Promise的介绍和使用

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

```
// Array 的静态方法
Array.isArray(arr);    // 判断参数是否为数组

// Array 的实例方法
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

JavaScript | 复制代码

```
// 1. 常见的报错
// Uncaught Error:
// throw  代表抛出的意思
// 报错之后，后续的代码不执行

// 1.1 顶级的报错构造函数
// throw new Error('报错了! ');
// Uncaught  报错了?
// throw '报错了? ';

// 1.1 引用错误
// Uncaught ReferenceError
// console.log(num);
// const num = 10;

// 1.2 类型错误
// Uncaught TypeError
// num = 100;

// 1.3 范围错误
// Uncaught RangeError
// new Array(-1);

// 1.4 语法错误
// Uncaught SyntaxError
// var 3x = 100;

// throw new TypeError('类型错误');
// throw new SyntaxError('hello, your code is no ok!');
// console.log(100);
```

2.1 try...catch 解决语法报错阻塞

JavaScript | 复制代码

```javascript
// 1. 使用 try...catch 解决报错的问题
// 核心： 解决报错之后的阻塞
const num = 10;
// 传统写法会阻塞代码的执行
// num = 20;
// 1.1 使用 try...catch 的语法
try {
    // 要执行的语句
    num = 20;
} catch (error) {
    // 捕获了错误之后执行的语句
    // 自动接收形参，是捕获到的错误类型和错误内容!
    console.log(error);
}
console.log('最后要执行的代码! ', num);


// 2. 一些失误的封装
function getData() {
    const num = Math.floor(Math.random() * 30 + 3);
    if (num % 2 === 0) {
        console.log('是偶数，可以做处理。');
    } else {
        throw new RangeError('不能给奇数啊兄嘚!')
        // throw 'hahaha';
    }
}

// 2.1 try...catch 捕获错误
try {
    getData();
} catch (err) {
    // 这些错误提示，可以使用一些弹窗组件完成
    console.log(err.message); // 错误提示的文本
    console.log(err.stack);
    console.log(err.name);
}
console.log('哈哈哈，今天天气真不错。');
```

2.2 异步回调和同步回调

```JavaScript
// 回调函数分为两种
const arr = [1, 2, 3, 4];
arr.forEach(item => {
    console.log(item);
})
console.log('ok');

// 2. 异步回调
// setTimeout(() => {
//     console.log(1);
// }, 0)
// console.log(2);

// 目前来说，异步任务，就3种
// 事件，定时器，xhr请求
// 服务器端: fs模块中的api...  连接mongoDB数据库

// 事件队列的事儿
// 异步任务的回调会进入事件队列  Event Queue
// 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。

JavaScript 复制代码

```
// 1. 创建 Promise 对象
// 有且仅有一个参数，为回调函数
const a1 = new Promise(function () { })

// 得到一个 pending 的状态
// pending: 等待，期待发生的;
console.log(a1);
```

3.1 正确的使用

JavaScript 复制代码

```
// 1.1 正确的使用 Promise 的姿势
const a1 = new Promise(function (resolve, reject) {
    // resolve  reject  属于 Promise 回调的形参
    // 这俩形参，都是方法，直接加小括号 () 调用
    // resolve 代表 成功，可以执行的意思
    // reject  代表 失败，拒绝考虑，不录用的意思

    // 执行了这个函数之后，状态会改变为 fulfilled =>  应验了
    // resolve();

    // 执行了这个函数之后，状态会改变为 rejected  =>  失败了
    // reject();
})
// console.log(a1);

// 状态1:  期待的状态: pending
// 状态2:  应验的状态: fulfilled
// 状态3:  失败的状态: rejected

// 状态1可操作性，状态2和状态3不可逆

// 只会有2种结果
// 要么就是 resolve   要么就是 reject
```

