

# Promise 学习笔记

---

## 1. Promise 的理解和使用

---

### 1.1. Promise 是什么？

#### 1.1.1. 理解

1. 抽象表达：

- Promise 是一门新的技术（ES6 规范）
- Promise 是 JavaScript 中进行异步编程的**新解决方案**
  - 异步编程：fs 文件操作、数据库操作、AJAX、定时器
- 备注：旧方案时单纯使用回调函数

2. 具体表达：

- 从语法上来说：Promise 是一个构造函数
- 从功能上来说：Promise 对象用来封装一个异步操作并可以获取其成功或者失败的值
- promise 中既可以异步任务，也可以同步任务

#### 1.1.2. promise 的状态改变

##### 1.1.2.1. Promise 状态

实例对象中的一个属性：PromiseState，该属性有三种值：

- pending 未决定的
- resolved / fulfilled 成功
- rejected 失败

状态变化只会有两种情况：

1. pending 变为 resolved
2. pending 变为 rejected

说明：

- 只有这两种，且一个 promise 对象只能改变一次
- 无论变为成功还是失败，都会有一个结果数据
- 成功的结果数据一般称为 value，失败的结果数据一般称为 reason

#### 1.1.3. Promise 结果属性

##### 1.1.3.1. Promise 对象的值

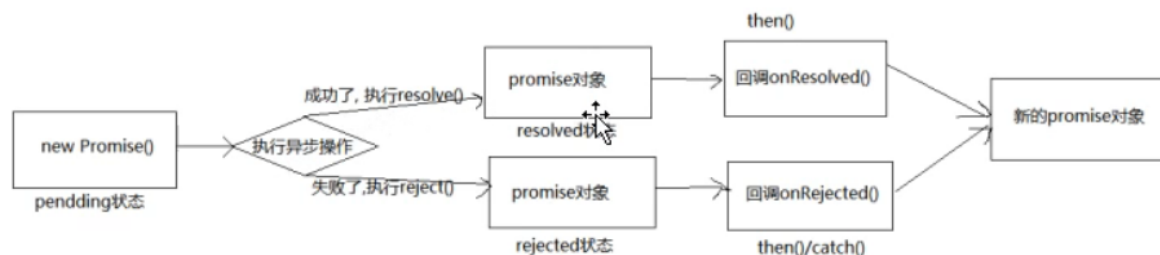
实例对象中的另一个属性：PromiseResult，保存着异步任务**成功/失败**的结果

修改该属性的值：

- resolve 函数
- reject 函数

修改之后在后续的 then 方法中就可以把这个值取出来，进行操作

### 1.1.4. Promise 的工作流程



## 1.2. 为什么要用 Promise?

### 1.2.1. 指定回调函数的方式更加灵活

1. 旧的：必须在启动异步任务前指定
2. promise：启动异步任务 => 返回 promise 对象 => 给 promise 对象绑定回调函数（甚至可以在异步任务结束后指定/多个）

### 1.2.2. 支持链式调用，可以解决回调地狱问题

1. 什么是回调地狱？  
回调函数嵌套调用，外部回调函数异步执行的结果是嵌套的回调执行的条件
2. 回调地狱的缺点？  
不便于阅读  
不便于异常处理

## 1.3. Promise 封装

### 1.3.1. fs 模块

```
1 function mineReadFile (path) {
2   return new Promise((resolve, reject) => {
3     // 读取文件
4     require('fs').readFile(path, (err, data) => {
5       // 判断
6       if (err) reject(err)
7       // 成功
8       resolve(data)
9     })
10  })
11 }
```

### 1.3.2. util.promisify方法

把回调函数风格的方法转变成 Promise 风格的方法

```
1 // 以读取文件为例
2
3 // 引入 util 模块
4 const util = require('util')
5 // 引入 fs 模块
6 const fs = require('fs')
```

```

7
8 // 返回一个新的函数
9 let minReadFile = util.promisify(fs.readFile);
10 minReadFile('./resource/content.txt').then(value => {
11     console.log(value.toString())
12 }, reason => {
13     console.log(reason)
14 })

