

Promise的介绍和使用

1. 构造函数
2. 常见的报错
 - 2.1 try...catch 解决语法报错阻塞
 - 2.2 异步回调和同步回调
3. Promise 的介绍
 - 3.1 正确的使用
 - 3.2 Promise 的封装
 - 3.3 then 方法的两个形参
 - 3.4 经典面试题
 - 3.5 封装请求
4. 回调地狱
 - 4.1 优化代码
 - 4.2 最终的优化1
5. Promise 的实例方法
 - 5.1 catch 方法
 - 5.2 finally 方法
6. Promise 的静态方法
 - 6.1 Promise.resolve()
 - 6.2 Promise.reject()
 - 6.3 Promise.all()
 - 6.4 Promise.race()
7. 注意事项
 - 7.1 改变状态的3种行为
 - 7.2 promise 的多个回调
 - 7.3 状态和回调的先后使用
 - 7.4 then 方法的返回状态
 - 7.5 异常穿透和断链
8. 手工封装一个 Promise (熟悉)

- 8.1 基本封装
- 8.2 处理状态
- 8.3 不可逆的操作
- 8.4 封装then方法的回调
- 8.5 优化封装then 方法的回调
- 8.6 处理then方法的执行为异步的操作
- 8.7 添加实例的 catch 方法
- 8.8 静态方法 resolve 和 reject
- 8.9 静态方法 all
- 8.10 静态方法 race

ES6 更新了一种异步的解决方案。

1. 构造函数

既然是首字母大写，那么就是构造函数，使用 **new** 关键字调用

构造函数有**静态**属性和方法，也有**实例**属性和方法，

JavaScript | 复制代码

```
1 // Array 的静态方法
2 Array.isArray(arr); // 判断参数是否为数组
3
4 // Array 的实例方法
5 arr.push(5); // 给数组末尾增加元素
```

2. 常见的报错

红色的才是报错，所有的报错也是方法产生的，都归属于 Error 构造函数

```
1 // 1. 常见的报错
2 // Uncaught Error:
3 // throw 代表抛出的意思
4 // 报错之后, 后续的代码不执行
5
6 // 1.1 顶级的报错构造函数
7 // throw new Error('报错了! ');
8 // Uncaught 报错了?
9 // throw '报错了? ';
10
11 // 1.1 引用错误
12 // Uncaught ReferenceError
13 // console.log(num);
14 // const num = 10;
15
16 // 1.2 类型错误
17 // Uncaught TypeError
18 // num = 100;
19
20 // 1.3 范围错误
21 // Uncaught RangeError
22 // new Array(-1);
23
24 // 1.4 语法错误
25 // Uncaught SyntaxError
26 // var 3x = 100;
27
28 // throw new TypeError('类型错误');
29 // throw new SyntaxError('hello, your code is no ok!');
30 // console.log(100);
```

2.1 try...catch 解决语法报错阻塞

```
1 // 1. 使用 try...catch 解决报错的问题
2 // 核心: 解决报错之后的阻塞
3 const num = 10;
4 // 传统写法会阻塞代码的执行
5 // num = 20;
6 // 1.1 使用 try...catch 的语法
7 try {
8     // 要执行的语句
9     num = 20;
10 } catch (error) {
11     // 捕获了错误之后执行的语句
12     // 自动接收形参, 是捕获到的错误类型和错误内容!
13     console.log(error);
14 }
15 console.log('最后要执行的代码!', num);
16
17
18 // 2. 一些失误的封装
19 function getData() {
20     const num = Math.floor(Math.random() * 30 + 3);
21     if (num % 2 === 0) {
22         console.log('是偶数, 可以做处理。');
23     } else {
24         throw new RangeError('不能给奇数啊兄嘚!')
25         // throw 'hahaha';
26     }
27 }
28
29 // 2.1 try...catch 捕获错误
30 try {
31     getData();
32 } catch (err) {
33     // 这些错误提示, 可以使用一些弹窗组件完成
34     console.log(err.message); // 错误提示的文本
35     console.log(err.stack);
36     console.log(err.name);
37 }
38 console.log('哈哈哈, 今天天气真不错。');
```

2.2 异步回调和同步回调

```
1 // 回调函数分为两种
2 const arr = [1, 2, 3, 4];
3 arr.forEach(item => {
4     console.log(item);
5 })
6 console.log('ok');
7
8 // 2. 异步回调
9 // setTimeout(() => {
10 //     console.log(1);
11 // }, 0)
12 // console.log(2);
13
14 // 目前来说，异步任务，就3种
15 // 事件，定时器，xhr请求
16 // 服务器端：fs模块中的api... 连接mongodb数据库
17
18 // 事件队列的事儿
19 // 异步任务的回调会进入事件队列 Event Queue
20 // 同步代码执行结束之后，根据事件循环 Event Loop，找到对应的回调执行
```