3.2 Promise 的封装

JavaScript 复制代码

```javascript
// 1. 封装的业务
function getData() {
    // 1.1 基于Promise的容器
    // const a1 = new Promise((resolve, reject) => {
    return new Promise((resolve, reject) => {
        const num = Math.floor(Math.random() * 30 + 3);
        if (num % 2 === 0) {
            // console.log('是偶数，可以做处理。');
            // 1.2 成功
            resolve('是偶数，可以做处理。');
        } else {
            // throw new RangeError('不能给奇数啊兄嗝!')
            // 1.3 失败
            reject('不能给奇数啊兄嗝!');
        }
    })
    // 1.3 返回 promise 的结果
    // return a1;
}


// 2. 其实，promise就是把之前应该写在回调函数里面的业务
// 可以放出来写了，还可以分不同的状态来写
// console.log(getData());
const p1 = getData();

// 3. p1 是实例对象，实例对象有俩方法
// then 处理成功   catch 捕获失败
p1.then(function (data) {
    console.log('成功了! ---', data)
})
p1.catch(function (err) {
    console.log('失败了，报错了----', err);
})

// promise 代码内部报错不会阻塞执行
console.log(1100);
```

3.3 then 方法的两个形参

JavaScript | 复制代码

```javascript
// 1. Promise 实例对象回调的详解
function fn() {
    // 1.1 返回 Promise 的实例
    return new Promise((resolve, reject) => {
        // 1.2 resolve 代表成功的回调，reject 代表失败的回调
        // resolve();
        // resolve(100);
        // reject();
        // reject('网络错误！');
        reject(new Error('网络不稳定！'))
    })
}
// 2. 返回的 Promise 实例叫做 p1
const p1 = fn();
// 2.1 使用 p1 的实例方法
// 参数1：成功的回调函数体
// 参数2：失败的回调函数体
p1.then(
    // resolve 执行这个，会传参
    data => {
        console.log('成功了，非常好。=> ', data);
    },
    // reject  执行这个，会传入错误信息
    reason => {
        console.log('sorry，失败了。=> ', reason.message);
    }
)
```

JavaScript | 复制代码

```javascript
// 1.1 封装一个业务
function random() {
    // 1.2 返回一个 promise 的实例
    return new Promise((resolve, reject) => {
        // 1.3 异步的耗时的任务
        setTimeout(function () {
            const num = Math.floor(Math.random() * 30 + 3);
            // 三目运算符
            num % 2 ? reject(new TypeError('err odd')) : resolve('ok even');
        }, 1500)
    })
}
// 2. 获得实例对象
const p1 = random();
// 2.1 实例的回调，是在内部结果产生之后才会执行
p1.then(
    data => {
        console.log(data);
    },
    reason => {
        console.log(reason.message);
    }
)

// 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

JavaScript | 复制代码

```javascript
// 1. 要求使用 Promise 封装，使得代码完整
let num = 10;
// 1.1 定义函数
function fn() {
    // 1.2 返回 Promise 的实例
    return new Promise((resolve, reject) => {
        // 1.3 开始异步任务
        setTimeout(function () {
            num = '10086';
            // 1.4 指定状态
            resolve(num);
        }, 1500);
    })
}
// 就是需要在定时器的外面打印 num
// 链式调用
fn()
    .then(data => {
        console.log(data);
    })
```

3.5 封装请求

JavaScript | 复制代码

```javascript
// 1.1 封装函数
function getData(url) {
    // 1.2 返回一个 promise 的实例对象
    return new Promise((resolve, reject) => {
        $.ajax({
            url,
            success(res) {
                // 成功之后，把数据传入进来
                res.code === 200 ? resolve(res) : reject('请求失败');
            },
            error(err) {
                // 错误处理在这边
                reject(err);
            }
        })
    })
}
// 2. 创建实例
getData('http://127.0.0.1:3000/demo')
    .then(
        data => {
            console.log(1111, data);
        },
        reason => {
            console.log(2222, reason);
        }
    )
```

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。

JavaScript | 复制代码

```javascript
// 1. 引入模块
const fs = require('fs');

// 2. 读取文件

// 2.1 同步代码不会影响顺序问题
// console.log(fs.readFileSync('./data/01.txt', 'utf8'));
// console.log(fs.readFileSync('./data/02.txt', 'utf8'));
// console.log(fs.readFileSync('./data/03.txt', 'utf8'));
// console.log(fs.readFileSync('./data/04.txt', 'utf8'));