```

### 1.3.3. 封装 Ajax 请求

```

1 function sendAJAX(url) {
2     return new Promise((resolve, reject) => {
3         const xhr = new XMLHttpRequest()
4         // 设置响应数据格式
5         xhr.responseType = 'json'
6         xhr.open('GET', url)
7         xhr.send();
8         // 处理结果
9         xhr.onreadystatechange = function () {
10             if (xhr.readyState === 4) {
11                 // 判断成功
12                 if (xhr.status >= 200 && xhr.status < 300) {
13                     // 成功的结果
14                     resolve(xhr.response)
15                 } else {
16                     reject(xhr.status)
17                 }
18             }
19         }
20     })
21 }
22
23 // 调用
24 sendAJAX('https://api.apiopen.top/getJoke')
25 .then(value => {
26     console.log(value)
27 }, reason => {
28     console.warn(reason)
29 })

```

## 1.4. 如何使用 Promise?

### 1.4.1. API

1. Promise 构造函数: `Promise(executor) {}`

- executor 函数: 执行器 (resolve, reject) => {}
  - resolve 函数: 内部定义成功时我们调用的函数 value => {}
  - reject 函数: 内部定义失败时我们调用的函数 reason => {}

说明: executor 会在 Promise 内部立即同步调用, 异步操作在执行器中执行

2. Promise.prototype.then 方法: (onResolved, onRejected) => {}

- onResolved 函数: 成功的回调函数 (value) => {}

- onRejected 函数：失败的回调函数 (reason) => {}

说明：指定用于得到成功 value 的成功回调和用于得到失败 reason 的失败回调

then 方法返回一个新的 promise 对象

### 3. Promise.prototype.catch 方法：(onRejected) => {}

- onRejected 函数：失败的回调函数 (reason) => {}

### 4. Promise.resolve 方法：(value) => {}

- value：成功的数据或 promise 对象
- 如果传递的参数为 非 promise 对象，则返回的结果为成功 promise 对象
- 如果传入的参数为 Promise 对象，则参数的结果决定了 resolve 的结果

说明：返回一个成功/失败的 promise 对象

```
1 let p1 = Promise.resolve(520)
2 let p2 = Promise.resolve(new Promise((resolve, reject) => {
3   resolve('OK')
4 }))) // 这时 p2 状态为成功，成功的值为 'OK'
```

### 5. Promise.reject 方法：(reason) => {}

- reason：失败的原因

说明：返回一个失败的 promise 对象

```
1 let p = Promise.reject(520) // 无论传入的是什么，返回的都是一个失败的promise 对象
2 // 传入什么，失败的结果就是什么
```

### 6. Promise.all 方法：(promises) => {}

- promises：包含 n 个 promise 的数组

说明：返回一个新的 promise，只有所有的 promise 都成功时才成功，只要有一个失败了就直接失败

- 成功的结果是每一个 promise 对象成功结果组成的数组（有顺序）
- 失败的结果是在这个数组中失败的那个 promise 对象失败的结果

```
1 let p1 = new Promise((resolve, reject) => {
2   resolve('OK')
3 })
4 let p2 = Promise.resolve('Success')
5 let p3 = Promise.resolve('Success')
6
7 const result = Promise.all([p1, p2, p3])
```

### 7. Promise.race 方法：(promises) => {}

- promises：包含 n 个 promise 的数组
- race：赛跑/比赛

说明：返回一个新的 promise，第一个完成的 promise 的结果状态就是最终的结果状态

```

1 let p1 = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('OK')
4   }, 1000)
5 })
6 let p2 = Promise.resolve('Success')
7 let p3 = Promise.resolve('Success')
8
9 const result = Promise.race([p1, p2, p3]) // =>结果为 p2 的结果，因为p2 先改
    变状态

```

### 1.4.2. promise 的几个关键问题

#### 1. 如何改变 promise 的状态?

- resolve(value): 如果当前是 pending 就会变为 resolved
- reject(reason): 如果当前是 pending 就会变为 rejected
- 抛出异常: 如果当前是 pending 就会变为 rejected