3. Promise 的介绍

Promise：承诺，预言；

来自于 ES6 的新语法，**用于解决常见的回调问题，异步管理问题。**



重要程度：

重要的意义，后期的很多方法都是基于 Promise 封装过的，原理不清楚就无法自如的调用和解析，也需要经常能手动封装一些关于 Promise 的东西。

```
1 // 1. 创建 Promise 对象
2 // 有且仅有一个参数，为回调函数
3 const a1 = new Promise(function () { })
4
5 // 得到一个 pending 的状态
6 // pending: 等待，期待发生的；
7 console.log(a1);
```

3.1 正确的使用

```
1 // 1.1 正确的使用 Promise 的姿势
2 const a1 = new Promise(function (resolve, reject) {
3     // resolve reject 属于 Promise 回调的形参
4     // 这俩形参，都是方法，直接加小括号 () 调用
5     // resolve 代表 成功，可以执行的意思
6     // reject 代表 失败，拒绝考虑，不录用的意思
7
8     // 执行了这个函数之后，状态会改变为 fulfilled => 应验了
9     // resolve();
10
11     // 执行了这个函数之后，状态会改变为 rejected => 失败了
12     // reject();
13 })
14 // console.log(a1);
15
16 // 状态1: 期待的状态: pending
17 // 状态2: 应验的状态: fulfilled
18 // 状态3: 失败的状态: rejected
19
20 // 状态1可操作性，状态2和状态3不可逆
21
22 // 只有2种结果
23 // 要么就是 resolve 要么就是 reject
```

3.2 Promise 的封装

```
1 // 1. 封装的业务
2 function getData() {
3     // 1.1 基于Promise的容器
4     // const a1 = new Promise((resolve, reject) => {
5     return new Promise((resolve, reject) => {
6         const num = Math.floor(Math.random() * 30 + 3);
7         if (num % 2 === 0) {
8             // console.log('是偶数，可以做处理。');
9             // 1.2 成功
10            resolve('是偶数，可以做处理。');
11        } else {
12            // throw new RangeError('不能给奇数啊兄嘚!');
13            // 1.3 失败
14            reject('不能给奇数啊兄嘚!');
15        }
16    })
17    // 1.3 返回 promise 的结果
18    // return a1;
19 }
20
21
22 // 2. 其实，promise就是把之前应该写在回调函数里面的业务
23 // 可以放出来写了，还可以分不同的状态来写
24 // console.log(getData());
25 const p1 = getData();
26
27 // 3. p1 是实例对象，实例对象有俩方法
28 // then 处理成功 catch 捕获失败
29 p1.then(function (data) {
30     console.log('成功了! ---', data)
31 })
32 p1.catch(function (err) {
33     console.log('失败了，报错了----', err);
34 })
35
36 // promise 代码内部报错不会阻塞执行
37 console.log(1100);
```

3.3 then 方法的两个形参

```
1 // 1. Promise 实例对象回调的详解
2 function fn() {
3     // 1.1 返回 Promise 的实例
4     return new Promise((resolve, reject) => {
5         // 1.2 resolve 代表成功的回调, reject 代表失败的回调
6         // resolve();
7         // resolve(100);
8         // reject();
9         // reject('网络错误! ');
10        reject(new Error('网络不稳定! '))
11    })
12 }
13 // 2. 返回的 Promise 实例叫做 p1
14 const p1 = fn();
15 // 2.1 使用 p1 的实例方法
16 // 参数1: 成功的回调函数体
17 // 参数2: 失败的回调函数体
18 p1.then(
19     // resolve 执行这个, 会传参
20     data => {
21         console.log('成功了, 非常好. => ', data);
22     },
23     // reject 执行这个, 会传入错误信息
24     reason => {
25         console.log('sorry, 失败了. => ', reason.message);
26     }
27 )
```



```
1 // 1.1 封装一个业务
2 function random() {
3     // 1.2 返回一个 promise 的实例
4     return new Promise((resolve, reject) => {
5         // 1.3 异步的耗时的任务
6         setTimeout(function () {
7             const num = Math.floor(Math.random() * 30 + 3);
8             // 三目运算符
9             num % 2 ? reject(new TypeError('err odd')) : resolve('ok even'
10         );
11         }, 1500)
12     })
13 }
14 // 2. 获得实例对象
15 const p1 = random();
16 // 2.1 实例的回调，是在内部结果产生之后才会执行
17 p1.then(
18     data => {
19         console.log(data);
20     },
21     reason => {
22         console.log(reason.message);
23     }
24 )
25 // 构造函数中，必须给定状态和最后的结果，不然then中的回调一个都不会触发
```