// 2.2 异步读取的任务
// 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
fs.readFile('./data/01.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
    // 读取01
    fs.readFile('./data/02.txt', 'utf8', (err, data) => {
        if (err) throw err;
        console.log(data);
        // 读取02
        fs.readFile('./data/03.txt', 'utf8', (err, data) => {
            if (err) throw err;
            console.log(data);
            // 读取03
            fs.readFile('./data/04.txt', 'utf8', (err, data) => {
                if (err) throw err;
                console.log(data);
                // 读取04
            })
        })
    })
})

// 不会阻塞同步的代码
console.log(100);
```

4.1 优化代码

JavaScript 复制代码

```javascript
// 1. 引入模块
const fs = require('fs');

// 2. 使用 Promise 来解决问题
function readFiles(path) {
    return new Promise((resolve, reject) => {
        // 2.1 执行异步任务
        fs.readFile(path, 'utf8', (err, data) => {
            if (err) throw err;
            // 成功
            resolve(data);
        })
    })
}

readFiles('./data/01.txt')
    .then(data => {
        console.log(data);
        // 读取了之后，再来读取第二个文件
        return readFiles('./data/02.txt')
    })
    .then(data => {
        console.log(data);
        // 读取了之后，再来读取第三个文件
        return readFiles('./data/03.txt')
    })
    .then(data => {
        console.log(data);
        // 读取了之后，再来读取四个文件
        return readFiles('./data/04.txt')
    })
    .then(data => {
        console.log(data);
    })


console.log(100);
```

4.2 最终的优化1

提前使用到了新语法 async / await

JavaScript 复制代码

```javascript
// 1. 引入模块
const fs = require('fs');

// 2. 使用 Promise 来解决问题
function readFiles(path) {
    return new Promise((resolve, reject) => {
        // 2.1 执行异步任务
        fs.readFile(path, 'utf8', (err, data) => {
            if (err) throw err;
            // 成功
            resolve(data);
        })
    })
}

// 3. ES8 中更新了一个语法糖  async / await
// 专门用于管理基于 Promise 封装的异步
async function fn() {
    const txt01 = await readFiles('./data/01.txt');
    console.log(txt01);

    const txt02 = await readFiles('./data/02.txt');
    console.log(txt02);

    const txt03 = await readFiles('./data/03.txt');
    console.log(txt03);

    const txt04 = await readFiles('./data/04.txt');
    console.log(txt04);
}

fn();


/*
    ; (async function () {
        console.log(await readFiles('./data/01.txt'))
        console.log(await readFiles('./data/02.txt'))
        console.log(await readFiles('./data/03.txt'))
        console.log(await readFiles('./data/04.txt'))
    })()
*/


console.log(100);
```

```
46
```

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

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

5.1 catch 方法

只能处理失败，只要reject了，必然会走到 catch 中

JavaScript | 复制代码

```javascript
/*
    a. 使用 catch 方法的前提，不能在then方法中写第2个形参回调
    b. catch 方法会捕获 reject 的调用
    c. 如果只写了then的成功回调，那么reject之后会报内部错误
    d. 实例可以多次的单独调用then方法，都可以获取值
    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
    f. 一般经典搭配就是 then 和 catch 一起写，两个回调的方式很少
*/
// 1. catch 方法的介绍
const p1 = new Promise((resolve, reject) => {
    // resolve('ok');
    reject('no');
})
// 1.1 使用方法
// p1
//     .then(data => {
//         console.log(data);
//     })
//     .catch(reason => {
//         console.log(reason, 111);
//     })

// 1.2 多次的调用then - 可以获取到值
// p1.then(data => {
//     console.log(data);
// });
// p1.then(data => {
//     console.log(data);
// });
// p1.then(data => {
//     console.log(data);
// });

// 后续的then方法需要前面的then方法返回内容
// p1
//     .then(data => {
//         console.log(data);
//         return 'chenwei'
//     })
//     .then(data => {
//         console.log(data);
//         return 10;
//     })
//     .then(data => {
//         console.log(data);
```