#### 2. 一个 promise 指定 (then方法) 多个成功/失败回调函数, 都会调用吗?

当 promise 改变为对应状态时都会调用

```

1 let p = new Promise((resolve, reject) => {
2   resolve('ok') // 这里状态改变了，所以下边两个回调都会执行，如果状态不改变，下面
    的回调都不执行
3 })
4
5 // 指定回调 - 1
6 p.then(value => {
7   console.log(value)
8 })
9
10 // 指定回调 - 2
11 p.then(value => {
12   alert(value)
13 })
14

```

#### 3. 改变 promise 状态和指定回调函数谁先谁后?

问题简单描述: promise 代码在运行时, resolve/reject改变状态先执行, 还是 then 方法指定回调先执行?

- 都有可能, 正常情况下是先指定回调再改变状态, 但也可以先改变状态再指定回调
  - 当执行器函数中的任务是一个同步任务(直接调 resolve()/reject()) 的时候, 先改变 promise 状态, 再去指定回调函数
  - 当执行器函数中的任务是一个异步任务的时候, then 方法先执行(指定回调), 改变状态后执行

```

1 // 这时是then 方法先执行(指定回调)，改变状态后执行
2 let p = new Promise((resolve, reject) => {
3   setTimeout(() => {
4     resolve('OK')
5   }, 1000)
6 })
7
8 p.then(value => {
9   console.log(value)
10 })

```

- 如何先改状态再指定回调?
    - 在执行器中直接调用 resolve()/reject()
    - 延迟更长时间才调用 then()
  - 什么时候才能得到数据(回调函数什么时候执行)?
    - 如果先指定的回调，那当状态发生改变时(调用resolve()/reject())时，回调函数就会调用，得到数据
    - 如果先改变的状态，那当指定函数时(then 方法)，回调函数就会调用，得到数据
4. promise.then() 返回的新 promise 的结果状态有什么决定?
- 简单表达：由 then() 指定的回调函数执行的结果决定
  - 详细表达：
    - 如果抛出异常，新 promise 变为 rejected，reason 为抛出的异常
    - 如果返回的是非 promise 的任意值，新 promise 变为 resolved，value 为返回的值
    - 如果返回的是另一个新的 promise，此 promise 的结果就会成为新 promise 的结果
5. promise 如何串联多个操作任务?
- promise 的 then() 返回一个新的 promise，可以看成 then() 的链式调用
  - 通过 then 的链式调用串联多个同步/异步任务
6. promise 异常穿透?
- 当使用 promise 的 then 链式调用时，可以在最后指定失败的回调，
  - 前面任何操作除了异常，都会传到最后失败的回调中处理
7. 中断 promise 链
- 当使用 promise 的 then 链式调用时，在中间中断，不再调用后面的回调函数
  - 办法：在回调函数中返回一个 pending 状态的 promise 对象

```

1 let p = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('OK')
4   }, 1000)
5 })
6
7 p.then(value => {
8   console.log(111)
9   return new Promise(() => {})
10 }).then(value => {
11   console.log(222)
12 })