3.4 经典面试题

```
1 // 1. 要求使用 Promise 封装, 使得代码完整
2 let num = 10;
3 // 1.1 定义函数
4 function fn() {
5     // 1.2 返回 Promise 的实例
6     return new Promise((resolve, reject) => {
7         // 1.3 开始异步任务
8         setTimeout(function () {
9             num = '10086';
10            // 1.4 指定状态
11            resolve(num);
12        }, 1500);
13    })
14 }
15 // 就是需要在定时器的外面打印 num
16 // 链式调用
17 fn()
18 .then(data => {
19     console.log(data);
20 })
```

3.5 封装请求

```
1 // 1.1 封装函数
2 function getData(url) {
3     // 1.2 返回一个 promise 的实例对象
4     return new Promise((resolve, reject) => {
5         $.ajax({
6             url,
7             success(res) {
8                 // 成功之后，把数据传入进来
9                 res.code === 200 ? resolve(res) : reject('请求失败');
10            },
11            error(err) {
12                // 错误处理在这边
13                reject(err);
14            }
15        })
16    })
17 }
18 // 2. 创建实例
19 getData('http://127.0.0.1:3000/demo')
20     .then(
21     data => {
22         console.log(1111, data);
23     },
24     reason => {
25         console.log(2222, reason);
26     }
27 )
```

4. 回调地狱

回调本身来说没有什么问题，也可以解决一些常见的业务

对于一些有经验的开发者来说，回调嵌套也是经常所用到的方式之一

但是在有些封装的方法中，有大量的回调嵌套，并且不方便在封装的内部函数中完成业务，所以需要在调用这个封装的方法之后操作业务。

所以基于这个考虑，才使用了 Promise

主要的特点： 在开启异步任务的同时，保持队列。

```
1 // 1. 引入模块
2 const fs = require('fs');
3
4 // 2. 读取文件
5
6 // 2.1 同步代码不会影响顺序问题
7 // console.log(fs.readFileSync('./data/01.txt', 'utf8'));
8 // console.log(fs.readFileSync('./data/02.txt', 'utf8'));
9 // console.log(fs.readFileSync('./data/03.txt', 'utf8'));
10 // console.log(fs.readFileSync('./data/04.txt', 'utf8'));
11
12 // 2.2 异步读取的任务
13 // 会引起回调地狱的问题，问题虽然能解决，但是不美观，不好维护
14 fs.readFile('./data/01.txt', 'utf8', (err, data) => {
15     if (err) throw err;
16     console.log(data);
17     // 读取01
18     fs.readFile('./data/02.txt', 'utf8', (err, data) => {
19         if (err) throw err;
20         console.log(data);
21         // 读取02
22         fs.readFile('./data/03.txt', 'utf8', (err, data) => {
23             if (err) throw err;
24             console.log(data);
25             // 读取03
26             fs.readFile('./data/04.txt', 'utf8', (err, data) => {
27                 if (err) throw err;
28                 console.log(data);
29                 // 读取04
30             })
31         })
32     })
33 })
34
35 // 不会阻塞同步的代码
36 console.log(100);
37
38
39
```

4.1 优化代码

```
1 // 1. 引入模块
2 const fs = require('fs');
3
4 // 2. 使用 Promise 来解决问题
5 function readFiles(path) {
6     return new Promise((resolve, reject) => {
7         // 2.1 执行异步任务
8         fs.readFile(path, 'utf8', (err, data) => {
9             if (err) throw err;
10            // 成功
11            resolve(data);
12        })
13    })
14 }
15
16 readFiles('./data/01.txt')
17     .then(data => {
18         console.log(data);
19         // 读取了之后, 再来读取第二个文件
20         return readFiles('./data/02.txt')
21     })
22     .then(data => {
23         console.log(data);
24         // 读取了之后, 再来读取第三个文件
25         return readFiles('./data/03.txt')
26     })
27     .then(data => {
28         console.log(data);
29         // 读取了之后, 再来读取四个文件
30         return readFiles('./data/04.txt')
31     })
32     .then(data => {
33         console.log(data);
34     })
35
36
37 console.log(100);
```

4.2 最终的优化1

提前使用到了新语法 `async / await`

```
1 // 1. 引入模块
2 const fs = require('fs');
3
4 // 2. 使用 Promise 来解决问题
5 function readFiles(path) {
6     return new Promise((resolve, reject) => {
7         // 2.1 执行异步任务
8         fs.readFile(path, 'utf8', (err, data) => {
9             if (err) throw err;
10            // 成功
11            resolve(data);
12        })
13    })
14 }
15
16 // 3. ES8 中更新了一个语法糖 async / await
17 // 专门用于管理基于 Promise 封装的异步
18 async function fn() {
19     const txt01 = await readFiles('./data/01.txt');
20     console.log(txt01);
21
22     const txt02 = await readFiles('./data/02.txt');
23     console.log(txt02);
24
25     const txt03 = await readFiles('./data/03.txt');
26     console.log(txt03);
27
28     const txt04 = await readFiles('./data/04.txt');
29     console.log(txt04);
30 }
31
32 fn();
33
34
35 /*
36     ; (async function () {
37         console.log(await readFiles('./data/01.txt'))
38         console.log(await readFiles('./data/02.txt'))
39         console.log(await readFiles('./data/03.txt'))
40         console.log(await readFiles('./data/04.txt'))
41     })()
42 */
43
44
45 console.log(100);
```