```
//     })

// 经典搭配
p1
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    });

// - 只能处理失败
// p1.catch(err => {
//     console.log(err, 100);
// })
```

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. 实例方法 finally
const p1 = new Promise((resolve, reject) => {
    // resolve('ok');
    reject('no');
})
p1
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    })
    .finally(info => {
        // 不要获取值，就是做一个终结或者提示
        console.log(info, 123);
    })
```

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. Promise.resolve() 方法
/*
    a. 默认丢一些非Promise的东西，快速创建一个成功的实例
    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
    c. 也可以传入 promise 实例对象，这样的话就是返回原结果
*/

// 1.1 直接调用 - 为了快速创建一个成功的promise实例
// const result = Promise.resolve(100);

// 1.2 传入一个对象，含有then方法
// const result = Promise.resolve({
//     then(resolve, reject) {
//         // resolve('ok');
//         reject('报错了吧');
//     }
// })

// 1.3 传入一个promise实例
const result = Promise.resolve(new Promise((resolve, reject) => {
    // resolve('很好，hello !')
    reject('很不好，error')
}))

// console.log(result);
result.then(data => {
    console.log(data);
}).catch(err => {
    console.log(err);
})
```

6.2 Promise.reject()

JavaScript | 复制代码

```javascript
// 1. Promise.rejct() 方法
/*
    a. 默认丢一些非Promise的东西，快速创建一个失败的实例
    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
    c. 也可以传入 promise 实例对象，这样的话就是返回原结果
*/

// 1.1 直接调用 - 为了快速创建一个失败的的promise实例
// const result = Promise.reject('报错的信息');

// 1.2 传入一个报错对象
// const result = Promise.reject(new SyntaxError('语法错误'));

// 1.3 传入一个promise实例
const result = Promise.resolve(new Promise((resolve, reject) => {
    // resolve('很好，hello !')
    reject('很不好，error')
}))

console.log(result);
result.catch(err => {
    console.log(err);
    // console.log(err.message);
})
result.then(() => { }, err => {
    // console.log(err.message, 111);
    console.log(err);
})
```

6.3 Promise.all()

JavaScript | 复制代码

```javascript
// 1. Promise.all 方法
// all: 所有的，这里的任务肯定不止一种;
/*
    a. Promise.all 方法可以处理多个异步，要求传入数组
    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
    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
*/
const p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('1, [1, 2, 3, 4]')
    }, 3000)
})
const p2 = Promise.resolve('2, success to response data.')
const p3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('3, 这是最后的结果.')
    }, 1500)
})

// 1.1 使用 Promise.all() 方法
// 参数，是数组的形式，用于存放多个任务
const result = Promise.all([p2, p3, p1]);
result
    .then(values => {
        console.log(values);
    })
    .catch(err => {
        console.log(err);
    })
```

6.4 Promise.race()

race: 赛跑的意思，跑的最快的就是第一名。

JavaScript | 复制代码

```javascript
// 1. Promise.race()   方法
// race 就是赛跑的意思，谁先谁返回
/*
    a. Promise.race 方法需要传入一个数组，执行多个实例
    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
    c. 如果有错误，直接返回错误，不会往后继续找
    d. 如果有定时器，或者耗时，则返回最先的resolve或者reject
*/

const p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('1, [1, 2, 3, 4]');
    }, 0)
})

// const pp = Promise.reject(new Error('错误了'));
const pp = Promise.resolve('成功了！');
const p2 = Promise.reject('2, success to response data.')
const p3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('3, 这是最后的结果.')
    }, 1500)
})

// 1.1 调用 Promise.race() 方法
const result = Promise.race([p1, pp, p2, p3]);
result
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    })
```

7. 注意事项

7.1 改变状态的3种行为

JavaScript | 复制代码