```

## 1.5. 手写 Promise

### 1.5.1. 函数方式

```
1  /*
2  自定义 Promise
3  */
4
5  function Promise(executor) {
6      // 添加属性
7      this.PromiseState = 'pending'
8      this.PromiseResult = null
9      // 声明属性 因为实例对象不能直接调用onResolve跟onReject 所以下面then中需要先保
      存在callback里面
10     this.callbacks = []
11     // 保存实例对象的 this 的值
12     const self = this // 常见的变量名有self _this that
13
14     // resolve 函数
15     function resolve(data) {
16         // 判断状态
17         if (self.PromiseState !== 'pending') return
18         // console.log(this) => 这里的this指向window, 下面用this的话时直接修改
            的window
19         // 1. 修改对象的状态 (PromiseState)
20         self.PromiseState = 'fulfilled'
21         // 2. 设置对象结果值 (PromiseResult)
22         self.PromiseResult = data
23         // 调用成功的回调函数
24         setTimeout(() => {
25             self.callbacks.forEach((item) => {
26                 item.onResolved(data)
27             })
28         })
29     }
30
31     // reject 函数
32     function reject(data) {
33         // 判断状态
34         if (self.PromiseState !== 'pending') return
35         // 1. 修改对象的状态 (PromiseState)
36         self.PromiseState = 'rejected'
37         // 2. 设置对象结果值 (PromiseResult)
38         self.PromiseResult = data
39         // 调用失败的回调函数
40         setTimeout(() => {
41             self.callbacks.forEach((item) => {
42                 item.onRejected(data)
43             })
44         })
45     }
46     try {
47         // 同步调用【执行器函数】
48         executor(resolve, reject)
49     } catch (e) {
50         // 修改 promise 对象状态
51         reject(e)
52     }
```

```

52     }
53 }
54
55 // 添加 then 方法
56 Promise.prototype.then = function (onResolved, onRejected) {
57     const self = this
58     // 判断回调函数参数
59     if (typeof onRejected !== 'function') {
60         onRejected = (reason) => {
61             throw reason
62         }
63     }
64     if (typeof onResolved !== 'function') {
65         onResolved = (value) => value
66     }
67     return new Promise((resolve, reject) => {
68         // 封装函数
69         function callback(type) {
70             try {
71                 // 获取回调函数的执行结果
72                 let result = type(self.PromiseResult)
73                 // 判断
74                 if (result instanceof Promise) {
75                     result.then(
76                         (v) => {
77                             resolve(v)
78                         },
79                         (r) => {
80                             reject(r)
81                         }
82                     )
83                 } else {
84                     // 结果的对象状态为 【成功】
85                     resolve(result)
86                 }
87             } catch (e) {
88                 reject(e)
89             }
90         }
91         // 调用回调函数 根据 PromiseState 去调用
92         if (this.PromiseState === 'fulfilled') {
93             setTimeout(() => {
94                 callback(onResolved)
95             })
96         }
97         if (this.PromiseState === 'rejected') {
98             setTimeout(() => {
99                 callback(onRejected)
100             })
101         }
102         // 判断 pending 状态
103         if (this.PromiseState === 'pending') {
104             // 保存回调函数
105             this.callbacks.push({
106                 onResolved: function () {
107                     callback(onResolved)
108                 },
109                 onRejected: function () {

```



```

110         callback(onRejected)
111     },
112 })
113 }
114 })
115 }
116
117 // 添加 catch 方法
118 Promise.prototype.catch = function (onRejected) {
119     return this.then(undefined, onRejected)
120 }
121
122 // 添加 resolve 方法
123 Promise.resolve = function (value) {
124     return new Promise((resolve, reject) => {
125         if (value instanceof Promise) {
126             value.then(
127                 (v) => {
128                     resolve(v)
129                 },
130                 (r) => {
131                     reject(r)
132                 }
133             )
134         } else {
135             // 状态设置为成功
136             resolve(value)
137         }
138     })
139 }
140
141 // 添加 reject 方法
142 Promise.reject = function (reason) {
143     return new Promise((resolve, reject) => {
144         reject(reason)
145     })
146 }
147
148 // 添加 all 方法
149 Promise.all = function (promises) {
150     // 声明变量
151     let count = 0 // 计数
152     let arr = [] // 结果数组
153     // 遍历
154     return new Promise((resolve, reject) => {
155         for (let i = 0; i < promises.length; i++) {
156             promises[i].then(
157                 (v) => {
158                     // 得知对象的状态是成功
159                     // 每个promise对象成功都加 1
160                     count++
161                     // 将当前每个promise对象成功的结果都存入到数组中
162                     arr[i] = v
163                     // 判断
164                     if (count === promises.length) {
165                         // 修改状态
166                         resolve(arr)
167                     }
168                 }
169             )
170         }
171     })
172 }

```

```

168         },
169         (r) => {
170             reject(r)
171         }
172     )
173 }
174 })
175 }
176
177 // 添加 race 方法
178 Promise.race = function (promises) {
179     return new Promise((resolve, reject) => {
180         for (var i = 0; i < promises.length; i++) {
181             promises[i].then(
182                 (v) => {
183                     // 修改返回对象的状态为成功
184                     resolve(v)
185                 },
186                 (r) => {
187                     // 修改返回对象的状态为成功
188                     reject(r)
189                 }
190             )
191         }
192     })
193 }
194