5. Promise 的实例方法

then方法, catch方法, finally方法

then 方法, 不仅可以处理成功, 也可以处理失败, 是俩参数。



JavaScript

复制代码

```
1 p1.then(data => {  
2     // 只会处理成功的回调。  
3 })
```

5.1 catch 方法

只能处理失败, 只要reject了, 必然会走到 catch 中

```
1  /*
2      a. 使用 catch 方法的前提, 不能在then方法中写第2个形参回调
3      b. catch 方法会捕获 reject 的调用
4      c. 如果只写了then的成功回调, 那么reject之后会报内部错误
5      d. 实例可以多次的单独调用then方法, 都可以获取值
6      e. 实例连续链式调用then方法, 后续的方法获取不到值, 需要return
7      f. 一般经典搭配就是 then 和 catch 一起写, 两个回调的方式很少
8  */
9  // 1. catch 方法的介绍
10 const p1 = new Promise((resolve, reject) => {
11     // resolve('ok');
12     reject('no');
13 })
14 // 1.1 使用方法
15 // p1
16 //     .then(data => {
17 //         console.log(data);
18 //     })
19 //     .catch(reason => {
20 //         console.log(reason, 111);
21 //     })
22
23 // 1.2 多次的调用then - 可以获取到值
24 // p1.then(data => {
25 //     console.log(data);
26 // });
27 // p1.then(data => {
28 //     console.log(data);
29 // });
30 // p1.then(data => {
31 //     console.log(data);
32 // });
33
34 // 后续的then方法需要前面的then方法返回内容
35 // p1
36 //     .then(data => {
37 //         console.log(data);
38 //         return 'chenwei'
39 //     })
40 //     .then(data => {
41 //         console.log(data);
42 //         return 10;
43 //     })
44 //     .then(data => {
45 //         console.log(data);
```



```

46 //    })
47 //
48 // 经典搭配
49 p1
50   .then(data => {
51     console.log(data);
52   })
53   .catch(err => {
54     console.log(err);
55   });
56
57 // - 只能处理失败
58 // p1.catch(err => {
59 //   console.log(err, 100);
60 // })

```

5.2 finally 方法

finally 就是最终的意思，也就是不管成功还是失败，都会执行一次

JavaScript | 复制代码

```

1 // 1. 实例方法 finally
2 const p1 = new Promise((resolve, reject) => {
3   // resolve('ok');
4   reject('no');
5 })
6 p1
7   .then(data => {
8     console.log(data);
9   })
10  .catch(err => {
11    console.log(err);
12  })
13  .finally(info => {
14    // 不要获取值，就是做一个终结或者提示
15    console.log(info, 123);
16  })

```

6. Promise 的静态方法

静态方法是挂载到 Promise 构造函数上的方法，实例不可以用。

6.1 Promise.resolve()

resolve 是之前的形参，但是这里是固定的方法：

```
JavaScript | 复制代码

1 // 1. Promise.resolve() 方法
2 /*
3     a. 默认丢一些非Promise的东西，快速创建一个成功的实例
4     b. 好处就是可以使用 then 方法，在某个函数的调用结束之后获取值
5     c. 也可以传入 promise 实例对象，这样的话就是返回原结果
6 */
7
8 // 1.1 直接调用 - 为了快速创建一个成功的promise实例
9 // const result = Promise.resolve(100);
10
11 // 1.2 传入一个对象，含有then方法
12 // const result = Promise.resolve({
13 //     then(resolve, reject) {
14 //         // resolve('ok');
15 //         reject('报错了吧');
16 //     }
17 // })
18
19 // 1.3 传入一个promise实例
20 const result = Promise.resolve(new Promise((resolve, reject) => {
21     // resolve('很好, hello !')
22     reject('很不好, error')
23 })))
24
25 // console.log(result);
26 result.then(data => {
27     console.log(data);
28 }).catch(err => {
29     console.log(err);
30 })
```