```javascript
/*
    a. new Promise的 excutor 回调函数是同步的
    b. 如果内部没有被改变状态，默认是 pending
*/
// 改变状态的3种行为
const p1 = new Promise((resolve, reject) => {
    // 先打印123，再输出状态，证明 executor 回调函数是同步的
    console.log(123);
    // resolve('ok');
    // reject('no')
    // throw '报错！';
    // 如果是new出来的，需要打印的时候输出 err.message
    throw new RangeError('报错！');
})
console.log(p1);
p1.catch(err => {
    console.log(err.message);
})
```

7.2 promise 的多个回调

JavaScript | 复制代码

```javascript
/*
    a. new Promise 的 executor 是同步，then 是异步
    b. then 方法调用后会返回一个 pending 状态的 Promise 实例
    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
    d. 也就是，then 可以连续的链式调用，但是 error 回调不会
    e. 如果第一个then没有error回调，会执行后面的error回调
*/
const p1 = new Promise((resolve, reject) => {
    console.log(1);
    reject('error');
})
// p1
//     .then(
//         data => console.log(data, '1'),
//         reason => console.log(reason, 'a')
//     )
//     .then(
//         data => console.log(data, '2'),
//         reason => console.log(reason, 'b')
//     )
//     .then(
//         data => console.log(data, '3'),
//         reason => console.log(reason, 'c')
//     )
const result = p1.then(
    data => console.log(data, '1'),
    reason => console.log(reason, 'a')
)
// 得到一个pending状态
// 在执行了之后，默认会转 fulfilled 状态
// 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）
console.log(result);
// const obj = {
//     uname: '张三'
// }
// setTimeout(() => {
//     obj.uname = '李四'
// }, 1000)
// console.log(obj);
```

7.3 状态和回调的先后使用

JavaScript 复制代码

```
// 1. 状态和回调的先后顺序
//   pending   fufilled   rejected  =>    resolve   reject  throw
//   then 方法里面的回调函数
// 1.1 先回调的
const p1 = new Promise((resolve, reject) => {
    // console.log(1);
    resolve('ok');
})
setTimeout(() => {
    p1.then(data => {
        console.log(2, data);
    })
}, 3000)
// console.log(3);
// 1.2 后改状态
new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(100)
    }, 2000)
})
    .then(data => {
        console.log(data);
    })
```

7.4 then 方法的返回状态

JavaScript | 复制代码

```javascript
// 1. then方法的返回值
/*
    a. 如果 then 方法没有指定返回值，默认返回一个 promis
        最终结果为 fulfilled，result 为undefined
    b. 如果 then 方法指定的返回值是非 Promise 对象
        还是会返回一个 promise 实例，状态为 fulfilled ，
    c. 如果 then 方法指定的返回值是 thorw 抛出的
        返回promise实例，状态为 rejected，result 就是这


*/
const p1 = new Promise((resolve, reject) => {
    resolve('ok')
})
const result = p1.then(data => {
    // console.log(data);
    // return true;
    // throw new Error('报错！')
    // return Promise.resolve('ok');
    // return Promise.reject('error');
    return new Promise((resolve, reject) => {
        // resolve('ok');
        // reject('no')
        throw 'hahaha';
    })
})
console.log(result);
```

7.5 异常穿透和断链

JavaScript | 复制代码

```javascript
new Promise((resolve, reject) => {
    // reject('error 错误! ')
    resolve('成功')
})
    .then(data => {
        console.log(data);
    }, err => Promise.reject(err))
    .then(data => {
        console.log(data); // undefined
        // 抛出错误
        throw 'error 抛出! ';
    })
    .then(data => {
        console.log(data);
    })
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    })
```

JavaScript | 复制代码

```javascript
new Promise((resolve, reject) => {
    // reject('error 错误! ')
    resolve('处理事情')
})
    .then(data => {
        console.log(data, 1);
    })
    .then(data => {
        console.log(data, 2);
        // 假设这一次任务成功了, 如何保障后续的代码不再调用了
        // 唯一的解决方案 - 返回了一个 pending, 没有任何的状态改变
        return new Promise(() => { })
    })
    .then(data => {
        console.log(data, 3);
    })
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    })
```

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

JavaScript | 复制代码