```

### 1.5.2. 类的方式 (封装成class)

```

1  /*
2  自定义 Promise
3  */
4
5  // 封装成类
6  class Promise {
7      // 构造方法
8      constructor(executor) {
9          // 添加属性
10         this.PromiseState = 'pending'
11         this.PromiseResult = null
12         // 声明属性 因为实例对象不能直接调用onResolve跟onReject 所以下面then中需要
           先保存在callback里面
13         this.callbacks = []
14         // 保存实例对象的 this 的值
15         const self = this // 常见的变量名有self _this that
16
17         // resolve 函数
18         function resolve(data) {
19             // 判断状态
20             if (self.PromiseState !== 'pending') return
21             // console.log(this) => 这里的this指向window, 下面用this的话时直接
           修改的window
22             // 1. 修改对象的状态 (PromiseState)
23             self.PromiseState = 'fulfilled'
24             // 2. 设置对象结果值 (PromiseResult)
25             self.PromiseResult = data

```

```

26         // 调用成功的回调函数
27         setTimeout(() => {
28             self.callbacks.forEach((item) => {
29                 item.onResolved(data)
30             })
31         })
32     }
33
34     // reject 函数
35     function reject(data) {
36         // 判断状态
37         if (self.PromiseState !== 'pending') return
38         // 1. 修改对象的状态 (PromiseState)
39         self.PromiseState = 'rejected'
40         // 2. 设置对象结果值 (PromiseResult)
41         self.PromiseResult = data
42         // 调用失败的回调函数
43         setTimeout(() => {
44             self.callbacks.forEach((item) => {
45                 item.onRejected(data)
46             })
47         })
48     }
49     try {
50         // 同步调用【执行器函数】
51         executor(resolve, reject)
52     } catch (e) {
53         // 修改 promise 对象状态
54         reject(e)
55     }
56 }
57
58 // then 方法封装
59 then(onResolved, onRejected) {
60     const self = this
61     // 判断回调函数参数
62     if (typeof onRejected !== 'function') {
63         onRejected = (reason) => {
64             throw reason
65         }
66     }
67     if (typeof onResolved !== 'function') {
68         onResolved = (value) => value
69     }
70     return new Promise((resolve, reject) => {
71         // 封装函数
72         function callback(type) {
73             try {
74                 // 获取回调函数的执行结果
75                 let result = type(self.PromiseResult)
76                 // 判断
77                 if (result instanceof Promise) {
78                     result.then(
79                         (v) => {
80                             resolve(v)
81                         },
82                         (r) => {
83                             reject(r)

```

```

84         }
85     )
86     } else {
87         // 结果的对象状态为 【成功】
88         resolve(result)
89     }
90     } catch (e) {
91         reject(e)
92     }
93 }
94 // 调用回调函数 根据 PromiseState 去调用
95 if (this.PromiseState === 'fulfilled') {
96     setTimeout(() => {
97         callback(onResolved)
98     })
99 }
100 if (this.PromiseState === 'rejected') {
101     setTimeout(() => {
102         callback(onRejected)
103     })
104 }
105 // 判断 pending 状态
106 if (this.PromiseState === 'pending') {
107     // 保存回调函数
108     this.callbacks.push({
109         onResolved: function () {
110             callback(onResolved)
111         },
112         onRejected: function () {
113             callback(onRejected)
114         },
115     })
116 }
117 })
118 }
119
120 // catch 方法
121 catch(onRejected) {
122     return this.then(undefined, onRejected)
123 }
124
125 // resolve 方法
126 static resolve(value) {
127     return new Promise((resolve, reject) => {
128         if (value instanceof Promise) {
129             value.then(
130                 (v) => {
131                     resolve(v)
132                 },
133                 (r) => {
134                     reject(r)
135                 }
136             )
137         } else {
138             // 状态设置为成功
139             resolve(value)
140         }
141     })