6.2 Promise.reject()

```
1 // 1. Promise.reject() 方法
2 /*
3     a. 默认丢一些非Promise的东西，快速创建一个失败的实例
4     b. 好处就是可以使用 catch 方法，在某个函数的调用结束之后获取报错信息
5     c. 也可以传入 promise 实例对象，这样的话就是返回原结果
6 */
7
8 // 1.1 直接调用 - 为了快速创建一个失败的promise实例
9 // const result = Promise.reject('报错的信息');
10
11 // 1.2 传入一个报错对象
12 // const result = Promise.reject(new SyntaxError('语法错误'));
13
14 // 1.3 传入一个promise实例
15 const result = Promise.resolve(new Promise((resolve, reject) => {
16     // resolve('很好, hello !')
17     reject('很不好, error')
18 })))
19
20 console.log(result);
21 result.catch(err => {
22     console.log(err);
23     // console.log(err.message);
24 })
25 result.then(() => { }, err => {
26     // console.log(err.message, 111);
27     console.log(err);
28 })
```

6.3 Promise.all()

```
1 // 1. Promise.all 方法
2 // all: 所有的, 这里的任务肯定不止一种;
3 /*
4     a. Promise.all 方法可以处理多个异步, 要求传入数组
5     b. 如果所有的异步都成功, 那么同步的返回成功之后的数据, 是数组
6     c. 如果有一个失败, 则直接返回第一个失败的结果, 其他的没有
7 */
8 const p1 = new Promise((resolve, reject) => {
9     setTimeout(() => {
10         resolve('1, [1, 2, 3, 4]')
11     }, 3000)
12 })
13 const p2 = Promise.resolve('2, success to response data.')
14 const p3 = new Promise((resolve, reject) => {
15     setTimeout(() => {
16         resolve('3, 这是最后的结果.')
17     }, 1500)
18 })
19
20 // 1.1 使用 Promise.all() 方法
21 // 参数, 是数组的形式, 用于存放多个任务
22 const result = Promise.all([p2, p3, p1]);
23 result
24     .then(values => {
25         console.log(values);
26     })
27     .catch(err => {
28         console.log(err);
29     })
```

6.4 Promise.race()

race: 赛跑的意思, 跑的最快的就是第一名。

```
1 // 1. Promise.race() 方法
2 // race 就是赛跑的意思, 谁先谁返回
3 /*
4     a. Promise.race 方法需要传入一个数组, 执行多个实例
5     b. 如果Promise内部没有异步任务, 则优先返回前面最早成功的那个
6     c. 如果有错误, 直接返回错误, 不会往后继续找
7     d. 如果有定时器, 或者耗时, 则返回最先的resolve或者reject
8 */
9
10 const p1 = new Promise((resolve, reject) => {
11     setTimeout(() => {
12         resolve('1, [1, 2, 3, 4]');
13     }, 0)
14 })
15
16 // const pp = Promise.reject(new Error('错误了'));
17 const pp = Promise.resolve('成功了! ');
18 const p2 = Promise.reject('2, success to response data.')
19 const p3 = new Promise((resolve, reject) => {
20     setTimeout(() => {
21         resolve('3, 这是最后的结果.')
22     }, 1500)
23 })
24
25 // 1.1 调用 Promise.race() 方法
26 const result = Promise.race([p1, pp, p2, p3]);
27 result
28     .then(data => {
29         console.log(data);
30     })
31     .catch(err => {
32         console.log(err);
33     })
```

7. 注意事项

7.1 改变状态的3种行为

```
1  /*
2      a. new Promise的 excutor 回调函数是同步的
3      b. 如果内部没有被改变状态，默认是 pending
4  */
5  // 改变状态的3种行为
6  const p1 = new Promise((resolve, reject) => {
7      // 先打印123，再输出状态，证明 excutor 回调函数是同步的
8      console.log(123);
9      // resolve('ok');
10     // reject('no')
11     // throw '报错! ';
12     // 如果是new出来的，需要打印的时候输出 err.message
13     throw new RangeError('报错! ');
14 })
15 console.log(p1);
16 p1.catch(err => {
17     console.log(err.message);
18 })
```

7.2 promise 的多个回调

```
1  /*
2      a. new Promise 的 executor 是同步, then 是异步
3      b. then 方法调用后会返回一个 pending 状态的 Promise 实例
4      c. 不管上一个then方法成功还是失败, 都会把返回的实例修改为 fulfilled
5      d. 也就是, then 可以连续的链式调用, 但是 error 回调不会
6      e. 如果第一个then没有error回调, 会执行后面的error回调
7  */
8  const p1 = new Promise((resolve, reject) => {
9      console.log(1);
10     reject('error');
11 })
12 // p1
13 //     .then(
14 //         data => console.log(data, '1'),
15 //         reason => console.log(reason, 'a')
16 //     )
17 //     .then(
18 //         data => console.log(data, '2'),
19 //         reason => console.log(reason, 'b')
20 //     )
21 //     .then(
22 //         data => console.log(data, '3'),
23 //         reason => console.log(reason, 'c')
24 //     )
25 const result = p1.then(
26     data => console.log(data, '1'),
27     reason => console.log(reason, 'a')
28 )
29 // 得到一个pending状态
30 // 在执行了之后, 默认会转 fulfilled 状态
31 // 打印的时候看到的是初始值, 打开三角之后是另外的值 (浏览器的机制)
32 console.log(result);
33 // const obj = {
34 //     uname: '张三'
35 // }
36 // setTimeout(() => {
37 //     obj.uname = '李四'
38 // }, 1000)
39 // console.log(obj);
```