```
// 1. 创建构造函数
function Promise(executor) {
    // 5. 声明状态和返回的值
    this.PromiseState = 'pending';
    this.PromiseResult = undefined;

    // 4. 准备对应的函数体
    const success = value => {
        // 6.1 标记状态为 fulfilled 再修改值
        this.PromiseState = 'fulfilled';
        this.PromiseResult = value;
    }
    const error = err => {
        // 6.2 标记状态为 rejected 再修改值
        this.PromiseState = 'rejected';
        this.PromiseResult = err;
    }

    // 3. 调用回调函数 executor
    executor(success, error);
}

// 2. 原型上挂载方法
Promise.prototype.then = function (onFulfilled, onRejected) {
    // 获取状态和结果

}

```

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

JavaScript | 复制代码

```javascript

// 1. 创建构造函数
function Promise(executor) {
    // 8.1 定义一个数组，专门用于存放异步的函数
    this.callbacks = [];

    // 4. 准备对应的函数体
    const success = value => {
        // 8.2 处理队列的函数
        if (this.callbacks.length > 0) {
            this.callbacks.forEach(cb => {
                // 在队列中找成功的执行
                cb.ok(value)
            })
        }
    }
    const error = err => {
        // 8.3 处理队列的函数
        if (this.callbacks.length > 0) {
            this.callbacks.forEach(cb => {
                // 在队列中找成功的执行
                cb.err(err)
            })
        }
    }
}

// 2. 原型上挂载方法
Promise.prototype.then = function (onFulfilled, onRejected) {
    // 8. 如果是异步的，那么状态肯定是 pending
    if (this.PromiseState === 'pending') {
        this.callbacks.push({
            ok(value) {
                onFulfilled(value);
            },
            err(err) {
                onRejected(err);
            }
        })
    }
}

```

8.3 不可逆的操作

JavaScript | 复制代码

```
// 都是要加在 success 和 error 里面的
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

JavaScript | 复制代码

```javascript
// 2. 设置实例方法挂载到原型上
Promise.prototype.then = function (onFulfilled, onRejected) {
    // 返回一个 pending 状态的 Promise 实例
    return new Promise((resolve, reject) => {
        // 先执行的then方法，还未处理状态，所以状态默认是pending
        if (this.promiseState === 'pending') {
            // 把这边的两个回调，放入到实例的数组中
            this.callbacks.push({
                ok(val) {
                    try {
                        const res = onFulfilled(val);
                        if (res instanceof Promise) {
                            res.then(data => resolve(data), err => reject(err))
                        } else {
                            resolve(res)
                        }
                    } catch (e) {
                        reject(e.message)
                    }
                },
                ng(err) {
                    try {
                        const res = onRejected(err);
                        if (res instanceof Promise) {
                            res.then(data => resolve(data), err => reject(err))
                        } else {
                            resolve(res)
                        }
                    } catch (e) {
                        reject(e.message)
                    }
                }
            })
        }

        // then 方法的回调1
        if (this.promiseState === 'fulfilled') {
            // 只有执行了resolve才会修改状态和携带数据
            // res 处理成功回调之后返回的Promise实例
            // then 方法，回调函数1： 成功的， 回调函数2：失败的
            // res.then(data => resolve之后获取data, err => reject之后获取报错)
            try {
```

```
                const res = onFulfilled(this.promiseResult);
                // 判断返回值是否为 Promise 的实例
                if (res instanceof Promise) {
                    res.then(data => resolve(data), err => reject(err))
                } else {
                    resolve(res);
                }
            } catch (e) {
                reject(e.message)
            }

        }

        // then 方法的回调2
        if (this.promiseState === 'rejected') {
            try {
                const res = onRejected(this.promiseResult);
                if (res instanceof Promise) {
                    res.then(data => resolve(data), err => reject(err))
                } else {
                    resolve(res);
                }
            } catch (e) {
                reject(e.message);
            }
        }
    })

}
```