```

```

142     }
143
144     // reject 方法
145     static reject(reason) {
146         return new Promise((resolve, reject) => {
147             reject(reason)
148         })
149     }
150
151     // all 方法
152     static all(promises) {
153         // 声明变量
154         let count = 0 // 计数
155         let arr = [] // 结果数组
156         // 遍历
157         return new Promise((resolve, reject) => {
158             for (let i = 0; i < promises.length; i++) {
159                 promises[i].then(
160                     (v) => {
161                         // 得知对象的状态是成功
162                         // 每个promise对象成功都加 1
163                         count++
164                         // 将当前每个promise对象成功的结果都存入到数组中
165                         arr[i] = v
166                         // 判断
167                         if (count === promises.length) {
168                             // 修改状态
169                             resolve(arr)
170                         }
171                     },
172                     (r) => {
173                         reject(r)
174                     }
175                 )
176             }
177         })
178     }
179
180     //race 方法
181     static race(promises) {
182         return new Promise((resolve, reject) => {
183             for (var i = 0; i < promises.length; i++) {
184                 promises[i].then(
185                     (v) => {
186                         // 修改返回对象的状态为成功
187                         resolve(v)
188                     },
189                     (r) => {
190                         // 修改返回对象的状态为成功
191                         reject(r)
192                     }
193                 )
194             }
195         })
196     }
197 }
198

```

## 1.6. async 与 await

### 1.6.1. mdn 文档

[https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/async\\_function](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/async_function)

<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/await>

### 1.6.2. async 函数

1. 函数的返回值为 promise 对象
2. promise 对象的结果由 async 函数执行的返回值决定

### 1.6.3. await 表达式

1. await 右侧的表达式一般为 promise 对象，但也可以是其它的值
2. 如果表达式是 promise 对象，await 返回的是 promise 成功的值
3. 如果表达式是其它值，直接将此值作为 await 的返回值

### 1.6.4. 注意

1. await 必须写在 async 函数中，但 async 函数中可以没有 await
2. 如果 await 的 promise 失败了，就会抛出异常，需要 try...catch 捕获处理

### 1.6.5. async 与 await 结合

```
1 // resource 1.html 2.html 3.html
2
3 const fs = require('fs')
4
5 // 回调函数的方式
6 fs.readFile('./resource/1.html', (err, data1) => {
7   if (err) throw err
8   fs.readFile('./resource/2.html', (err, data2) => {
9     if (err) throw err
10    fs.readFile('./resource/3.html', (err, data3) => {
11      if (err) throw err
12      console.log(data1 + data2 + data3)
13    })
14  })
15 })
```

```
1 // resource 1.html 2.html 3.html
2
3 const fs = require('fs')
4 const util = require('util')
5 const readFile = util.promisify(fs.readFile)
6
7 // async 与 await 结合
8 async function main() {
9   try {
10     // 读取第一个文件的内容
11     let data1 = await readFile('./resource/1.html')
12     let data2 = await readFile('./resource/2.html')
13     let data3 = await readFile('./resource/3.html')
```

```
14
15     console.log(data1 + data2 + data3)
16 }catch(e) {
17     console.log(e)
18 }
19 }
20 main()
```

```
1 // async 与 await 结合发送 Ajax 请求
2 function sendAJAX(url) {
3     return new Promise((resolve, reject) => {
4         const xhr = new XMLHttpRequest()
5         // 设置响应数据格式
6         xhr.responseType = 'json'
7         xhr.open('GET', url)
8         xhr.send();
9         // 处理结果
10        xhr.onreadystatechange = function () {
11            if (xhr.readyState === 4) {
12                // 判断成功
13                if (xhr.status >= 200 && xhr.status < 300) {
14                    // 成功的结果
15                    resolve(xhr.response)
16                } else {
17                    reject(xhr.status)
18                }
19            }
20        }
21    })
22 }
23
24 // 段子接口地址: https://api.apiopen.top/getJoke
25 let btn = document.querySelector('#btn')
26
27 btn.addEventListener('click', async function () {
28     // 获取段子信息
29     let duanzi = await sendAJAX('https://api.apiopen.top/getJoke')
30     console.log(duanzi)
31 })
```