7.3 状态和回调的先后使用

```
1 // 1. 状态和回调的先后顺序
2 // pending fulfilled rejected => resolve reject throw
3 // then 方法里面的回调函数
4 // 1.1 先回调的
5 const p1 = new Promise((resolve, reject) => {
6     // console.log(1);
7     resolve('ok');
8 })
9 setTimeout(() => {
10     p1.then(data => {
11         console.log(2, data);
12     })
13 }, 3000)
14 // console.log(3);
15 // 1.2 后改状态
16 new Promise((resolve, reject) => {
17     setTimeout(() => {
18         resolve(100)
19     }, 2000)
20 })
21 .then(data => {
22     console.log(data);
23 })
```

7.4 then 方法的返回状态


```
1 // 1. then方法的返回值
2 /*
3     a. 如果 then 方法没有指定返回值, 默认返回一个 promise
4         最终结果为 fulfilled, result 为undefined
5     b. 如果 then 方法指定的返回值是非 Promise 对象
6         还是会返回一个 promise 实例, 状态为 fulfilled ,
7     c. 如果 then 方法指定的返回值是 throw 抛出的
8         返回promise实例, 状态为 rejected, result 就是这
9
10 */
11 */
12 const p1 = new Promise((resolve, reject) => {
13     resolve('ok')
14 })
15 const result = p1.then(data => {
16     // console.log(data);
17     // return true;
18     // throw new Error('报错! ')
19     // return Promise.resolve('ok');
20     // return Promise.reject('error');
21     return new Promise((resolve, reject) => {
22         // resolve('ok');
23         // reject('no')
24         throw 'hahaha';
25     })
26 })
27 console.log(result);
```

7.5 异常穿透和断链

```
1 new Promise((resolve, reject) => {
2   // reject('error 错误! ')
3   resolve('成功')
4 })
5 .then(data => {
6   console.log(data);
7 }, err => Promise.reject(err))
8 .then(data => {
9   console.log(data); // undefined
10  // 抛出错误
11  throw 'error 抛出! ';
12 })
13 .then(data => {
14   console.log(data);
15 })
16 .then(data => {
17   console.log(data);
18 })
19 .catch(err => {
20   console.log(err);
21 })
```

```
1 new Promise((resolve, reject) => {
2     // reject('error 错误! ')
3     resolve('处理事情')
4 })
5 .then(data => {
6     console.log(data, 1);
7 })
8 .then(data => {
9     console.log(data, 2);
10    // 假设这一次任务成功了，如何保障后续的代码不再调用了
11    // 唯一的解决方案 - 返回了一个 pending，没有任何的状态改变
12    return new Promise(() => { })
13 })
14 .then(data => {
15     console.log(data, 3);
16 })
17 .then(data => {
18     console.log(data);
19 })
20 .catch(err => {
21     console.log(err);
22 })
```

8. 手工封装一个 Promise (熟悉)

意义：并不是要把代码写的多高级，而是同过手动封装的过程，了解为什么Promise会产生那么多奇怪的情景。

8.1 基本封装

```

1
2 // 1. 创建构造函数
3 function Promise(executor) {
4     // 5. 声明状态和返回的值
5     this.PromiseState = 'pending';
6     this.PromiseResult = undefined;
7
8     // 4. 准备对应的函数体
9     const success = value => {
10         // 6.1 标记状态为 fulfilled 再修改值
11         this.PromiseState = 'fulfilled';
12         this.PromiseResult = value;
13     }
14     const error = err => {
15         // 6.2 标记状态为 rejected 再修改值
16         this.PromiseState = 'rejected';
17         this.PromiseResult = err;
18     }
19
20     // 3. 调用回调函数 executor
21     executor(success, error);
22 }
23
24 // 2. 原型上挂载方法
25 Promise.prototype.then = function (onFulfilled, onRejected) {
26     // 获取状态和结果
27
28 }
29