> html 里面的调用情况

JavaScript | 复制代码

```javascript
// 1. 调用自定义的Promise
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(100);
    // reject('err')
    // 公司里面会经常做一些封装
    // try {
    //     throw Error('异步内部的错误! ')
    // } catch (err) {
    //     reject(err.message);
    // }
  }, 1000)
  // resolve(100);
  // reject('error123');
  // throw Error('报错了! ')
})
// 2. 打印成功或者失败之后的值
const result = p1.then(data => {
  console.log(data);
  // return 100;
  // return new Promise((resolve, reject) => {
  //     // resolve('ok')
  //     reject('error,haha...')
  // });
  throw new Error('err 成功之后手动的错误')
}, err => {
  console.log(err, '参数2的回调');
  // return new Promise((resolve, reject) => {
  //     reject('错误处理的错误回调')
  // })
  throw new Error('err 手动的错误')
})
// 函数没有返回值默认是undefined
// 但是内置的Promise的then方法默认返回一个 pending 状态的实例
console.log(result);
```

8.5 优化封装then 方法的回调

then 方法中 promise内部封装 JavaScript 复制代码

```
// 8.5 封装一个处理回调的函数
const handler = type => {
    try {
        const res = type(this.promiseResult);
        if (res instanceof Promise) {
            res.then(data => resolve(data), err => reject(err))
        } else {
            resolve(res)
        }
    } catch (e) {
        reject(e.message)
    }
}

// 使用
handler(onFulfilled);
handler(onRejected);
```

8.6 处理then方法的执行为异步的操作

JavaScript | 复制代码

```javascript
// 修改了四个部分的代码
// 异步then方法结束之后，循环队列数组中的回调
setTimeout(() => {
    this.callbacks.forEach(cb => {
        cb.ok(value);
    })
})

// 异步then方法结束之后，循环队列数组中的回调
setTimeout(() => {
    this.callbacks.forEach(cb => {
        cb.ng(reason);
    })
})

// then 方法的回调1
if (this.promiseState === 'fulfilled') {
  setTimeout(() => {
      handler(onFulfilled);
  })
}
// then 方法的回调2
if (this.promiseState === 'rejected') {
  setTimeout(() => {
      handler(onRejected);
  }, 0)
}
```

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

JavaScript 复制代码

```
// 在then方法中回调
return new Promise((resolve, reject) => {
    // console.log(onFulfilled, onRejected, 1234);

    // 调用undefniend参数报错的问题
    if (typeof onFulfilled !== 'function') {
        onFulfilled = v => v;
    }
    // 有报错
    if (typeof onRejected !== 'function') {
        onRejected = r => { throw r };
    }
})


Promise.prototype.catch = function (onRejected) {
    // then 方法本身就会调用 new Promise()
    return this.then(undefined, onRejected);
}
```

8.8 静态方法 resolve 和 reject

JavaScript 复制代码

```
// 3. 添加静态方法
Promise.resolve = function (value) {
    return new Promise((resolve, reject) => {
        if (value instanceof Promise) {
            // value 就是传入的 new Promise 的实例
            value.then(resolve, reject)
        } else {
            resolve(value);
        }
    })
}

Promise.reject = function (reason) {
    return new Promise((resolve, reject) => {
        reject(reason);
    })
}
```

8.9 静态方法 all

JavaScript | 复制代码

```javascript
// 最后的功能
Promise.all = function (pArr) {
    // 任务队列的数组
    const task = [];
    count = 0;

    // 返回一个实例
    return new Promise((resolve, reject) => {
        // 1. 数组遍历
        pArr.forEach((item, index) => {
            // 2. 取值
            item.then(
                v => {
                    // 3. 按顺序提供内容
                    task[index] = v;
                    count++;
                    // 4. 既然能走到成功的回调，证明异步也结束了
                    if (count === pArr.length) {
                        resolve(task);
                    }
                },
                r => reject(r)
            )
        })
    })
}
```

8.10 静态方法 race

JavaScript | 复制代码

```javascript
// race 方法
Promise.race = function (pArr) {
    return new Promise((resolve, reject) => {
        // 任务执行
        pArr.forEach(item => {
            item.then(v => resolve(v), r => reject(r))
        })
    })
}
```