValue.then(resolve);
5         return;
6     }
7
8     self.state = 'fulfilled';
9     self.value = endValue;
10    // 让所有回调函数进入下一个事件循环执行
11    setTimeout(function(){
12        self.callbacks.forEach(function(callback) {
13            handleCallback(callback);
14        })
15    },0);
16 }

```

## 失败处理

和成功 `fulfilled` 的处理逻辑一样，我们引入失败的状态 `rejected` 和失败回调 `onRejected`。

```

1 function _Promise(fn) {
2     var self = this;
3
4     this.state = 'pending';
5     this.value = null;
6     this.callbacks = [];
7
8     this.then = function(onFulfilled, onRejected) {
9         // 返回一个新的Promise对象
10        return new _Promise(function(resolve, reject) {
11            handleCallback({
12                onFulfilled: onFulfilled || null,
13                onRejected: onRejected || null,
14                resolve: resolve,
15                reject: reject
16            })
17        })
18    }
19
20    function handleCallback(callback) {
21        if(self.state === 'pending') {
22            self.callbacks.push(callback);return;
23        }
24
25        var cb = self.state === 'fulfilled' ? callback.onFulfilled : callback.onRejected;
26        if(cb === null) {
27            cb = self.state === 'fulfilled' ? callback.resolve : callback.reject;
28            cb(self.value);
29            return;
30        }
31
32        // 加入try-catch防止执行回调出错
33        try {
34            var res = cb(self.value);
35            callback.resolve(res);
36        }catch(e) {
37            callback.reject(e);
38        }
39    }
40
41    function resolve(endValue) {
42        if(endValue && (typeof endValue === 'object') && typeof endValue.then === 'function') {
43            endValue.then(resolve, reject);
44            return;
45        }
46
47        self.state = 'fulfilled';
48        self.value = endValue;
49        excute();

```

```
50     }
51
52     function reject(reason) {
53         self.state = 'rejected';
54         self.value = reason;
55         excute();
56     }
57
58     function excute() {
59         // 让所有回调函数进入下一个事件循环执行
60         setTimeout(function(){
61             self.callbacks.forEach(function(callback) {
62                 handleCallback(callback);
63             })
64         },0);
65     }
66
67     fn(resolve, reject)
68 }
```

加入 `try-catch` 保证在执行回调出错的时候能捕捉得到。如果执行回调成功，新的 `Promise` 总是成功 `fulfilled` 的，不管你之前的 `Promise` 对象是调用 `resolve` 还是 `reject` 。

总结

理解 `Promise` 源码的关键点如下：

- `then` 函数在 `Promise` 为 `pending` 状态时为注册回调，统一压到一个回调数组，所以我们会发现上面的测试例子的 `callbacks` 都是空数组，然后在 `resolve` 或者 `reject` 时才会统一执行。在其他状态注册都会直接执行。
- `then` 函数返回一个新的 `Promise` 对象，两个对象通过 `resolve(前者对象的终值)` 关联起来。

参考文章

[promise/A+规范](#)

[30分钟，让你彻底明白Promise原理](#)

[# js](#) [# es6](#)

昵称

邮箱

网址(http://)

👍 / ≥v≤ o来啊，快活啊!

表情 | 预览

回复

2 评论

Anonymous

Chrome 58.0.3029.110

Windows 7

2019-04-24

454546545645

回复

Anonymous

Chrome 71.0.3578.98

Windows 10.0

2019-03-01

做第一个评论的人

回复

13728790850

Chrome 72.0.3626.119

Windows 10.0

2019-03-01

@Anonymous

🙏

回复

Anonymous

Chrome 58.0.3029.110

Windows 7

2019-04-24

@13728790850

, 1221

回复