```

8.2 处理状态

调用了3次then方法，如果有异步任务，则会创建3个pending的状态实例 ✓

3个异步任务，到时间了就会改变状态，三个任务就分别改变的是3个状态 ✓

每个状态改变，肯定使用了 resolve 或者 reject，再不然就是 throw ✓

resolve 会执行 success，reject 或者 throw 执行 error，总共就3次 ✓

解决办法：

每一次创建pending的时候，都指定一个成功的回调ok，和失败的回调err

这一组成功和失败的回调，放到构造函数的数组中，push的方式

那么，3个异步任务，每个任务都会有 成功 和 失败 的回调，总共6个。

[{ ok: fn, err: fn }, { ok: fn, err: fn }, { ok: fn, err: fn }] √

最终异步结束会改变状态，也就是会执行 resolve 或者 reject

resolve 调用了，构造函数中的 success 就会被执行

reject 调用了，构造函数中的 error 就会被执行 √

调用遍历的 ok或者err方法，就是在调用then中的成功或者失败的回调

```
1
2 // 1. 创建构造函数
3 function Promise(executor) {
4     // 8.1 定义一个数组，专门用于存放异步的函数
5     this.callbacks = [];
6
7     // 4. 准备对应的函数体
8     const success = value => {
9         // 8.2 处理队列的函数
10        if (this.callbacks.length > 0) {
11            this.callbacks.forEach(cb => {
12                // 在队列中找成功的执行
13                cb.ok(value)
14            })
15        }
16    }
17    const error = err => {
18        // 8.3 处理队列的函数
19        if (this.callbacks.length > 0) {
20            this.callbacks.forEach(cb => {
21                // 在队列中找成功的执行
22                cb.err(err)
23            })
24        }
25    }
26 }
27
28 // 2. 原型上挂载方法
29 Promise.prototype.then = function (onFulfilled, onRejected) {
30     // 8. 如果是异步的，那么状态肯定是 pending
31     if (this.PromiseState === 'pending') {
32         this.callbacks.push({
33             ok(value) {
34                 onFulfilled(value);
35             },
36             err(err) {
37                 onRejected(err);
38             }
39         })
40     }
41 }
42
```

8.3 不可逆的操作



JavaScript

复制代码

```
1 // 都是要加在 success 和 error 里面的
2 if (this.PromiseState !== 'pending') return
```

8.4 封装then方法的回调

- 1, then方法有返回值了, 是promise实例对象
- 2, then方法的返回值promise实例对象, 会根据回调函数的执行改变状态
- 3, then方法的返回值promise实例对象, 会根据return的情况, 实时改变内容
- 4, 当缺省回调函数, 或者报错的情况, 也能及时捕获到

```

1 // 2. 设置实例方法挂载到原型上
2 Promise.prototype.then = function (onFulfilled, onRejected) {
3     // 返回一个 pending 状态的 Promise 实例
4     return new Promise((resolve, reject) => {
5         // 先执行的then方法, 还未处理状态, 所以状态默认是pending
6         if (this.promiseState === 'pending') {
7             // 把这边的两个回调, 放入到实例的数组中
8             this.callbacks.push({
9                 ok(val) {
10                     try {
11                         const res = onFulfilled(val);
12                         if (res instanceof Promise) {
13                             res.then(data => resolve(data), err => reject(
14                                 err))
15                             } else {
16                                 resolve(res)
17                             }
18                         } catch (e) {
19                             reject(e.message)
20                         }
21                     },
22                     ng(err) {
23                         try {
24                             const res = onRejected(err);
25                             if (res instanceof Promise) {
26                                 res.then(data => resolve(data), err => reject(
27                                     err))
28                             } else {
29                                 resolve(res)
30                             }
31                         } catch (e) {
32                             reject(e.message)
33                         }
34                     }
35                 })
36             }
37         // then 方法的回调1
38         if (this.promiseState === 'fulfilled') {
39             // 只有执行了resolve才会修改状态和携带数据
40             // res 处理成功回调之后返回的Promise实例
41             // then 方法, 回调函数1: 成功的, 回调函数2: 失败的
42             // res.then(data => resolve之后获取data, err => reject之后获取报
43             错)
44             try {

```



```

43         const res = onFulfilled(this.promiseResult);
44         // 判断返回值是否为 Promise 的实例
45         if (res instanceof Promise) {
46             res.then(data => resolve(data), err => reject(err))
47         } else {
48             resolve(res);
49         }
50     } catch (e) {
51         reject(e.message)
52     }
53
54 }
55
56 // then 方法的回调2
57 if (this.promiseState === 'rejected') {
58     try {
59         const res = onRejected(this.promiseResult);
60         if (res instanceof Promise) {
61             res.then(data => resolve(data), err => reject(err))
62         } else {
63             resolve(res);
64         }
65     } catch (e) {
66         reject(e.message);
67     }
68 }
69 })
70
71 }

```

html 里面的调用情况

```
1 // 1. 调用自定义的Promise
2 const p1 = new Promise((resolve, reject) => {
3   setTimeout(() => {
4     resolve(100);
5     // reject('err')
6     // 公司里面会经常做一些封装
7     // try {
8     //   throw Error('异步内部的错误! ')
9     // } catch (err) {
10    //   reject(err.message);
11    // }
12  }, 1000)
13  // resolve(100);
14  // reject('error123');
15  // throw Error('报错了! ')
16 })
17 // 2. 打印成功或者失败之后的值
18 const result = p1.then(data => {
19   console.log(data);
20   // return 100;
21   // return new Promise((resolve, reject) => {
22   //   // resolve('ok')
23   //   reject('error,haha...')
24   // });
25   throw new Error('err 成功之后手动的错误')
26 }, err => {
27   console.log(err, '参数2的回调');
28   // return new Promise((resolve, reject) => {
29   //   reject('错误处理的错误回调')
30   // });
31   throw new Error('err 手动的错误')
32 })
33 // 函数没有返回值默认是undefined
34 // 但是内置的Promise的then方法默认返回一个 pending 状态的实例
35 console.log(result);
```

8.5 优化封装then 方法的回调

```
1 // 8.5 封装一个处理回调的函数
2 const handler = type => {
3   try {
4     const res = type(this.promiseResult);
5     if (res instanceof Promise) {
6       res.then(data => resolve(data), err => reject(err))
7     } else {
8       resolve(res)
9     }
10  } catch (e) {
11    reject(e.message)
12  }
13 }
14
15 // 使用
16 handler(onFulfilled);
17 handler(onRejected);
```

8.6 处理then方法的执行为异步的操作

```
1 // 修改了四个部分的代码
2 // 异步then方法结束之后，循环队列数组中的回调
3 setTimeout(() => {
4     this.callbacks.forEach(cb => {
5         cb.ok(value);
6     })
7 })
8
9 // 异步then方法结束之后，循环队列数组中的回调
10 setTimeout(() => {
11     this.callbacks.forEach(cb => {
12         cb.ng(reason);
13     })
14 })
15
16 // then 方法的回调1
17 if (this.promiseState === 'fulfilled') {
18     setTimeout(() => {
19         handler(onFulfilled);
20     })
21 }
22 // then 方法的回调2
23 if (this.promiseState === 'rejected') {
24     setTimeout(() => {
25         handler(onRejected);
26     }, 0)
27 }
```

8.7 添加实例的 catch 方法

catch 方法，

- 1, 当多个then方法中，其中有一个 reject了，那么直接跳到 catch中执行
- 2, 正常搭配 then 使用，一个处理成功，一个处理失败
- 3, 先调用 catch 是没有问题的

```
1 // 在then方法中回调
2 return new Promise((resolve, reject) => {
3     // console.log(onFulfilled, onRejected, 1234);
4
5     // 调用undefined参数报错的问题
6     if (typeof onFulfilled !== 'function') {
7         onFulfilled = v => v;
8     }
9     // 有报错
10    if (typeof onRejected !== 'function') {
11        onRejected = r => { throw r };
12    }
13 })
14
15
16 Promise.prototype.catch = function (onRejected) {
17     // then 方法本身就会调用 new Promise()
18     return this.then(undefined, onRejected);
19 }
```

8.8 静态方法 resolve 和 reject

```
1 // 3. 添加静态方法
2 Promise.resolve = function (value) {
3     return new Promise((resolve, reject) => {
4         if (value instanceof Promise) {
5             // value 就是传入的 new Promise 的实例
6             value.then(resolve, reject)
7         } else {
8             resolve(value);
9         }
10    })
11 }
12
13 Promise.reject = function (reason) {
14     return new Promise((resolve, reject) => {
15         reject(reason);
16    })
17 }
```

8.9 静态方法 all

JavaScript | 复制代码

```
1  // 最后的功能
2  Promise.all = function (pArr) {
3      // 任务队列的数组
4      const task = [];
5      count = 0;
6
7      // 返回一个实例
8      return new Promise((resolve, reject) => {
9          // 1. 数组遍历
10         pArr.forEach((item, index) => {
11             // 2. 取值
12             item.then(
13                 v => {
14                     // 3. 按顺序提供内容
15                     task[index] = v;
16                     count++;
17                     // 4. 既然能走到成功的回调，证明异步也结束了
18                     if (count === pArr.length) {
19                         resolve(task);
20                     }
21                 },
22                 r => reject(r)
23             )
24         })
25     })
26 }
```

8.10 静态方法 race

```
1 // race 方法
2 Promise.race = function (pArr) {
3   return new Promise((resolve, reject) => {
4     // 任务执行
5     pArr.forEach(item => {
6       item.then(v => resolve(v), r => reject(r))
7     })
8   })
9 }
```